Acta Cybernetica **17** (2005) 289–310.

An approach for compacting XMI documents

Miklós Kálmán*

Abstract

One of the most common formats for storing information is XML. It is used in many areas, with its spectrum expanding day by day. A big drawback of the XML format is that the documents can be quite large. This causes problems wherever size is an important issue, for example in embedded systems or whenever the document has to be transferred over a network.

Another widely used format is XMI (XML Metadata Interchange), which is derived from the XML format. Since XMI is an extension of XML the same problems are inherited. A Metalanguage called SRML was defined which provided a good solution to describe the relationships between XML attributes making the compacting of XML documents possible. The main idea behind our paper is to extend the SRML definition in such a way that it supports the XMI environment. This results in a method for compacting XMI documents using semantic rules.

Introduction

It has become an accepted fact that in the electronic age information exchange and storage is quite important. One of the most common formats for this is XML[3]. This format is rather versatile making it a good choice for common information exchange. More and more applications are able to store information in this format. The application areas span military use[7], medical science(human genome mapping [5], component modelling [6]. If the growth continues at this rate, XML documents will span every area in computing.

XML documents can be quite large, but in many cases systems can only handle smaller files, like in the case of embedded systems. Size is an important issue also when the files have to be transferred over the internet. One solution to overcome this issue is to use general compressors (e.g: gzip) or XML compressors like XMill[4]. Unfortunately the compressed file may still be too large.

In the XML environment the attributes often have some sort of relationship, therefore they may be calculated from each other. To store these calculation rules a metalanguage called SRML[11] (Semantic Rule Metalanguage) was defined. It enabled the compaction of XML documents.

*Department of Software Engineering, Aradi vértanuk tere 1., H-6720 Szeged, Hungary, +36 62 544143, email: kalman@inf.u-szeged.hu

One of the most widespread applications for XML is the XMI format. The term XMI[13] stands for XML Metadata Interchange. It is an XML application that is used to manage the standardized interchange of object models and metadata among groups working in team development environments using tools and applications from various vendors. XMI can also be used to exchange information about data warehouses. XMI is based on three industry standards: XML[3], UML[14] (Unified Modelling Language), and MOF[16] (an OMG modelling and metadata repository standard). The architecture enables tools to share metadata programmatically using XML or CORBA[15] interfaces specified in the UML or MOF standards. Since XMI is an extension of XML it inherits its problems also. Size becomes and issue in the XMI environment as well.

For this reason we wish to extend the SRML definition to support the storage of calculation rules in the XMI environment. The SRML language is based on describing relationships between attributes. The current SRML format for example cannot describe calculation rules which are based on text elements. In XMI documents it is quite common to have text elements, which can and may be calculated. The article describes how the SRML language can be extended to provide a method for XMI compaction. This extension makes the SRML metalanguage more generic and may be applied to other XML based formats as well. We have implemented an XML compactor in [10], which provides an effective way of compacting XML documents. With the help of this extension to the SRML metalanguage it can be modified to handle XMI compaction as well.

In the following sections first some background knowledge will be provided. Then we will show how the SRML metalanguage is extended using examples to illustrate how it can be used. Afterwards some areas will be described, where this method can be of great use. Finally, we round off the paper by mentioning related works, a brief summary and topics for future study.

1 Preliminaries

In this section a basic introduction to XML files will be given as well as the XMI format. The necessary preliminaries for Attribute Grammars will be presented. This will be needed to better understand parts in the subsequent sections.

1.1 XML

The first concept that must be introduced is the XML format. The complete XML description can be found in [3] and [17]. XML documents are quite 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. *Figure 1* describes a simple example, which stores automobiles in an XML format. Each automobile has the following attributes: Make, Model, Year, Color, NetPrice, Tax, SalesPrice.

```
<XML>
 <Auto Make="Ford" Model="Mondeo" Year="2000" Color="Black"
       NetPrice="25000" Tax="15" SalesPrice="28750"/>
 <Auto Make="Opel" Model="Astra" Year="2004" Color="Papyrus"
       NetPrice="20000" Tax="15" SalesPrice="23000"/>
 <Auto Make="Volvo" Model="S40" Year="2000" Color="Red"
       NetPrice="30000" Tax="15" SalesPrice="34500"/>
 <Auto Make="Fiat" Model="Stilo" Year="2000" Color="Red"
       NetPrice="18000" Tax="15" SalesPrice="20700"/>
 <Auto Make="Toyota" Model="Corolla" Year="2000" Color="Red"
       NetPrice="24000" Tax="15" SalesPrice="27600"/>
</XML>
```

Figure 1: Example for automobile storage in XML

1.2 DTD

The term DTD[3] is short for Document Type Definition. The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. The DTD can be viewed as he grammar of a class of documents. A DTD can be declared inline in the XML document, or as an external reference. A DTD contains markup declarations. Each declaration can be either an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration. Below some explanation will be given for the first two declaration types, since this article uses only these.

- **Element type declaration:** The element structure of an XML document may be constrained using element type and attribute-list declarations for validation purposes. An element type declaration constrains the element's content. For example if an XMI/XML file can only contain <expr> elements it can be listed here.

- **Attribute-list declaration:** Attribute-list declarations specify the name, data type, and default value (if any) of each attribute associated with a given element type. The definition of an attribute-list can be done using the <!ATTLIST ..> keyword. For example if the <expr> element can have a "type" attribute it would be declared the following way:
  <!ATTLIST expr type CDATA #REQUIRED> The word #REQUIRED denotes that the attribute must exists on all "expr" elements. If the word #IMPLIED is used it is not obligatory to have the given attribute present in all cases.

If an XML document uses a DTD file then all of its elements must conform to the DTD specification. Using a DTD it is possible to define a standard form for interchanging data. An application can use a standard DTD to verify that data that it receives from the outside world is valid, but it can also verify that the data it is storing conforms to this specification.

1.3 XMI

The XMI format (XML Metadata Interchange) is an extension of MOF[16] into an XML environment. It is an OMG[12] standard that defines the rules for generating an XML DTD from a metamodel. The MOF (Meta Object Facility) was defined before XML became so widespread. However, the technology-neutral nature of the MOF Core made it relatively easy to produce a mapping from the MOF Core's elements to XML so that, given a metamodel, a Document Type Definition (DTD) could be generated. This DTD[3] can be used to stream models that conform to the metamodel.

The MOF Core, as a subset of UML[14], is object oriented; XML is not. Good object models make liberal use of object orientation and subtyping, therefore they should not be restricted in any way. The architects of XMI wished to avoid having DTD elements repeat all the properties of all their ancestors since that would make it quite cumbersome to render typical object models to DTDs

The UML DTD is the most widely used XMI DTD. UML is not only a notation, but the official specification has a complete MOF-compliant metamodel. It is MOF-compliant because its elements are defined via the constructs of the MOF Core. This metamodel was fed into an XMI DTD generator to produce the UML DTD used by tools to export and import UML models.

XMI provides an open interchange between applications written by different vendors. These applications include:

- Design Tools: these tools include object-oriented UML tools like Rational Rose
- Development tools including integrated development environments like IBM VisualAge for Java and Microsoft Visual Studio .NET
- Databases, Data Warehouses and Business Intelligence tools like IBM DB/2, Visual Warehouse, Intelligent Miner for Data and Oracle 8i
- Software assets like program source code (C/C++, Java) and CASE tools such as TakeFive SniFF+
- Repositories, including IBM VisualAge TeamConnection and Unisys Universal Repository
- Reports, report generation tools, documentations tools, and web browsers

In *Figure 2* the XMI Open Interchange can be seen for the applications mentioned above. If a vendor wishes to participate in the architecture they only need to add XMI support to their product to leverage access to all the other tools.

The XMI version of the example described in *Figure 1* can be created. In *Figure 3* the DTD of the XMI example is presented and *Appendix A* describes the automobile example in an XMI form.

The MOF-compliant metamodel used in this example can be seen in *Figure 4*. It is a simple class with the attributes "Make", "Model", "Color", "NetPrice","Tax","SalesPrice".

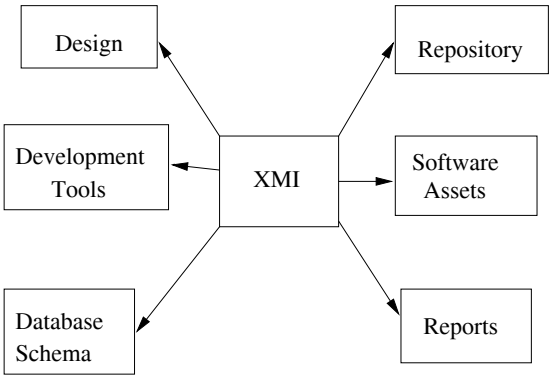

Figure 2: Open interchange with XMI

```
<!ELEMENT Auto (Auto.Make, Auto.Model, Auto.Year, Auto.Color,
                Auto.NetPrice, Auto.Tax, Auto.SalesPrice, XMI.extension*)?>
<!ATTLIST Auto %XMI.element.att; %XMI.link.att;>
<!ELEMENT Auto.Make (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.Model (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.Year (#PCDATA |XMI.reference)*>
<!ELEMENT Auto.Color (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.NetPrice (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.Tax (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.SalesPrice (#PCDATA | XMI.reference)*>
```

Figure 3: The DTD for the XMI automobile example.

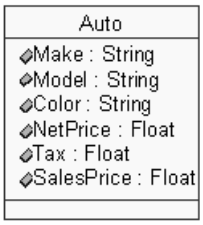

Figure 4: The MOF-compliant metamodel used in the XMI automobile example

1.4 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 nonterminal

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 nonterminal 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.

```
N = { expr, multexpr, addexpr, num }
S = expr     T ={"ADD","MUL",NUM}
P :
 (1) expr     -> num                (5) addexpr  -> expr "SUB" expr
 (2) expr     -> multexpr           (6) multexpr -> expr "MUL" expr
 (3) expr     -> addexpr            (7) multexpr -> expr "DIV" expr
 (4) addexpr  -> expr "ADD" 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] [9]. In this section only those definitions will be mentioned which may be needed to understand the 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 given 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 *Figure 5* shows an AG for computing the *type* of the simple numeric expression described above.

In the example the "type" of *addexpr* is *real* if the first or the second *expr* has a *real* type; otherwise it is *int*.

An analogy between AG and XML documents can be discovered. 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. An attribute specification in the AG corresponds to an attribute list declaration in the DTD. A key concept in Attribute Grammars which has no XML counterpart are the semantic rules. It might be useful to apply these semantic rules in the XML environment as well. Keeping this concept in view the SRML[11] metalanguage was introduced. It provides XML documents with the support for semantic rules. The metalanguage is based on the analogy described above.

```
(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";
```

Figure 5: An example for the semantic rules of a numeric expression.

2 An approach for XMI compaction

In the previous section it was shown that there is a relationship between XML and AG. This analogy also applies to XMI, as it is an extension of XML. The SRML[11] metalanguage provided a method for compacting XML documents. This metalanguage had its weaknesses. One of these was that it was impossible to reference text nodes, whereas most XML documents use this type of representation alot. If we could extend this metalanguage to be more generic it would be possible to store rules which enable XMI compaction. This section will detail how SRML needs to be extended. An example of XMI compaction will also be provided. This extension would enable our XML compactor [10] to handle XMI files as an input, making the XMI compaction possible.

2.1 Extending the SRML metalanguage

An XML-based metalanguage called SRML[11] (Semantic Rule Meta Language) has been defined to describe semantic rules. With the help of this metalanguage XML files could be compacted using semantic rules. The current SRML definition only allows the description of attribute based rules. It has no way of describing rules which contain references to text elements. This was a restriction in the current SRML definition. In an XMI environment using these text elements as a basis for rule creation is a crucial aspect. In order to provide a way for describing such rules the SRML metalanguage has to be extended. This section will detail how this extension can is accomplished.

The following examples will demonstrate why an extension is required. The example in *Figure 6* contains three attributes *x,y,z*. The *z* attribute can be calculated by adding *x* and *y* together. The SRML for this XML can be seen in *Figure 7*. The keyword *srml:root* refers to the root of the rule, in this case the *Number* element.

```
<XML>
    <Number x="13" y="10" z="23"/>
</XML>
```

Figure 6: A simple XML containing three attributes

```
<SRML>
 <rules-for root="Number">
  <rule element="srml:root" attrib="z">
   <expr>
    <binary-op op="add">
     <expr><attribute element="srml:root" attrib="x"/></expr>
     <expr><attribute element="srml:root" attrib="y"/></expr>
    </binary-op>
   </expr>
  </rule>
 </rules-for>
</SRML>
```

Figure 7: The SRML rules for *Figure 6*

If we introduce a new attribute called "reftype" into the elements rule and attribute it makes it possible to describe references to text nodes. A text node refers to a value being stored between two element tags (e.g. <value>10</value>) instead of storing them as attributes (e.g: <node1 value="10"/>). The value inside the text node is not important, it can be text or numeric. This reference is a vital part of XMI compaction, since most references are text-node based in an XMI environment. In *Figure 8* the XMI form of the XML defined in *Figure 6*.

```
<XMI>
    <Number>
     <Number.x>13</Number.x>
     <Number.y>10</Number.y>
     <Number.z>23</Number.z>
    </Number>
</XMI>
```

Figure 8: A simple XMI containing three attributes

After extending the SRML metalanguage *Figure 9* shows the SRML rules that can now be defined for *Figure 8*.

```
<SRML>
 <rules-for root="Number">
  <rule element="srml:root" attrib="z" reftype="text">
   <expr>
    <binary-op op="add">
     <expr><attribute element="srml:root" attrib="x" reftype="text"/></expr>
     <expr><attribute element="srml:root" attrib="y" reftype="text"/></expr>
    </binary-op>
   </expr>
  </rule>
 </rules-for>
</SRML>
```

Figure 9: The extended SRML rules for *Figure 8*

It is now also possible to combine references between attributes and text nodes. Considering the example in *Figure 10* new types of rules can be defined. Earlier

it was not possible to define such rules. The example contains a *total* attribute, which can be calculated from the three text nodes that the element has.

```
<XMI>
    <Number total="46">
     <Number.x>13</Number.x>
     <Number.y>10</Number.y>
     <Number.z>23</Number.z>
    </Number>
</XMI>
```

Figure 10: A simple XMI with combined attribute types

The modified SRML rule set can be seen below. It contains both text and attribute references.

```
<SRML>
 <rules-for root="Number">
  <rule element="Number" attrib="z" reftype="text">
   <expr>
    <binary-op op="add">
     <expr><attribute element="Number" attrib="x" reftype="text"/></expr>
     <expr><attribute element="Number" attrib="y" reftype="text"/></expr>
    </binary-op>
   </expr>
  </rule>
  <rule element="srml:root" attrib="total" reftype="attrib">
   <expr>
    <binary-op op="add>
     <expr>
      <binary-op op="add">
       <expr><attribute element="Number" attrib="x" reftype="text"/></expr>
       <expr><attirubte element="Number" attrib="y" reftype="text"/></expr>
      </binary-op>
     </expr>
     <expr><attribute element="Number" attrib="z" reftype="text"/></expr>
     </binary-op>
    </expr>
  </rule>
 </rules-for>
</SRML>
```

The complete modified SRML description can be seen in *Appendix G*. Using the "reftype" extension the compactor would know where to look for the given value. It would either look in an attribute or in a node containing the value as text. Another approach would be to convert all value occurrences to a standard form. This form can be the attribute form, since the SRMLTool[10] can handle these easily, however this parsing would take up more time, compared to just telling the compactor to look for the value elsewhere.

2.2 Compacting XMI documents

After the SRML extension has been described in the previous section now it is possible to define rules for XMI text-nodes, thus making XMI compaction possible.

This section will provide an example for this. First an SRML description will be provided for the example shown in *Appendix A*. The set of SRML rules can be found in *Appendix B*. Before showing the result of the compaction first some explanation will be provided for the rules mentioned in the Appendix. It can be noticed that the "SalesPrice" can be calculated from the "NetPrice" by adding the "Tax" as a percent value. The example contains a constant "Tax" value of 15%. Another rule that can be written is that the "Color" of the car is usually "Red" if the "Year" is 2000. If the rule does not match the text element it will not be removed since then it cannot be restored later. When the SRML rules described in *Appendix B* are applied to the example detailed in *Appendix A* the XMI document can be compacted (see *Appendix C*). The original input XMI was 1788 bytes and the compacted XMI became 1256 bytes. This resulting XMI file could be compacted to 70.2% of the original file.

Using a tool called Columbus[8] makes it possible to create XML/XMI files from C++ programs. It is a reverse engineering tool that analyzes the structure of the code and creates an XML/XMI output, which makes it possible to view the relationship between functions, parameter references...etc. If we use the code sniplet described in *Figure 11* Columbus can generate an XMI output from it. This XMI output is partially shown in *Appendix D*.

Examining the output XMI it can be seen that there are some rules that can be described. One of these is that the *Core.DataType* is usually referencing id_dt_1. Another rule that can be written is that if the *ModelElement.name* ends with *Return* then the *Parameter.kind* is *return*, otherwise it is *inout*. It is also visible that the *ModelElement.visibility* is always *public*. Using these rules it is possible to create the SRML file, which can be seen in *Appendix E*. The original input file size is 14439 and the resulting compacted XMI file becomes 11263, achieving a 12% compaction. The output of the compaction can be seen in *Appendix F*.

Compacting XMI files can be quite effective, since they usually contain many values which can be described using semantic rules through SRML. Some of these have been mentioned earlier and can be seen in *Appendix E*. There are however other rule types which can be described using the SRML metalanguage. One of these would be the path name for the object. These path names can be rather long and are usually repetitive, since they are mostly part of the same object base. If for instance the XMI file is referencing objects in "*/home/Development/ProjectX/...*" then it is possible to create concatenation rules for it. The rule would contain the base path name and concatenate the actual attribute value to the end, making the compaction effective.

One might ask why isn't it possible to describe element names using SRML. For example in the CPPML example described before the element prefix "Foundation.Core." is always present. It might be possible to create an extension which could refer to these types of rules, however this would make the compacted XMI file no longer comply to the DTD. So the alternate solution would be to create a DTD modification as well. This however is only theoretical, it requires future research.

```
class Math
{
      public:
             float result;

      public:
             float GetResult()
             {
                return result;
             }

             float Add(float a, float b)
             {
                result = a+b;
                return result;
             }

             float Mul(float a, float b)
             {
                result = a*b;
                return result;
             }

             float Div(float a, float b)
             {
                if (b==0)
                   result = 0;
                else
                    result = a / b;
                return result;
             }

             float Avg(float a, float b)
             {
                result = Div( Add(a,b), 2);
                return result;
             }
...
```

Figure 11: A simple C++ program sniplet

3 Applicability of the method

The method introduced in this article can be applied in many fields of computing. One of these areas is the field of Relational Databases. The reason why XML/XMI compressors are not effective in this field is that the file has to be decompressed completely in order to access parts of the document. Compaction on the other hand is a much more feasible approach, since the SRML file contains the calculation rules and the compacted XMI/XML file is not in a binary format. If a query is placed against an attribute/text-node of the document only the parts that are affected to respond to the query need to be decompacted. This provides a very optimal solution, since the document can be stored in a compacted form, making the resource requirements much smaller. If a node/attribute has to be added to the

file it is added, then when the file is to be closed it would be compacted once again. We are planning to implement a library which would serve as a layer between the user and the compacted XML/XMI document. In *Figure 12* the architecture of the proposed method is shown.

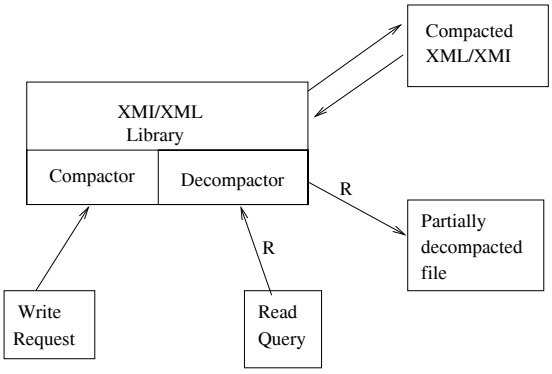

Figure 12: The demand-driven XML/XMI compactor library architecture

4 Related Work

After describing our method it must be mentioned what other research has been done in this area. These articles contain parts which are similar to our approach, but not identical.

The first notion of adding semantics to the XML environment was introduced in [1]. It starts off with a brief introduction to XML. The paper provides a method for transforming the element description of DTD into EBNF formal rule description. Afterwards it introduces its own SRD (Semantics Rule Definition). The reason why we didn't try to extend SRD instead of SRML is that 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. This would make referencing text elements or regular expressions quite hard to accomplish.

The theory of compacting XML documents using SRML was published in [11]. This definition is quite durable and easily extendible. This is the reason why we chose to extend it into the XMI environment. Once the extension is completed it can be used to compact both XML and XMI documents, which makes it a very valuable asset.

We have implemented an XML Compactor[10] which uses SRML rules to compact XML documents. Our implementation of the XML compactor could achieve a 30% compaction on larger XML files. Using the extension mentioned in this article now it is possible to compact XMI documents with minor modifications to the code.

XMill[4] is an XML compressor. It creates a binary output, therefore we cannot extend it. However if we first compact and XMI file and then compress it with XMill, then the size can be reduced considerably. This means that our method can increase the efficiency of third party compressors.

5 Summary and future work

XML documents have become very widespread. They span many areas of computing. The problem is that they can be quite large at times. The SRML metalanguage is able to store rules which describe how attributes can be calculated from each other. XMI is an extension of the XML architecture. The XMI documents inherit the problems that arise with XML. The SRML description did not have a way to describe rules for the XMI document architecture. We have extended the SRML metalanguage to provide an effective method for describing calculation rules in an XMI environment making the compaction of XMI documents possible. Using this extension the tool we have implemented for XML compaction[10] can be used with minor modifications to enable XMI compaction.

In the future we plan to modify our existing XML compressor using this extension to create a universal XML/XMI compacting tool. The idea to create a demand-driven XMI compactor library is also planned, which would make the method a very effective tool in every day use, since the size decrease would mean that larger amount of information could be stored without sacrificing disk space. It could be used for example as an aid for third party database engines utilizing the XMI/XML format.

References

[1] Psaila, G. and Crespi-Reghizzi, S, 1999. *Adding Semantics to XML*, In Proceedings of the Second Workshop on Attribute Grammars and their Applications, WAGA'99 Amsterdam, The Netherlands, pages 113-132

[2] Knuth, D.E., 1968. *Semantics of Context-Free Languages*, Mathematical Systems Theory Vol 2., pages 127-145

[3] Bray, T. and Paoli, J. and Sperberg-McQueen, C., 1998. *Extensible markup language*, XML 1.0 W3C recommendation, http://www.w3.org/TR/REC-xml

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

[5] Hirakawa M. and Tanaka T., 2002. *JSNP: a database of common gene variations in the Japanese population*, Nucleic Acids Research Vol. 30, pages 158-162, Oxford University Press

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

[7] Cokus, M. and Winkowski D., 2002. *XML Sizing and Compression Study For Military Wireless Data*, XML Conference and Exposition, Baltimore Convention Center, Baltimore, MD

[8] Ferenc, R. and Beszédes, Á. et al, 2002., *Columbus – Reverse Engineering Tool and Schema for C++*, In Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002), IEEE Computer Society, pages 172-181, Montréal, Canada

[9] Alblas, H., 1991. *Introduction to Attribute Gammars*, In Proceedings of SAGA LNCS Vol 545.,Springer-Verlag, pages 1-16

[10] Kálmán, M., Havasi, F., Gyimóthy, T., 2005. *Compacting XML documents*, Accepted for publication in The Journal of Information and Software Technology, Elsevier.

[11] Havasi, F., 2002. *XML Semantics Extension*, Acta Cybernetica Vol 15 No. 2, pages 509-528

[12] *OMG Unified Modeling Language Specification, Version 2.0*, http://www.omg.org

[13] *XML Metadata Interchange (XMI) Version 1.1*, http://cgi.omg.org/docs/ad/99-10- 02.pdf

[14] *Unified Modeling Language (UML)*, http://www.uml.org/

[15] *Common Object Request Broker Architecture (CORBA)*, http://www.omg.org/gettingstarted/corbafaq.htm

[16] *Meta-Object Facility (MOF), version 1.4*, http://www.omg.org/technology/documents/formal/mof.htm

[17] Goldfarb, C.F. and Prescod, P. 2001. *The XML Handbook*, Prentice-Hall

Appendix

A Example for automobile storage in XMI

```
<!DOCTYPE XMI SYSTEM "auto.dtd">
<XMI xmi.version="1.0">
 <XMI.header>
   <XMI.documentation>Some automobiles.</XMI.documentation>
 </XMI.header>
   <XMI.content>
      <Auto>
         <Auto.Make>Ford</Auto.Make>
         <Auto.Model>Mondeo</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Black</Auto.Color>
         <Auto.NetPrice>25000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>28750</Auto.SalesPrice>
     </Auto>
     <Auto>
         <Auto.Make>Opel</Auto.Make>
         <Auto.Model>Astra</Auto.Model>
         <Auto.Year>2004</Auto.Year>
         <Auto.Color>Papyrus</Auto.Color>
         <Auto.NetPrice>20000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>23000</Auto.SalesPrice>
     </Auto>
     <Auto>
         <Auto.Make>Volvo</Auto.Make>
         <Auto.Model>S40</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Red</Auto.Color>
         <Auto.NetPrice>30000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>34500</Auto.SalesPrice>
     </Auto>
     <Auto>
         <Auto.Make>Fiat</Auto.Make>
         <Auto.Model>Stilo</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Red</Auto.Color>
         <Auto.NetPrice>18000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>20700</Auto.SalesPrice>
     </Auto>
     <Auto>
         <Auto.Make>Toyota</Auto.Make>
         <Auto.Model>Corolla</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Red</Auto.Color>
         <Auto.NetPrice>24000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>27600</Auto.SalesPrice>
     </Auto>
```

```
   </XMI.content>
 </XMI.header>
</XMI>
```

B The SRML for the automobile example

```
<SRML>
 <rules-for root="Auto">
  <rule element="Auto" attrib="SalesPrice" reftype="text">
   <expr>
   <binary-op op="mul">
    <expr>
     <binary-op op="add">
       <expr>
         <binary-op op="div">
          <expr><attribute element="Auto" attrib="Tax" reftype="text"/></expr>
          <expr><data>100</data></expr>
         </binary-op>
        </expr>
       <expr><data>1</data></expr>
     </binary-op>
    </expr>
    <expr><attribute element="Auto" attrib="NetPrice" reftype="text"/></expr>
   </expr>
   </binary-op>
   </expr>
  </rule>
  <rule element="Auto" attrib="Tax" reftype="text">
   <expr>
    <data>15</data>
   </expr>
  </rule>
  <rule element="Auto" attrib="Color" reftype="text">
   <expr>
    <if-expr>
     <expr>
      <binary-op op="equals">
       <expr><attribute element="Auto" attrib="Year" reftype="text"/></expr>
       <expr><data>2000</data></expr>
      </binary-op>
     </expr>
     <expr><data>Red</data></expr>
     <expr><no-data/></expr>
    </if-expr>
   </expr>
  </rule>
 </rules-for>
</SRML>
```

C The compacted XMI example

```
<!DOCTYPE XMI SYSTEM "auto.dtd">
<XMI xmi.version="1.0">
 <XMI.header>
   <XMI.documentation>Some automobiles.</XMI.documentation>
 </XMI.header>
   <XMI.content>
      <Auto>
         <Auto.Make>Ford</Auto.Make>
         <Auto.Model>Mondeo</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Black</Auto.Color>
         <Auto.NetPrice>25000</Auto.NetPrice>
     </Auto>
     <Auto>
         <Auto.Make>Opel</Auto.Make>
         <Auto.Model>Astra</Auto.Model>
         <Auto.Year>2004</Auto.Year>
         <Auto.Color>Papyrus</Auto.Color>
         <Auto.NetPrice>20000</Auto.NetPrice>
     </Auto>
     <Auto>
         <Auto.Make>Volvo</Auto.Make>
         <Auto.Model>S40</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.NetPrice>30000</Auto.NetPrice>
     </Auto>
     <Auto>
         <Auto.Make>Fiat</Auto.Make>
         <Auto.Model>Stilo</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.NetPrice>18000</Auto.NetPrice>
     </Auto>
     <Auto>
         <Auto.Make>Toyota</Auto.Make>
         <Auto.Model>Corolla</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.NetPrice>24000</Auto.NetPrice>
     </Auto>
   </XMI.content>
 </XMI.header>
</XMI>
```

D An XMI output for the CPP example

```
...
<!-- ====================     Math  [class]       ==================== -->
  <Foundation.Core.Class xmi.id = 'id101'>
    <Foundation.Core.ModelElement.name>Math</Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
    <Foundation.Core.ModelElement.isSpecification xmi.value = 'true'/>
    <Foundation.Core.GeneralizableElement.isRoot xmi.value = 'true' />
    <Foundation.Core.GeneralizableElement.isLeaf xmi.value = 'true' />
    <Foundation.Core.GeneralizableElement.isAbstract xmi.value = 'false' />
```

```
<Foundation.Core.Class.isActive xmi.value = 'false' />
<Foundation.Core.ModelElement.namespace>
  <Model_Management.Package xmi.idref = 'id100' />  <!--  global namespace  -->
</Foundation.Core.ModelElement.namespace>
  <Foundation.Core.Classifier.feature>
  <!-- ============     Math  [class]  result  [Attribute]     =========== -->
  <Foundation.Core.Attribute xmi.id = 'id102' >
    <Foundation.Core.ModelElement.name>result
    </Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
    <Foundation.Core.Feature.ownerScope xmi.value = 'classifier'/>
    <Foundation.Core.StructuralFeature.type>
      <Foundation.Core.DataType xmi.idref = 'id_dt_1'/>    <!-- float -->
    </Foundation.Core.StructuralFeature.type>
  </Foundation.Core.Attribute>
  </Foundation.Core.Classifier.feature>
  <Foundation.Core.Classifier.feature>
  <!-- ==========     Math  [class]  GetResult  [Operation]     =========== -->
  <Foundation.Core.Operation xmi.id = 'id105' >
    <Foundation.Core.ModelElement.name>GetResult
    </Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
    <Foundation.Core.BehavioralFeature.parameter>
      <Foundation.Core.Parameter xmi.id = 'id105.Return' >
        <Foundation.Core.ModelElement.name>GetResult.Return
        </Foundation.Core.ModelElement.name>
        <Foundation.Core.Parameter.kind xmi.value = 'return'/>
        <Foundation.Core.Parameter.type>
          <Foundation.Core.DataType xmi.idref = 'id_dt_1' />    <!-- float -->
        </Foundation.Core.Parameter.type>
      </Foundation.Core.Parameter>
    </Foundation.Core.BehavioralFeature.parameter>
  </Foundation.Core.Operation>
  </Foundation.Core.Classifier.feature>
  <Foundation.Core.Classifier.feature>
  <!-- =============     Math  [class]  Add  [Operation]     ============= -->
  <Foundation.Core.Operation xmi.id = 'id111' >
    <Foundation.Core.ModelElement.name>Add</Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
    <Foundation.Core.BehavioralFeature.parameter>
      <Foundation.Core.Parameter xmi.id = 'id111.Return' >
        <Foundation.Core.ModelElement.name>Add.Return
        </Foundation.Core.ModelElement.name>
        <Foundation.Core.Parameter.kind xmi.value = 'return'/>
        <Foundation.Core.Parameter.type>
          <Foundation.Core.DataType xmi.idref = 'id_dt_1' />    <!-- float -->
        </Foundation.Core.Parameter.type>
      </Foundation.Core.Parameter>
        <Foundation.Core.Parameter xmi.id = 'id112' >
          <Foundation.Core.ModelElement.name>a
          </Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.visibility xmi.value = 'public'/>
          <Foundation.Core.Parameter.kind xmi.value = 'inout'/>
          <Foundation.Core.Parameter.type>
            <Foundation.Core.DataType xmi.idref = 'id_dt_1' />    <!-- float -->
          </Foundation.Core.Parameter.type>
```

```
          </Foundation.Core.Parameter>
          <Foundation.Core.Parameter xmi.id = 'id113' >
            <Foundation.Core.ModelElement.name>b
            </Foundation.Core.ModelElement.name>
            <Foundation.Core.ModelElement.visibility xmi.value = 'public'/>
            <Foundation.Core.Parameter.kind xmi.value = 'inout'/>
            <Foundation.Core.Parameter.type>
              <Foundation.Core.DataType xmi.idref = 'id_dt_1' />    <!-- float -->
            </Foundation.Core.Parameter.type>
          </Foundation.Core.Parameter>
        </Foundation.Core.BehavioralFeature.parameter>
      </Foundation.Core.Operation>
      </Foundation.Core.Classifier.feature>
      <Foundation.Core.Classifier.feature>
      ...
```

E The SRML of the XMI form of the CPP program

```
<SRML>
 <rules-for root="Foundation.Core.ModelElement.visibility>
  <rule element="srml:root" attrib="xmi.value">
   <expr>
    <data>public</data>
   </expr>
  </rule>
 </rules-for>
 <rules-for root="Foundation.Core.DataType">
  <rule element="srml:root" attrib="xmi.idref">
   <expr>
    <data>id_dt_1</data>
   </expr>
  </rule>
 </rules-for>
 <rules-for root="Foundation.Core.Parameter">
  <rule element="Foundation.Core.Parameter.kind" attrib="xmi.value">
   <epxr>
    <if-expr>
     <epxr>
      <binary-op op="ends-with">
       <expr>
        <attribute element="Foundation.Core.ModelElement.name" reftype="text">
       </expr>
       <expr><data>.Return</data></expr>
      </binary-op>
     </expr>
     <expr><data>return</data></expr>
     <expr><data>inout</data></expr>
    </if-expr>
   </expr>
  </rule>
 </rules-for>
</SRML>
```

F The compacted XMI of the CPP program

```
<!-- ====================     Math  [class]       ==================== -->
<Foundation.Core.Class xmi.id = 'id101' xmi.value='c:\\temp\\Math.cpp'>
  <Foundation.Core.ModelElement.name>Math</Foundation.Core.ModelElement.name>
  <Foundation.Core.ModelElement.visibility/>
  <Foundation.Core.ModelElement.isSpecification xmi.value = 'true'/>
  <Foundation.Core.GeneralizableElement.isRoot xmi.value = 'true' />
  <Foundation.Core.GeneralizableElement.isLeaf xmi.value = 'true' />
  <Foundation.Core.GeneralizableElement.isAbstract xmi.value = 'false' />
  <Foundation.Core.Class.isActive xmi.value = 'false' />
  <Foundation.Core.ModelElement.namespace>
    <Model_Management.Package xmi.idref = 'id100' />  <!--  global namespace  -->
  </Foundation.Core.ModelElement.namespace>
    <Foundation.Core.Classifier.feature>
    <!-- ==========     Math  [class]  result  [Attribute]     ============= -->
    <Foundation.Core.Attribute xmi.id = 'id102' >
      <Foundation.Core.ModelElement.name>result
      </Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.visibility/>
      <Foundation.Core.Feature.ownerScope xmi.value = 'classifier'/>
      <Foundation.Core.StructuralFeature.type>
        <Foundation.Core.DataType/>    <!-- float -->
      </Foundation.Core.StructuralFeature.type>
    </Foundation.Core.Attribute>
    </Foundation.Core.Classifier.feature>
    <Foundation.Core.Classifier.feature>
    <!-- =========     Math  [class]  GetResult  [Operation]     ========== -->
    <Foundation.Core.Operation xmi.id = 'id105'>
      <Foundation.Core.ModelElement.name>GetResult
      </Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.visibility/>
      <Foundation.Core.BehavioralFeature.parameter>
        <Foundation.Core.Parameter xmi.id = 'id105.Return' >
          <Foundation.Core.ModelElement.name>GetResult.Return
          </Foundation.Core.ModelElement.name>
          <Foundation.Core.Parameter.kind/>
          <Foundation.Core.Parameter.type>
            <Foundation.Core.DataType/>    <!-- float -->
          </Foundation.Core.Parameter.type>
        </Foundation.Core.Parameter>
      </Foundation.Core.BehavioralFeature.parameter>
    </Foundation.Core.Operation>
    </Foundation.Core.Classifier.feature>
    <Foundation.Core.Classifier.feature>
    <!-- ============     Math  [class]  Add  [Operation]     ============= -->
    <Foundation.Core.Operation xmi.id = 'id111'>
      <Foundation.Core.ModelElement.name>Add</Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.visibility/>
      <Foundation.Core.BehavioralFeature.parameter>
        <Foundation.Core.Parameter xmi.id = 'id111.Return' >
          <Foundation.Core.ModelElement.name>Add.Return
          </Foundation.Core.ModelElement.name>
          <Foundation.Core.Parameter.kind/>
          <Foundation.Core.Parameter.type>
            <Foundation.Core.DataType/>    <!-- float -->
```

```
          </Foundation.Core.Parameter.type>
        </Foundation.Core.Parameter>
          <Foundation.Core.Parameter xmi.id = 'id112' >
            <Foundation.Core.ModelElement.name>a
            </Foundation.Core.ModelElement.name>
            <Foundation.Core.ModelElement.visibility/>
            <Foundation.Core.Parameter.kind/>
            <Foundation.Core.Parameter.type>
              <Foundation.Core.DataType/>    <!-- float -->
            </Foundation.Core.Parameter.type>
          </Foundation.Core.Parameter>
          <Foundation.Core.Parameter xmi.id = 'id113' >
            <Foundation.Core.ModelElement.name>b
            </Foundation.Core.ModelElement.name>
            <Foundation.Core.ModelElement.visibility/>
            <Foundation.Core.Parameter.kind/>
            <Foundation.Core.Parameter.type>
              <Foundation.Core.DataType/>    <!-- float -->
            </Foundation.Core.Parameter.type>
          </Foundation.Core.Parameter>
      </Foundation.Core.BehavioralFeature.parameter>
    </Foundation.Core.Operation>
    </Foundation.Core.Classifier.feature>
    <Foundation.Core.Classifier.feature>
    ...
```

G The DTD of the extended SRML metalanguage

```
<!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
              reftype (text|attrib) "attrib">
<!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
                  reftype (text|attrib) "attrib">
<!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"
                reftype (text|attrib) "attrib">
<!ELEMENT if-any (expr, expr, expr)>
<!--cond,if,else-->
<!ATTLIST if-any element NMTOKEN "srml:all"
                attrib NMTOKEN "srml:all"
                reftype (text|attrib) "attrib">
<!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)>
```