Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information and Software Technology 48 (2006) 90–106
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Compacting XML documents

Miklós Kálmán*, Ferenc Havasi, Tibor Gyimóthy

Department of Software Engineering, Aradi vértanuk tere 1., H-6720 Szeged, Hungary

Received 5 October 2004; revised 27 February 2005; accepted 3 March 2005
Available online 12 April 2005

Abstract

Nowadays, one of the most common formats for storing information is XML. The biggest drawback of XML documents is that their size is rather large compared to the information they store. XML documents may contain redundant attributes, which can be calculated from others. These redundant attributes can be deleted from the original XML document if the calculation rules can be stored somehow. In an Attribute Grammar environment there is an analog description for these rules: semantic rules. In order to use this technique in an XML environment we defined a new metalanguage called SRML. We have developed a method, which enables us to use this SRML metalanguage for compacting XML documents. After compaction it is possible to use XML compressors to make the compacted document much smaller. By using this combined approach we could achieve a significant size reduction compared to the compressed size of the XML specific compressors. This article extends the method published earlier to provide the possibility of automatically generating rules using machine learning techniques, with which it can find relationships between attributes which might not have been noticed by the user beforehand.

Keywords: XML; SRML; XML compaction; XML semantics

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. The applicability of the XML format spans medical science (human genome mapping [7]), database storage [8], military use [10], component modelling [9]. 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). The size factor is also important when an XML document has to be transferred via a network. One solution to overcome this problem is to compress the documents using a general (e.g., zip) or XML compressor (XMill [6]). Unfortunately, the compressed size of the files may still be too large.

Compressors are the most effective when they can find the most dependencies in a set of data and utilize these dependencies in order to store the data in a smaller form. XML documents 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 [18] 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 rules of Attribute Grammars [2], and can be used for compacting the XML document by removing computable attributes.

The generation of these SRML files can be performed 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 have 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).

* Corresponding author. Tel.: +36 62 544143.
E-mail address: kalman@inf.u-szeged.hu (M. Kálmán).

During the testing of the implementation, the input XML files were compacted to 70–80% of their original size while maintaining further compressibility (e.g., the XMill XML compressor could compress the file after first being compacted making it even smaller). Keeping the compressibility of XML files is the main advantage of our method, apart from gaining a general understanding of the relationships between attributes, since the compacted and then compressed file will be smaller than the merely compacted file.

For testing our approach we used XML documents generated from large C++ programs. XML can be considered as a common format for information exchange between software development tools (e.g.: XMI in case of UML documents). This trend can be seen in the field of reverse engineering, where it is important for tools (e.g.: source analyzers, visual modelers, metric calculators [22], program analyzers [16]) to communicate with each other during the analysis of large 'legacy' systems. One of these is the Columbus [15] system, which is a widely used tool for the analysis of C++ programs. This system offers the opportunity of storing derived information in an XML format. The output of this system is an XML based file called CPPML, which contains detailed information about the C++ code that was analyzed and aids developers in the reverse engineering process. The size of CPPML documents can be quite large on real systems (e.g.: *StarWriter* containing 1,764,574 lines of code results in a 507 MB CPPML file). This is why applying the technique mentioned in this article is very important, since compacting CPPML documents using this technique, followed by XMill can cause the compressibility ratio to increase by 10% (of the original compressed size). This could make the new method a useful partner in future XML compressors. The method works using SRML rules. These rules can be generated by hand or by machine learning methods. The effectiveness of an SRML file created via machine learning can attain values similar to those of manual SRML generation. It is also possible to combine the two, making the compaction process more effective.

This article mainly focuses on the XML domain in terms of compaction, however, the methods described can also be applied to additional domains, for example databases or data warehouse data. If the data is stored in an XML format it can be directly applied without any alterations.

In the following sections, first some background knowledge will 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 functionality. The following section explains how the learning of SRML files is achieved and the advantages these rules offer. Afterwards, we show the efficiency of combining machine and manually generated rules and the efficiency of each learning module. Finally, we round off the paper by mentioning related works, a brief summary and topics for future study.

2. Preliminaries

In this section, a basic introduction to XML files will be given along with the necessary preliminaries for Attribute Grammars. This will be needed to better understand parts in the subsequent sections.

2.1. XML

The first concept that must be introduced is the XML format. A more thorough description of the XML documents can be found in [3] and [21]. XML documents are very similar to *html* files, as they are both text-based. The components in both are called *elements*, which may contain further *elements* and/or text, or they may be left empty. *Elements* may have attributes like the html anchor tag *a* attribute of *href* elements in html files. In Fig. 1 there is an example for storing a numeric expression in XML format. This example has an additional attribute called 'type', which stores the type of the expression. The values can be *int* or *real*.

It is possible to define the syntactic form of XML files. This is done through a DTD (Document Type Definition) file. This DTD file specifies the accepted format of the XML document. If an XML document uses a DTD all elements and attributes in the XML document must conform to the syntactic validity of the DTD. The DTD of Fig. 1 can be found in Fig. 2.

To analyze XML files they must be parsed. There are two ways of parsing XML files: one is based on the DOM (Document Object Model) tree, while the other is a sequential parser called SAX.

A DOM [5] tree is a tree containing all the tags and attributes of an XML document as leaves and nodes (Fig. 5(a) is the DOM tree of Fig. 1). This DOM tree is used by the XML processing library for internal data representation. The DOM is a platform- and language-independent interface that allows the dynamic accessing and updating of the content and structure of XML documents. When DOM tree parsing is used it makes the XML document handling easier, but it requires more memory to accomplish this, since it creates a tree of the XML in the memory. This method is quite effective on smaller XML documents.

```
<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).

```
<!ELEMENT num (#PCDATA) >
  <!ATTLIST num type ( real | int )#REQUIRED >
<!ELEMENT expr ( num | multextr | 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 >
```

Fig. 2. The DTD of the simple expression in Fig. 1.

The SAX parser can handle large input XML files, but since it is a file-based parser it can be quite slow, especially when trying to access attributes that are not in the current *element*. The memory requirements of this method is constant and not in direct proportion to the size of the input XML document.

2.2. Attribute grammars

Another key concept that should be mentioned is that of Attribute Grammars. Attribute Grammars are based on the context-free grammars. Context Free (CF) Grammars can be used to specify derivation rules for structured documents. A *CF Grammar* is a four tuple $G=(N,T,S,P)$, where N is the set of non-terminal symbols, T is a set of terminal symbols, S is a start-symbol and P is a set of syntactic rules. It is required that at the left side of every rule only one non-terminal can be present. Given a grammar, a derivation tree can be generated based on a specific input. The grammar described below specifies the format of a simple numeric expression described in Fig. 1.

N={ expr, multexpr, addexpr, num } S=expr T={“ADD”,”MUL”,NUM}

```
P:
(1) expr→num
(2) expr→multexpr
(3) expr→addexpr
(4) addexpr→expr "ADD" expr
(5) addexpr →expr "SUB" expr
(6) multexpr→expr "MUL" expr
(7) multexpr→expr "DIV" expr
(8) num→NUM
```

An *Attribute Grammar* contains a CF grammar, attributes and semantic rules. The precise definition of Attribute Grammars can be found in [2,14]. In this section, we will only mention those definitions which are required for better understanding later parts of this article.

An attribute grammar is a three tuple $AG=(G,AD,R)$, where

1. $G=(N,T,S,P)$ is the underlying context-free grammar.
2. $AD=(Attr,Inh,Syn)$ is a description of attributes. Each grammar symbol $X\in N\bigcup T$ has a set of attributes $Attr(X)$, where $Attr(X)$ can be partitioned into two disjoint subsets denoted by $Inh(X)$ and $Syn(X)$. $Inh(X)$ and $Syn(X)$ denote the inherited and synthesized attributes of X, respectively. We will denote the attribute a of the grammar symbol X by $X.a$.
3. R orders a set of evaluation rules (called *semantic rules*) to each production, as follows: Let p: $X_{p,0}\ldots X_{p,n_p}$ be an arbitrary production of P. An attribute occurrence $X_{p,k}.a$ is said to be a *defined occurrence* if $a\in Syn(X_{p,k})$ and $k=0$, or $a\in Inh(X_{p,k})$ and $k>0$. For each defining attribute occurrence there is exactly one rule in $R(p)$ that determines how to compute the value of this attribute occurrence. The evaluation rule defining attribute occurrence $X_{p,k}.a$ has the form: $X_{p,k}.a=f(X_{p,k_1}.a_1,\ldots,X_{p,k_m}.a_m)$.

The example in Fig. 3 shows an AG for computing the *type* of the simple expression described in Fig. 1. In the example, the ‘type’ of *addexpr* is *real* if the first or the second *expr* has a *real* type; otherwise it is *int*. If, we supplement the derivation tree with the values of attribute occurrences we get an attributed derivation tree.

Each attribute occurrence is calculated once and only once. The attributed derivation tree for the expression 3*(2.5+4) can be found in Fig. 5(b). The main task in AG is to calculate these attribute occurrences in the attributed derivation tree. This process is called attribute evaluation. There are several ways of evaluating the attributed tree. In the case of a simple attribute grammar the occurrences can be evaluated in one Left-to-Right pass, however, there can be more complex grammars where more passes may be required. The algorithm for a simple Left-to-Right pass evaluator can be seen in Fig. 4. This algorithm evaluates the inherited attributes of the root’s children recursively. When all inherited attributes have been evaluated then the synthesized attributes are processed.

```
(1) expr     -> num expr.type=num.type;
(2) expr     -> multexpr expr.type=multexpr.type;
(3) expr     -> addexpr expr.type=addexpr.type;
(4) addexpr  -> expr "ADD" expr
    addexpr.type=(expr[1].type=="real"|| expr[2].type=="real")? "real":"int";
(5) addexpr  -> expr "SUB" expr
    addexpr.type=(expr[1].type=="real"|| expr[2].type=="real")? "real":"int";
(6) multexpr -> expr "MUL" expr
    multexpr.type=(expr[1].type=="real"|| expr[2].type=="real")? "real":"int";
(7) multexpr -> expr "DIV" expr multexpr.type="real";
```

Fig. 3. Example for semantic rules.

```
function evaluate_L(node r) begin
   children = left_to_right_list_of_children_of(r)
   while not_empty(children)
   begin
     c = first(children)
     c_inh = inherited_attributes_of(c)
     evaluate_attributes(c_inh)
     evaluate_L(c);
     remove_first_from_list(children)
   end
   r_synth = synthetized_attributes_of(r)
   evaluate_attributes(r_synth)
end
```

Fig. 4. A simple Left-to-Right evaluator.

Attribute grammars can be classified according to the evaluation method used. The L-Attributed Grammars can be evaluated in a single Left-to-Right pass, while a multi-pass AG needs several Left-to-Right or Right-to-Left passes to evaluate all of its attributes.

2.3. The relationship between XML and Attribute Grammars

Examining the DOM tree and the attributed derivation tree in Fig. 5(a) and (b), it can be seen that XML documents are visibly similar to attributed derivation trees. There is an analogy between AG and XML documents. In Attribute Grammars, the Non-terminals correspond to the elements in the XML document. Syntactic Rules are presented as an element type declaration in the DTD of the XML file. An attribute specification in the AG corresponds to an attribute list declaration in the DTD. There is an important concept in Attribute Grammars which has no XML counterpart; semantic rules. It might be useful to apply these semantic rules in the XML environment as well.

This would be advantageous as the attribute instances and their values are stored directly in the XML document files. If it were possible to define semantic rules, it would be sufficient to store the rules applying to specific attributes, since their correct values could then be calculated. With the aid of this it would then be possible to avoid having to store those attributes, which could be calculated. In the future the definition of semantic rules will be an integral part of XML document files.

3. An approach for the compaction of XML documents

There are several possible approaches for defining semantic rules. One solution would be to use a DTD file. The problem with this is that the DTD cannot be expanded enough to store the rules in a structured format. The other problem with the DTD file is that the elements are defined using regular expressions making it rather hard to reference each item separately.

Another approach might be to introduce a new metalanguage which has its own parser. This is an ideal solution since it provides the freedom of adding complex semantic rules. These semantic rules have to be stored somehow, since we are using an XML environment it seems self-evident to store the semantic rules in an XML based format as well.

3.1. The SRML metalanguage

An XML-based metalanguage called *SRML* (*Semantic Rule Meta Language*) has been defined to describe semantic rules [18]. The DTD of SRML files can be seen in Appendix A.

The exact meaning of each SRML element defined in the DTD file can be seen in Appendix B. We will provide an example rule set to demonstrate the advantages of using the SRML language. The rule set in Fig. 6 defines the same semantic rules as those mentioned in Fig. 3, the example only covers the *addexpr* type calculation, as all other elements can be calculated using similar rules.

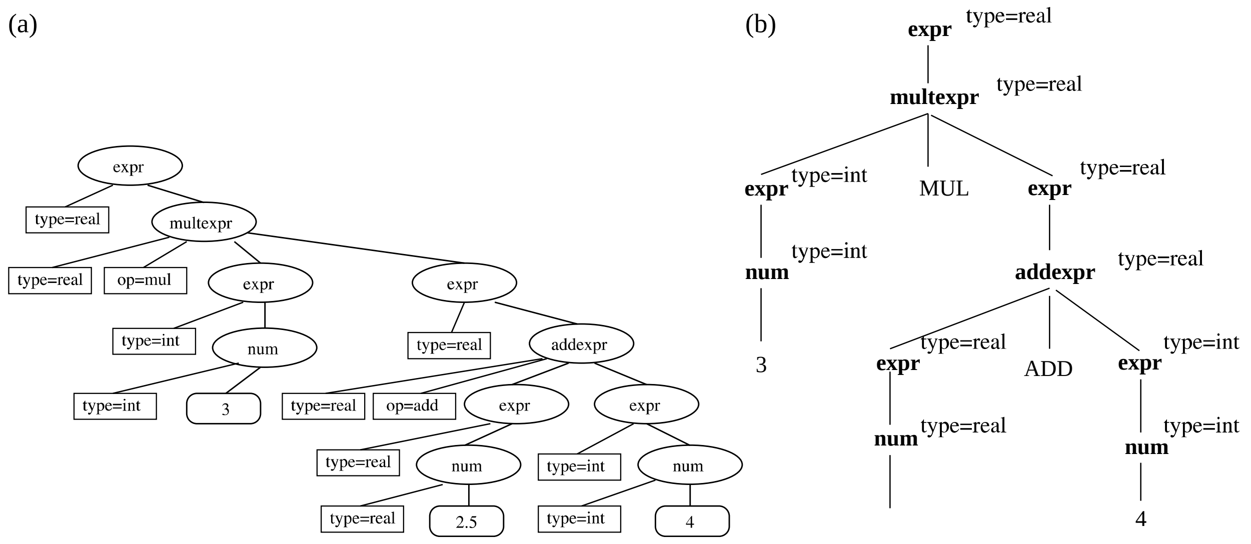

Fig. 5. (a) XML document DOM tree and (b) attributed derivation tree.

```
AG rule:
(4) addexpr -> expr "ADD" expr
addexpr.type=(expr[1].type=="real"|| expr[2].type=="real")? "real":"int";

SRML rule:
<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>
```

Fig. 6. An SRML example for 'type' attribute of the addexpr element.

There are a few comments that should be made about Fig. 6. The first is that the rule set is much bigger than that for AG rules. The reason is that this is already in an interpreted form, whereas the AG rules have to be interpreted first. Another thing that has to be explained is the 'from' and 'num' attributes in the <attribute/> tag. Although, the complete list of attributes is mentioned in Appendix B we will explain these two attributes, as they are important for this example. The 'from' attribute specifies which direction in which the attribute value is taken from, and 'num' is the offset number. So the following term means that we are referring to the 'type' attribute of the first *expr* element:

```
<attribute element="expr" from="begin" num="1" attrib=
"type"/>
```

If we apply the rule set described in Fig. 6 to the input XML shown in Fig. 3, also including the additional rules, which are very similar. The compacted XML can be seen in Fig. 7.

The size of this XML is considerably smaller than the input (Fig. 1). The only 'type' attribute that was kept was the *num* element's "type" attribute. The reason was that, if we have the 'type' of *num*, then we can calculate the *expr* 'type'. Once we have the *expr* 'type' the *addexpr* and *multexpr* 'type' attributes can be calculated as well. This is why there is no sense of storing all the types if only the *num* 'type' is needed.

An SRML definition has to be consistent. This means that an attribute instance can only have one corresponding rule. In SRML an attribute is either inherited (Fig. 8(a)) or synthesized (Fig. 8(b)) like that of Attribute Grammars, hence in the SRML language it is not permitted to have two separate rules for the same attribute.

3.2. *Evaluating attributes using SRML*

Since the semantic rules, inherited and synthesized attributes were defined in the XML (SRML) environment it is now possible to port the single and multiple pass evaluation algorithms into this new environment. Moreover, it is now possible to define SRML classes based on the AG classes.

For example, in the AG environment there is a language class (L-Attribute Grammar) which can be evaluated in one Left-to-Right pass (Fig. 4).

Based on an SRML definition it is straightforward to determine whether an attribute is synthesized or inherited (as mentioned earlier). An SRML definition is an L-SRML definition if:

- the root only defines synthesized attributes and the evaluation rules can only contain its inherited attributes and/or any attributes of its children.
- the children only define inherited attributes and the evaluation rules can only contain the root's inherited attributes and/or attributes of siblings to the left of the child.

Since this class can be evaluated in one pass, it is not necessary to keep the whole XML document in the memory

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

Fig. 7. Compacted expression XML.

(a)
```
<rules-for root="A">
 <rule element="B" attrib="x">
  <expr>
   <attribute element="srml:root" attrib="x"/>
  </expr>
 </rule>
</rules-for>


Rule: B.x = A.x
```

(b)
```
<rules-for root="A">
 <rule element="srml:root" attrib="y">
  <expr><binary-op op="add">
   <attribute element="srml:root" attrib="x"/>
  </expr>
  <expr><data>3</data></expr>
  </binary-op></expr>
  </rule></rules-for>


  Rule: A.y = A.x+3
```

Fig. 8. (a) An inherited attribute (B.x) and (b) a synthesized attribute (A.y).

to evaluate all the attributes. Moreover as there is only one pass the parsing and evaluation can be done using SAX. If the evaluation requires more than one pass it is sensible to keep the whole XML document in the memory (e.g., using the DOM interface). Using the DOM interface will make the parsing and evaluation faster, but it is at the expense of greater memory usage.

3.3. Compacting/decompacting

Using the evaluation methods outlined earlier, it is now possible to define how they can be used in the compaction of XML documents. Before going into detail on how the method is built up some definitions are needed for Fig. 9.

The 'Complete XML Document' refers to the document in its original size and form. The 'Reduced XML Document' is the output of the compaction, which of course has a smaller size relative to the original. The 'Semantic Rules' are the SRML rules used to compact and decompact the XML document. In Fig. 9 it is clear that the *decompacting (complete)* procedure expands the document and recreates the original XML document.

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. It is important to note that if the semantic rule for a given attribute does not give the correct value the attribute is not removed, thus maintaining the document's integrity.

After the XML document has been compacted it can then be compressed using a traditional compressor like gzip or XML compressors like XMill. An example of a compacted XML is shown in Fig. 7.

The file containing the SRML description is an extension of the DTD, thus maintaining the XML format. If the XML files were to be compressed instead of compacted this possibility would be lost. One can say that this method of compacting makes the XML format lose its self-describing ability, however, the size reduction we gain by removing attributes and the fact that it still is readable by any XML viewer and can be further compressed by any XML compressor makes this sacrifice worthwhile.

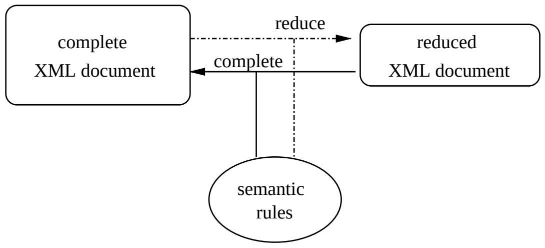

Fig. 9. The compacting/decompacting method.

4. SRMLTool: a compactor/decompactor for XML documents

We have implemented a tool for the compaction of XML documents. This tool uses SRML to store semantic rules, 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 can be a reduced document (some attributes are removed) or a completed document (the missing attributes with corresponding rules have been calculated and restored). Fig. 10 shows the modular structure of the tool.

The implementation uses a general evaluator. Currently attribute occurrences are evaluated depth first using Left-to-Right passes. This is a multi-pass algorithm where the algorithm described in Fig. 4 is executed several times. During each pass a separate attribute occurrence is evaluated. 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

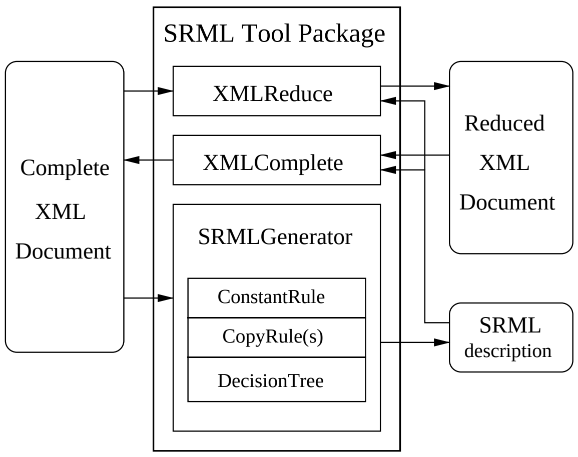

Fig. 10. SRMLTool package.

routine that examines every node and every attribute. The purpose of this examination is to find out which attributes have corresponding SRML rules. Unfortunately this approach has high resource requirements. A way of optimizing it is to allow only L-SRML rules (see Section 3.2) and use a SAX parser (see Section 2.1).

4.1. The reduce algorithm (compacting)

This algorithm removes those attributes that can be correctly calculated using the SRML semantic rules. Every attribute can have only one rule in the SRML rule set. If there are more rules for the same attribute the first one will be used. There is a possibility that the attribute cannot be correctly calculated, which means that the value of the attribute differs from the value given in the rule. In this case the attribute will not be removed, since the value is different. 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[1] 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. Consider the following SRML rule set:

```
<rules-for root="A">
   <rule element="srml:root" attribute="x">
      <expr><data>100</data></expr>
   </rule>
   <rule element="srml:root" attribute="y">
      <expr><data>200</data></expr>
   </rule>
</rules-for>
```

Suppose the input XML document is the following:

```
<calc>
   <A x="100" y="200"/>
   <A x="99" y="200"/>
   <A x="88" y="201"/>
   <A/>
</calc>
```

When applying the rules to the input XML document the result of compaction (*Reduce*) will be the following:

```
<calc>
   <A/>
   <A x="99"/>
   <A x="88" y="201"/>
   <A srml:var=" x y "/>
</calc>
```

[1] Non-defined attributes can occur if there are IMPLIED attributes in the XML file.

The example shows that in the first *A* element both *x* and *y* attributes were removed as their value matches the SRML rule value. In the second only the *y* attribute could be removed since the *x* attribute had a different value. In the third *A* element both *x* and *y* attributes were kept as their value differed from the SRML rule values. In the last *A* element a new attribute called *srml:var* had to be inserted because the input XML document had neither attribute in this element. If the attributes had not been marked then the decompacting would have added both attributes with the value mentioned in the SRML rules.

Our implementation can reduce XML documents even when there is a circular reference in the rules. For example, if the rules $A.x=B.x$, $B.x=C.x$ and $C.x=A.x$ (Fig. 12) 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. The following algorithm is used to remove the attributes (this algorithm can also resolve circular dependencies):

1. Create a dependency list (every attribute may have input and output dependencies, the input dependencies being those attributes on which it depends and the output are those that depend on it. The dependencies are represented as edges).
2. If the list is empty then goto 5. Look for an attribute that has no input or output edges. If there is one then it can be deleted and removed from the list (since this can be restored using the rule file), then goto 2. If there is not any goto 3.
3. Look for an attribute that has only output edges. This means that other attributes depend on this attribute (e.g.: the *A.x* attribute in Fig. 11). Delete this attribute and the output edges remove it from the list then goto 2. If there is not any, goto 4.
4. Check for circular references. This is the last case possible since if all the attributes remaining in the dependency list have both input and output edges it means that the attributes are transitively dependent on each other (Fig. 12). Select the first element in the list (this will serve as the base of the circular reference). Keep this attribute and remove it from the list (keeping it means that it is added to a *keep* vector). Goto 2.
5. END

The algorithm always terminates since the dependency list is always emptied. The algorithm has a vector, which contains those attributes that will not be removed. Note: This algorithm is not the most optimal solution, but it is

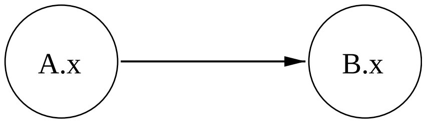

Fig. 11. Simple dependency.

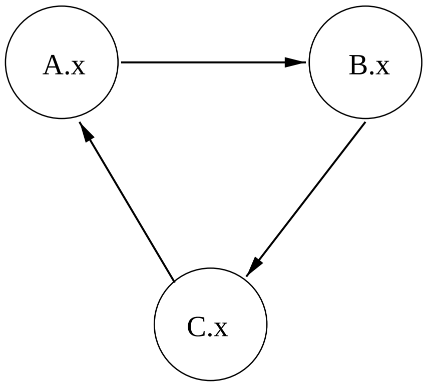

Fig. 12. Circular dependency.

reduction safe and the completion process can be performed without any losses.

4.2. The complete algorithm (decompacting)

This algorithm restores the attributes which have corresponding SRML semantic rules. The attributes which were marked with the *srml:var* attribute will not be restored (see Section 4.1). In contrast to the traditional evaluator where we define which attribute is evaluated in which pass our approach tries to evaluate every attribute occurrence during each pass using an attribute occurrence queue. As mentioned in Section 4, a DOM tree is built from the XML file. After this an inorder tree visitor is called to find out which attributes have corresponding SRML rules. If an attribute having an SRML rule is found it is stored in a vector that is later processed. This 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, therefore, he cannot provide SRML rules. The input of the module is the XML document, which needs to be analyzed. The output will be a set of rules optimized for the input, which enable the compaction. 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 a defined interface.

The process of learning is as follows:

1. Read the input XML file.
2. Enumerate the plug-in algorithms and execute them sequentially.

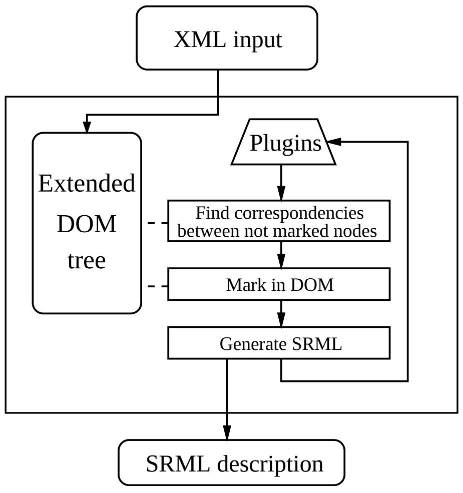

Fig. 13. Learning SRML rules.

3. After a plug-in has finished 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.
4. If all plug-ins have been executed then write the output SRML rule file, otherwise continue processing the XML file.

Fig. 13 shows the process of learning. The generated rules are concatenated one after each other, creating a single SRML rule file at the end of the process.

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, which may not have been visible to the user previously. In this case of course the SRML file has to be created dynamically via machine learning and other statistical methods. The SRML files created by machine learning can be used as 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.

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. The execution order of the plug-ins is crucial, since the order defines the efficiency of the compaction and also the execution time. During our testing we found the following order to be the most effective (an observation based on experiment):

1. SRMLCopyChildRule
2. SRMLCopyAttribRule
3. SRMLCopyParentRule
4. SRMLDecisionTree
5. SRMLConstantRule

In the following subsections we will describe each learning module in detail.

5.1.1. SRMLConstantRule

This is the simplest learning algorithm in the package. This algorithm uses statistical analysis to retrieve the number of attribute occurrence values and then decides whether to make a rule for it. For instance this algorithm searches for $B.x=4$ 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 Fig. 14(a)).

The decision is based on whether the size of the new rule would be bigger than that of the size decrease achieved by removing the attributes. The tree in Fig. 14(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: the *attribName.value* of *elementName* (synthesized case) and one of *parentElementName* (inherited case).

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

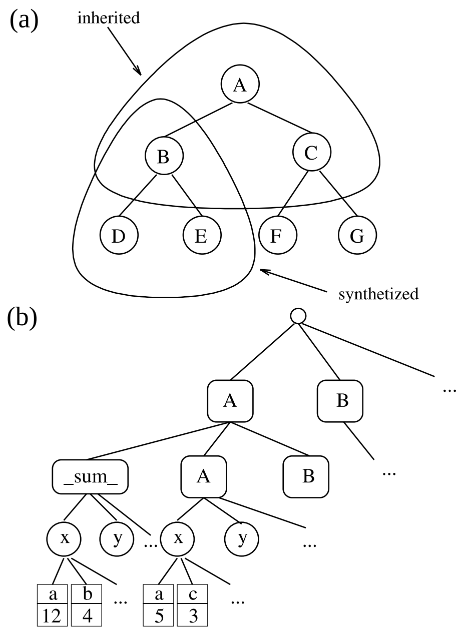

Fig. 14. (a) The two contexts of a node and (b) the statistical tree of the SRMLConstantRule.

5.1.2. 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 can be rather 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 mentioned above. The algorithm determines whether the attribute was inherited or synthesized. Inherited attributes are handled by the *SRMLCopyParentRule* module, whereas synthesized attributes are handled with the *SRMLCopyChildRule*, *SRMLCopyAttribRule* modules. In Fig. 15 examples can be seen for each type. In the case of (1), the *book.firstSection* is equal in value to the *section.name* making this attribute synthesized. In (2), the *word* attribute of the book matches the total attribute, thus making it a synthesized attribute as well. In (3), *section.author* is the same as the *book.author* this is an inherited attribute. In the example, there are parts where the attributes do not match. This is permitted since it is not obligatory that every attribute with the same name match.

5.1.3. SRMLDecisionTree

The *SRMLDecisionTree* plug-in is by far the most advanced of the currently implemented algorithms. It makes use of machine learning approach [17] in order to discover relationships and builds if-else decisions using a binary tree similar to the ID3 [4] algorithm. Next, we will go

```
1. SRMLCopyChildRule XML: <book firstSection="Introduction">
  <section name="Introduction" word="200"/>
  <section name="Summary" word="400"/>
</book>

SRML:
<rules-for root="book">
 <rule element="srml:root" attrib="firstSection">
  <expr><attribute element="section" attrib="name"/></expr>
 </rule>
</rules-for>

2. SRMLCopyAttribRule XML:
<book word="200" total="200">
 <section word="200"/>
</book>

SRML:
<rules-for root="book">
 <rule element="srml:root" attrib="word">
  <expr><attribute element="srml:root" attrib="total"/></expr>
 </rule>
</rules-for>

3. SRMLCopyParentRule XML:
 <book author="James West">
 <section author="James West" title="Introduction">
 <section author="Mary Jones" title="Summary">
</book>

SRML:
<rules-for root="book">
 <rule element="section" attrib="author">
  <expr><attribute element="srml:root" attrib="author"/></expr>
 </rule>
</rules-for>
```

Fig. 15. Examples for the Copy rules.

into detail on how the *SRMLDecisionTree* plug-in actually works.

The algorithm first narrows the set of attributes for which rules need to be made. The criteria of this 'narrowing' is that the given attribute must have at least two different values whose occurrences are *dominant*. The meaning of *dominant* in this case is that the attribute value occurs frequently enough in the input, making it an excellent target for compaction. The operation of finding *dominant* attributes is done though a function which can be changed. Currently the algorithm considers an attribute value *dominant* if the value occurrence exceeds 5. Although we defined the dominance limit to 5 it does not necessarily mean that this is a hard limit. The dominance function can be replaced at will. This will be a basis for future study. If the number of *dominant* attribute values is less than two it is not a problem since the next plug-in algorithm in this case the *SRMLConstantRule* will try to find a rule for that attribute. The reason why the attribute value number is limited to two is that with two different dominant values it is possible to create if-else branches making a more optimal rule generation possible.

We generate a learning table for every 'narrowed' *dominant* attribute. To demonstrate how these tables are created and what their contents are a simple example will be provided. The example creates a car-pool database for cars. Each car can have the following attributes: color, ccode (color code), doors, type. The XML format of the example is shown in Fig. 16.

Based on the input XML file described in Fig. 16 two learning tables will be generated, assuming that there are enough dominant attributes. At this stage of the processing it is not known whether the attribute is synthesized or inherited, so learning tables are generated for both contexts. For the synthesized case (Fig. 17) only one table is created, but for the inherited case (Fig. 18) the number of tables depends on how many parents the attribute's element had in the whole input file.

The headers of the columns are the names of the attributes which are present in the rule contexts (inherited, synthesized) of the current attribute. If an attribute occurs more than once in the rule contexts a new columns is appended for every new attribute occurrence.

The number of rows depend on how many rule contexts there are containing this attribute. The values of the fields are the values that the attribute takes in the given context. If the attribute is not present in a context the value is marked by a minus (−) sign.

The learning tables created by this algorithm can be used as inputs for external learning systems. The learning algorithm builds up a binary tree similar to that of the ID3 algorithm with a depth limit of 3. The reason why we use a binary tree is that the current specification of the SRML format only handles if-else statements and it would be rather difficult to code a multiple branch tree.

With every table, a new rule is learned for the given attribute. After this the algorithm decides (based on the generated tree) whether to select the synthesized rule or some of the inherited rules or neither. This decision is made using the information stored in the leaves of the tree. This information helps the algorithm effectively decide how many attributes will be eliminated during compaction if the given rule is added to the rule set. The algorithm will select those rules which achieve the maximum compaction.

At the end of the algorithm, the SRML rules are generated from the selected trees and the appropriate attributes are marked in the input DOM tree. In the case of Fig. 16 the learned rule set will be a relationship between the *ccode* (numeric) and the *color* (text) attribute.

NOTE: Every plug-in tries to make the optimal decision, the only factor that is currently not considered is the length of the *srml:var* attribute. This is why in some cases the SRMLConstantRule increases the size of the compacted file. The framework detects when a size increase occurs and does not execute the specific learning module.

.find[color]	ccode	doors	type
blue	5	5	minivan
red	3	3	compact
green	2	5	standard
black	1	3	convertible
blue	5	5	standard
blue	5	3	compact
green	2	3	convertible
...			...

Fig. 17. The synthesized learning table of Fig. 16.

```
<car-set num="100">
 <car color="blue" ccode="5" doors="5" type="minivan"/>
 <car color="red" ccode="3" doors="3" type="compact"/>
 <car color="blue" ccode="5" doors="5" type="standard"/>
 <car color="green" ccode="2" doors="5" type="standard"/>
 <car color="black" ccode="1" doors="3" type="convertible"/>
 <car color="blue" ccode="5" doors="3" type="compact"/>
 <car color="green" ccode="2" doors="3" type="convertible"/>
 ...
</car-set>
```

Fig. 16. An example for a decision tree input.

.find[car.color]	car.ccode	car.doors	car.type.1	...
blue	5	5	minivan	...
red	3	3	compact	...
green	2	5	standard	...
black	1	3	convertible	...
blue	5	5	standard	...
blue	5	3	compact	...
green	2	3	convertible	...
...				...

Fig. 18. An inherited learning table of Fig. 16.

6. Experimental results

In the field of Software Engineering the XML representation is considered typical, therefore, it is widely used (e.g.: XMI based models). The testing of our implementation was done via CPPML [15], an XML exchange format that is used as an output of the Columbus Reverse Engineering package [12], however, this method can be applied to any XML domain without major restrictions.

6.1. A real sized case study: CPPML

CPPML files can be created from a CPP file. CPPML is a metalanguage capable of describing the structure of programs written in C++. Creating CPPML files can be done via the Columbus Reverse engineering package [12] (CPPML is XML based).

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

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

The CPPML form of the program can be the following:

```
<class id="id20097" name="_guard" path="D:\SymbolTable\-
CANGuard.h" line="71"
   end-line="90" visibility="global" abstract="no"defi-
   ned="yes" template="no"
   template-instance="no" class-type="class">
      <function id="id20102" name="registerConstruction"
      path="D:\SymbolTable\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:\SymbolTable\CANGuard.h">
         <return-type>void</return-type>
         <parameter id="id20106" name="ti" path="D:\Sym-
         bolTable\CANGuard.h" line="74"
         end-line="74" const="yes">
         <type>type_info&amp;</type>
         </parameter>
      </function>
   ...
</class> ...
```

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. This *kind* can be *normal*, a *constructor* or *destructor*. If the function name matches that of the class name then it is a *constructor*, if the function name starts with a ∼ 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" ele-
                  ment="srml:root"/></expr>
               </binary-op>
            </expr>
            <expr><data>constructor</data></expr>
            <expr>...</expr>
         </if-expr>
      </expr>
   </rule>
</rules-for>
```

It is not necessary to provide precise rules, since the *Compact* algorithm will only remove those attributes which can be correctly calculated from the rules. 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 previous function's implementation.
3. The parameters of a function are usually in the same file, perhaps somewhere in the same line.
4. The visibility of the class members usually are the same as that for the previously defined members.

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:\SymbolTable\-
CANGuard.h" line="71"
   end-line="90" visibility="global" abstract="no" defi-
   ned="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:\SymbolTable\CANGuard.h">
         <return-type>void</return-type>
         <parameter id="id20106" name="ti" line="74"
         const="yes">
            <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 compacted was 1.502 bytes. This ratio can be further improved with the introduction of new SRML rules.

6.2. Compacting CPPML with SRML rules created by hand

We have implemented the rules in SRML mentioned in the previous section and applied them to 3 different sized examples. These are:

symboltable (399 KB):	one of the source files of the Columbus system
jikes (2233 KB):	the IBM Java compiler
appwiz (3546 KB):	a base application generated by Microsoft Visual Studio's AppWizard

The results achieved using SRML files created by hand are shown in Fig. 19. Input files were in CPPML form. The (C) bracket indicates that the compressors were applied to the compacted version of the XML file.

Although, the compaction ratio is smaller than the that achieved by compression, but when compaction and compression are combined the method is able to increase the efficiency of the compressor. The result of using XMill on the compacted SymbolTable resulted to an overall size reduction of about 10%, since XMill compressed the original SymbolTable to 19,786 bytes, whereas applying XMill after the file had been compacted resulted in a size of 18,008 bytes. This 10% efficiency increase can be very useful for embedded systems. This also extends the applicability of our method to more areas.

Manual rule generation is not a hard task for a domain expert who knows the system he wishes to apply the method to, since he knows the attributes and this can create the appropriate rule sets. Creating hand written rules for the CPPML environment took less than 1 h.

6.3. Compacting CPPML with machine learning SRML rules

In Fig. 20, a comparison is made between the efficiency of the machine learned and hand-generated SRML rules. The percentage values shown in the *Diff* field show how big the difference was compared to the original file size.

In some cases, the compaction ratio achieved using SRML files generated via machine learning can attain those of hand-generated SRML files (e.g.: *Jikes*). However, creating hand-generated rules requires time and effort, since the user needs to know the structure of the XML file. The machine learning approach takes this burden off the user and generates rules automatically. These results can be improved by adding new plug-in algorithms into the *SRMLGenerator* module like those using advanced decision making systems, other forms of statistical methods and concatenation parsers (which searches for relationships between concatenated attribute values).

Using Machine Learning to generate rules can be costly, however, it is enough to generate the rules once, then they can be reused over time for each XML file in the given domain.

Since, the execution order of the plug-ins really matters (an effective order was described in Section 5.1) that is why using the Copy rules (*SRMLCopyChildRule, SRMLCopyAttribRule, SRMLCopyParentRule*) seems initially to provide the optimal solution, an observation based on experiment. First the copy rules are processed. The order of these if actually not important, but they offer simple relationships which cover more attribute occurrences. The reason why using the SRMLDecision-Tree plug-in is applied before last is that it takes a long time to process large files and is specialized for more complex and specific occurrence discovery and most attributes can be removed beforehand using other plug-ins. Using *SRMLConstantRule* in the end is useful, since it may remove and constant occurrences that were left untouched by the previous plug-ins. It is possible that in

File	SymbolTable	Jikes	AppWiz
Original	399 321	2 233 824	3 547 297
gzip [ratio]	30 460 [7.62 %]	177 051 [7.92 %]	244 174 [6.68 %]
XMill [ratio]	19 786 [4.95 %]	114 275 [5.11 %]	145 738 [4.10 %]
Compacted [ratio]	296 193 [74.10 %]	1 736 267 [77.70 %]	2 238 308 [63.10 %]
gzip(C) [ratio]	26 308 [6.58 %]	160 609 [7.18 %]	206 522 [5.82 %]
XMill(C) [ratio]	18 008 [4.50 %]	108 458 [4.85 %]	134 217 [3.78 %]

Fig. 19. Compaction table using hand written rules.

Filename	Manual	Machine	Diff
SymbolTable (399 321)	296 193 [74.10 %]	313 873 [78.60 %]	17 680 [4.42 %]
Jikes (2 233 824)	1 736 267 [77.70 %]	1 737 872 [77.79 %]	1 605 [0.07 %]
AppWiz (3 547 297)	2 238 308 [63.10 %]	2 589 526 [73.00 %]	351 218[9.90 %]

Fig. 20. Comparison of machine learned and hand written rules.

some cases it would be more optimal to choose another order, however, this can be a basis of future studies.

6.4. Analyzing the learning modules

In this section, we list the efficiency of each learning module in the sequence defined in Section 5.1. A comparison is made by building up the rule set, adding one rule type at a time, then noting the compaction ratio it achieves. This provides a good basis for future amendments. The results are shown in Fig. 21. The sizes include the size of the generated SRML file as well. The defined order is listed in Section 5.1.

As is clear from the table, during the comparison *SRMLConstantRule* decreased the efficiency in *Jikes* and *AppWiz*. The reason for this is that the current cost function does not take into account the case when an attribute is absent in an element. In this case the compactor has to mark this attribute with the *srml:var* attribute (see Section 4.1). If there were many 'missing' attributes it is possible that the file size might increase. The framework detects this and does not apply the given learning module. This in many cases is quite effective. The *SRMLGenerator* module saves the SRML files-after each plugin has been executed- to separate temporary SRML files. This is good for checking the efficiency of each plug-in. This option can be disabled using a command line parameter.

6.5. Combining machine learning and manual rule generation

It is possible to combine machine learning and manual rule generation into a single rule set. This is useful when the user knows some relationships between attributes, but not all. The module accepts preliminary SRML files as well, meaning that there are some rules in the file but not all. This file is processed and new rules are appended, making it more efficient. Below the efficiency of this method is shown in a table format.

The table in Fig. 22 shows that combining machine learning with manual rule generation is quite effective. When running XMill on the compacted XML created with the combined rules an increased compression ratio can be achieved (Fig. 23). When combining machine learning and manual rule generation the additional compressibility of the compacted document can be much as 26%. For example in Fig. 23 AppWiz was compacted to 145,738 bytes, which is 4.1% of the original document. If we first compact the XML using combined rules, then execute the *XMill* on it, the end file size is 106,773 which is 3.01% of the original and the overall file size decrease is 26.73% (this is the size difference between the manually generated then compressed and the combined generation and compressed file). Since, the whole system is based on a plug-in framework it is easily extendible by more effective machine learning plug-ins.

6.6. Resource requirements of the tool

For testing, a Debian Linux environment was used on a PC (AMD Athlon XP 2500+512 MB DDR). The package requires about 200 MB of memory since the DOM tree takes up a lot of space (this point is mentioned in Section 4). The *AppWiz* file was *compacted* in approximately 2 min and decompacted in 30 s. The execution time was long in the case of machine learning (*SRMLDecisionTree*) since it had to generate lots of learning tables, which can be rather large at times. Depending on the complexity and level of recursion, the execution time ranged from 2 to 30 h.

It is true that learning compacting rules is not very fast, however, once the rules are generated they can be reused over and over again. It is also possible to create a more effective compacting implementation to increase speed even more. In Section 3, we mention new methods for increasing speed. One of these alternatives would be to use SAX (see Section 2.1).

Trying to determine the running time of the method is not easy. It can be only estimated: The decision tree learning algorithm is a standard ID3 algorithm. This algorithm is applied to every attribute, which has at least two dominant values. The size of the learning table depends on all of the attribute occurrences and their environments. This is the reason why it would be hard to categorize the running time

Module	SymbolTable	Jikes	AppWiz
original	399 321	2 233 824	3 547 297
1	359 964 [90%]	2 022 497 [90%]	2 959 427 [83%]
1,2	351 999 [88%]	1 938 549 [86%]	2 889 183 [81%]
1,2,3	346 830 [86%]	1 871 154 [83%]	2 753 721 [77%]
1,2,3,4	337 825 [84%]	1 737 872 [77%]	2 589 526 [73%]
1,2,3,4,5	313 873 [79%]	1 821 228 [82%]	2 773 946 [78%]

Fig. 21. Comparing learning modules.

File	Manual	Machine	Combined
SymbolTable (399 321)	296 193 [74.17 %]	313 873 [78.60 %]	281 088 [70.40 %]
Jikes (2 233 822)	1 736 285 [77.72 %]	1 737 872 [77.79 %]	1 367 244 [61.20 %]
AppWiz (3 547 297)	2 238 308 [63.09 %]	2 589 526 [73.00 %]	2 038 569 [57.46 %]

Fig. 22. Combining machine learning and manual rule generation.

into standard cubic/quadratic classes, however, it strongly depends on the input file size and the complexity of the relationships contained within.

7. Related work

The first notion of adding semantics to XML documents was published in [1]. The authors furnished a method for transforming the element description of DTD into an EBNF Syntactic rule description. It introduces its own SRD (Semantics Rule Definition) comprised of two parts: the first one describes the semantic 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 their article and ours is that we provide semantic rules not just for newly defined attributes but also for real XML attributes. Our approach makes the SRML description an organic part of XML documents. This kind of semantic definition could offer a useful extension for XML techniques. We can also generate the SRML files using machine learning. 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 attribute 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.

The original idea of using SRML as a basis for compaction was introduced in [18], there a standard hard coded example was provided to show how compaction can be done. Later we published a University Technical Report [13] where the method was implemented in a JAVA environment, making the compaction possible on other input files as well. This new article focuses on extending these publications even more by extending the SRML language and adding the support for automatically generated rules, making it an even more effective compaction tool for XML files.

We have no knowledge of any research which would generate rules for XML. We have found a publication which generates rules for Attribute Grammars, which is introduced in [11]. The idea is to provide a way of learning attribute grammars. The learning problem of semantic rules is transformed to a propositional form. The hypothesis induced by a propositional learner is transformed back into semantic rules. AGLEARN was motivated by ILP learning and can be summarized in the following steps:

(i) the learning problem of the semantic rules is transformed to propositional form;
(ii) a propositional learning method is applied to solve the problem in propositional form;
(iii) the induced propositional hypothesis is transformed back into semantic rules.

This method is similar to ours since it learns and uses semantic rules based on examples as training data, but it is only effective on attributes with very small domains. In contrast to our method, it searches for precise rules which can use approximated rules as well.

The reason why these papers are cited here is that they in some way try to accomplish what our algorithm accomplishes. They mostly use semantic relations, and some even define a separate language (XML based) to describe the operations. It should be emphasized that our algorithm is not a compressor it is a compactor. We do not wish to compare our algorithm to a compressor algorithm, but if we apply a compressor to the compacted document, then we can achieve better results than other standalone compressors. Furthermore, if we generate the SRML file for a group of specific XML documents then it is not necessary to generate SRML rules for each input XML document in that group. This makes the applicability more feasible.

In the next part of this section some compression algorithms will be shown, since they can be combined with our approach, making their overall compression more effective.

In article [20], a Minimum Length Encoding algorithm is used. The algorithms operate in a breadth-first order, considering the children of each element from the root in turn. The encoding for the sequence of children of the element with name *n* in the document is based on how that

File	Original	Manual	Combined
SymbolTable 399 321	19 786 4.95 %	18 008 4.50 % 8.98 %	17 876 4.47 % 9.70 %
Jikes 2 233 822	114 275 5.11 %	108 458 4.85 % 5.09 %	92 102 4.12 % 19.40 %
AppWiz 3 547 297	145 738 4.10 %	134 217 3.78 % 7.90 %	106 773 3.01 % 26.73 %

Fig. 23. XMill compression for combined and manual compaction.

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

sequence is parsed using the regular expression in the content model for *n* in the DTD. This algorithm is a DTD centered approach.

Another article that ought to be mentioned is [19], where the authors create a multi-channel access to XML documents. An XML query processor is used to accomplish the structural compression. Lossless and lossy semantic compressors are defined along with an XML-based language for describing possible associations between similar parts of documents and semantic compressors. The main idea is to process the XML document in such a way that elements can be regarded as tuples of a relation. These tuples are structured as a datacube, with aggregate data on suitable dimension intervals. This is a relational approach, which considers the XML attributes as relations (formal expressions). The drawback is that it is a lossy compression.

XMill [6] is an open source research prototype developed by the University of Pennsylvania and ATandT Labs, building on the gzip library to provide an XML-specific compressor. It uses a path encoder to support the selective application of type-specific (i.e., based on design knowledge of the source document) encoders to the data content. The data and content of the source document are compressed separately using redundancy compression. The idea is to transform the XML into three components: (1) elements and attributes (2) text, and (3) document structure, and then to pipe each of these components through existing text compressors. This is a compressor as well, mostly used by other algorithms. It makes use of LZW as a compression algorithm.

The idea behind [23] is to use multiplexed hierarchical PPM (prediction by partial match) models to compress XML files. The approaches detailed in the article can in some cases better then XMill. The reason is that XMill's base transformation has drawbacks like precluding incremental compressed document processing and requires user intervention to make it efficient. It mentions that MHM (multiplexed hierarchical modelling) can achieve a compression ratio up to 35% better than the other approaches, but it is rather slow. In the future we are also planning to combine MHM with our approach making a more effective tool.

Another interesting approach for XML compression is described in [24]. This article describes how this tool can be used to execute queries against the compressed XML document. It uses several compression methods like Meta-data compression, Enumerated-type Attribute Value compression, Homomorphic compression. Using these techniques it creates a semi-structured Compressed XML document, which is binary, but retaining the query capabilities. It creates a frequency table and a symbol table, which are passed on to the XGrind kernel. This approach is better than using completely binary chunks of data, since it can query the file. Our approach can also be modified to have query capabilities, this is planned in the future.

8. Summary and future work

One of the most common problems associated with XML documents is that they can become rather large. Our method can make their compression more efficient. This method is based on the relationship between Attribute Grammars and XML documents. Using the SRML metalanguage a 20–30% size reduction 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. It can also generate rules. These rules can be used for compacting a document or for 'understanding' it (e.g. Data Mining) and decompact XML files using rules written by domain experts or generated by machine learning methods.

In [18], the SRML language was defined with an example program for compacting/decompacting CPPML files using rules written by hand. Our approach, however, has made it possible to make this compaction more universal, making it utilizable in several independent fields. The machine generation of SRML files was also introduced to take the burden off the user's shoulders. It is also possible to combine machine generated SRML files with hand written rules making it very efficient. This package can also become a very effective XML compressor supplement, since the combined approach is able to produce smaller compressed documents than the original XML compressor.

In the future, we plan to introduce new plug-ins to our framework in order to further increase the efficiency. The new plug-ins will be able to learn more complex functions like concatenation, multiplication (e.g., $A.x = B.x{*}C.x$) and addition (e.g. $A.x = B.x + C.x$). We also plan to optimize the current plug-ins like *SRMLConstantRule*. The analysis of the plug-in execution order is also planned and a more effective dominance function. Our method will also be implemented based on SAX so that it will have the ability to handle large files (of course this will lead to a restriction on the type of rules that can be learned). We are also planning a demand-driven query interface for compacted SRML files. This would allow real-time queries against the compacted files, making the method even more useful. Our method would be effective in this query-based solution as well, since only the affected nodes have to be decompacted in contrast to a compressed file, where the whole file has to be decompressed before any queries can be executed. We also plan to research the possibilities of combining our method with MHM (Multiplexed Hierarchical Modelling) to create a more efficient complement to compressors.

Appendix A. DTD of SRML elements

```
<!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"
type (temp | permanent) "permanent" 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)>
```

Appendix B. Description of SRML elements

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 provided which *attribute* of which *element* the semantic rule is defining 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 facilitates 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 direction of which the index is taken from can be given. The possible directions are *begin* and *end*. This makes it possible to reference any element in the list (e.g.: second element from the end with the name of the value given in the *element* attribute). It is possible to use the *srml:all* constant as well in the *element* attribute, which results in an index describing which child the element is at the DOM level. If an *element* name is provided, then the index returned 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, type* and *num* attributes. In 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*. The *type* attribute specifies whether the current attribute is permanent or temporary. The default is permanent. If an attribute is temporary it means that it is used for temporary calculation and should not be stored in the output XML file.
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, and 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 sufficient 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.

References

[1] G. Psaila, S. Crespi-Reghizzi, Adding semantics to XML, in: Proceedings of the Second Workshop on Attribute Grammars and their Applications, WAGA'99 Amsterdam, The Netherlands, 1999, pp. 113–132.

[2] D.E. Knuth, Semantics of context-free languages, Math. Syst. Theor. 2 (1968) 127–145.

[3] T. Bray, J. Paoli, C. Sperberg-McQueen, Extensible Markup Language, XML 1.0 W3C Recommendation, 1998 http://www.w3.org/TR/REC-xml.

[4] J.R. Quinlan, Induction of decision trees, in: Readings in Machine Learning, San Mateo, CA, 1990, pp. 57–69.

[5] DOM, http://www.w3.org/DOM/.

[6] H. Liefke, D. Suciu, XMill: an efficient compressor for XML data, in: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, 2000, pp. 153–164. http://www.research.att.com/sw/tools/xmlill/.

[7] M. Hirakawa, T. Tanaka, JSNP: a database of common gene variations in the Japanese population, Nucleic Acids Research, vol. 30, Oxford University Press, New York, 2002. pp. 158–162.

[8] UserLand Frontier XML-Database, 2003 http://frontier.userland.com/stories/storyReader$1085.

[9] B. Schafner, Model XML DTDs with Rational Rose, 2003, http://builder.com.com/5100-31-5075476.html.

[10] M. Cokus, D. Winkowski, XML sizing and compression study for military wireless data, XML Conference and Exposition, Baltimore Convention Center, Baltimore, MD, 2002.

[11] T. Gyimóthy, T. Horváth, Learning semantic functions of attribute grammars, Nordic J. Comput. 4 (3) (1997) 287–302.

[12] R. Ferenc, Á. Beszédes et al., Columbus—reverse engineering tool and schema for C++, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002), IEEE Computer Society, Montréal, Canada, 2002, pp. 172–181.

[13] M. Kálmán, F. Havasi et al., Compacting XML documents, in: Proceedings of the Eighth Symposium on Programming Languages and Software Tools (SPLST'03), University of Kuopio, Department of Computer Science, Kuopio, Finland, June 17–18 2003, pp. 137–151.

[14] H. Alblas, Introduction to Attribute Gammars, in: Proceedings of SAGA LNCS, vol. 545, Springer, Berlin, 1991, pp. 1–16.

[15] R. Ferenc, S. Elliot et al., Towards a standard schema for C/C++, in: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE 2001), IEEE Computer Society, Stuttgart, Germany, 2001, pp. 49–58.

[16] CodeSurfer, http://www.grammatech.com/products/codesurfer/.

[17] T. Mitchell, Machine learning, McGraw-Hill, New York, 1997.

[18] F. Havasi, XML semantics extension, Acta Cybernetica 15 (2) (2002) 509–528.

[19] M. Cannataro, G. Carelli, et al., Semantic Lossy compression of XML data, In Knowledge Representation Meets Databases, 2001.

[20] M. Levene, P. Wood, XML Structure Compression, 2002, http://citeseer.nj.nec.com/501982.html.

[21] C.F. Goldfarb, P. Prescod, The XML handbook, Prentice-Hall, Englewood Cliffs, NJ, 2001.

[22] M. Lanza, CodeCrawler—lessons learned in building a software visualization tool, in: Proceedings of Conference on System Maintenance and Reengineering, IEEE Computer Society Press, Stuttgart, Germany, 2003, pp. 409–418.

[23] J. Cheney, Compressing XML with multiplexed hierarchical PPM models, in: Proceedings of the Data Compression Conference, IEEE Computer Society Press, Stuttgart, Germany, 2001.

[24] P. Tolani, J. Haritsa, XGRIND: a query-friendly XML compressor, in: Proceedings of the 18th International Conference on Data Engineering, IEEE Computer Society Press, Stuttgart, Germany, 2002.

Miklós Kálmán was born in Szeged, Hungary in 1979. He received the diploma from the University of Szeged as a Software Engineer mathematician in 2003. He is currently a PhD student at the Department of Software Engineering where his research area is XML compaction. He has experience in several low-level hardware programming, artificial intelligence and image processing.

Ferenc Havasi has received the his diploma from the University of Szeged as a Software Engineer mathematician in 2001. He is currently a PhD student at the Department of Software Engineering where his research areas are XML compaction and code compression. He already has a publication in this area (XML Semantics Extension).

Dr Tibor Gyimóthy is the head of the Software Engineering Department at the University of Szeged, Hungary. His research interests are in program comprehension, slicing, reverse engineering and compiler optimalization. He has published more than 100 papers in international journals and conference proceedings. Dr Gyimothy was the leader of several software engineering R&D projects. He is the Program Co-Chair of the 21th International Conference on Software Maintenance, will be held in Budapest, Hungary in 2005.