XML I/O IN ROOT

S. Linev[#], H. Essel, GSI, Darmstadt, Germany
R. Brun, FNAL, CERN, Geneva, Switzerland

Abstract

This paper describes recent development in XML I/O in ROOT, which allows storing and retrieving objects from XML files. XML format makes' easy to view and edit the objects data directly. Several implementation aspects and usage of ROOT XML files as exchange format between different applications are discussed.

MOTIVATION

XML (eXtensible Markup Language) provides well structured text data format, which can be viewed and edited with standard text editing tools. Now XML is widely used as a storage format for different kinds of metadata likes configurations, parameters, conditions, geometries etc. Another application field for XML is exchange format between different platforms and frameworks. But most frameworks implement XML I/O by themselves, frequently introducing new definition for XML file structures.

The ROOT framework has a powerful I/O system, but had support for binary file format only to store and read objects. Without the ROOT environment the data stored in such files were not directly accessible. Therefore one decides to provide possibility of XML I/O to ROOT.

IMPLEMENTATION

Used XML package

There are general-purpose XML packages, which provide wide range of functions for manipulation with XML files. One can mention several C/C++ based packages:

- expat from Mozilla project [1],
- libxml from Gnome [2],
- Xerces-C++ from Apache [3].

Typically such packages make parsing and validation of XML files and provides full access to each element of XML tree. Performance benchmark for these and several other packages is shown on Figure 1 [4]. It shows execution time for different operations like parsing XML files, building document in memory, validating XML files with DTD and so on.

These packages have obviously different performance. They also differ in support of advanced features like XML schema, XSLT and so on. With regard to performance and functionality requirements libxml2 package was chosen to be used in ROOT.

[#]S.Linev@gsi.de

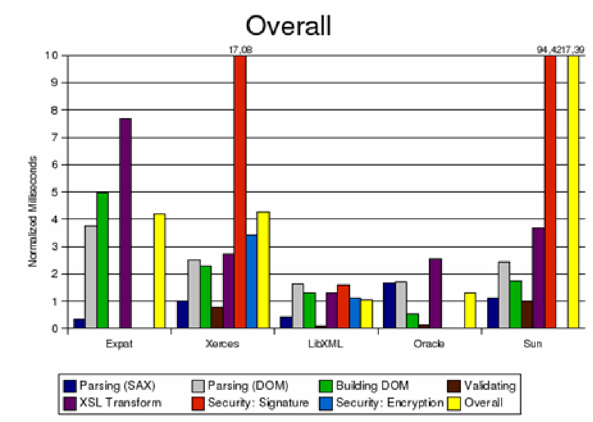

Figure 1: Benchmark between XML packages.

New classes in ROOT

A TXMLEngine class was introduced to provide a narrow interface to libxml2 library. This keeps us a possibility to implement similar interface with other XML packages if required.

Main changes were done in TBuffer class. Most of its functions, which translate basic data types to binary representation, were declared virtual. Derived TXMLBuffer class was introduced to allow translation of such basic types to XML nodes and attributes. Making a lot of functions virtual leads up to 1÷5% performance penalty in case of intensive I/O application.

The TXMLFile class, inherited from TFile, provides standard ROOT interface to file, where several objects can be stored.

One should mentions that before any data can be read from any XML file, all XML structures from this file will be allocated in memory. This imposes limitation on the size of file. Typically, file size should not exceed limit of 100 MB.

A short example to create XML file is shown on Figure 2.

```
void example() {
   TNamed named("ObjectName","ObjectTitle");
   TBox box(0.0, 0.0, 1.0, 1.0);
   TFile* fxml=TFile::Open("example.xml", "recreate");
   named.Write("named");
   box.Write("box");
   delete fxml;
}
```

Figure 2: Script to create XML file.

Example shows usage of static TFile::Open() method, which internally invokes ROOT plug-in manager. If name of file contains “.xml” extension, TXMLFile instance will be created.

Since May 2004 XML classes are included in ROOT distribution.

Supported features

“Native” ROOT I/O interface is provided. This means, that most of user I/O code should not be touched. Only creation of TXMLFile instance should be done exlusevely or via TFile::Open() method.

Except very special objects, like TTree and TBranch, most of ROOT classes are supported by XML I/O.

Classes with custom streamers are also supported. The limitation in that case is, that reading and writing sequence should correspond to each other. If user writes array of integer values, array of integer should be read and not array of bytes or doubles. In fact, a number of ROOT classes use custom streamers, therefore support of custom streamers was strongly required.

Two different layout options are supported – generic and class-specific. In first case only several node names are allowed: “Object”, “Class”, “Member”, “Item” (see Fig. 3). All class-dependent information is coded in nodes attributes. For that case DTD (Document Type Definition) is short and simple. In second case names of XML tags are derived from class names and class members names. This makes XML more clear and readable. DTD, generated for such file, includes special part for each used classes, and provides very strict syntax check by standard XML tools. Example of class-specific layout with enabled namespaces is shown on Figure 4.

```
<Object class="TBox">
  <Class name="TBox" version="2">
    <Member name="TObject">
      <Item name="Version" v="1"/>
      <Item name="UInt_t" v="0"/>
      <Item name="UInt_t" v="50331648"/>
    </Member>
    <Member name="TAttLine" version="1">
      <Member name="fLineColor" v="1"/>
      <Member name="fLineStyle" v="1"/>
      <Member name="fLineWidth" v="1"/>
    </Member>
    <Member name="TAttFill" version="1">
      <Member name="fFillColor" v="19"/>
      <Member name="fFillStyle" v="1001"/>
    </Member>
    <Member name="fX1" v="0.000000"/>
    <Member name="fY1" v="0.000000"/>
    <Member name="fX2" v="1.000000"/>
    <Member name="fY2" v="1.000000"/>
  </Class>
</Object>
```

Figure 3: Example of generic layout (TBox class).

```
<Object class="TBox">
  <TBox xmlns:TBox="http://..." version="2">
    <TObject fUniqueID="0" fBits="3000000"/>
    <TAttLine xmlns:TAttLine="http://..." version="1">
      <TAttLine:fLineColor v="1"/>
      <TAttLine:fLineStyle v="1"/>
      <TAttLine:fLineWidth v="1"/>
    </TAttLine>
    <TAttFill xmlns:TAttFill="http://.." version="1">
      <TAttFill:fFillColor v="19"/>
      <TAttFill:fFillStyle v="1001"/>
    </TAttFill>
    <TBox:fX1 v="0.000000"/>
    <TBox:fY1 v="0.000000"/>
    <TBox:fX2 v="1.000000"/>
    <TBox:fY2 v="1.000000"/>
  </TBox>
</Object>
```

Figure 4: Class-specific layout with namespaces.

DATA EXCHANGE

Main motivation for this development was the possibility to exchange data between different kinds of applications. Let’s consider different approaches.

Communication between ROOT applications

This is a situation when XML I/O can be applied directly. Both ROOT applications require only to have same dictionaries. The advantage compared to standard ROOT I/O is that data in XML files can be viewed and edited. Of course it should be taken into account that big data volumes can be problematic in XML files.

Import data from C++ program

In that case one would like to get data from C++ application, which internally does not use any ROOT classes and ROOT I/O. This can be done with special TXMLPlayer class.

The algorithm how user can get possibility of XML I/O in it’s application shown on Figure 5.

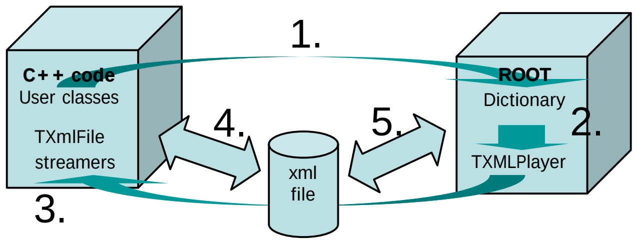

Figure 5: Generation of XML streamers.

Algorithm consists of following steps:

1. Generate ROOT dictionary of user classes
2. Using TXMLPlayer class, create streamers for user classes
3. Add generated streamers and TXmlFile class to user project
4. Use “ROOT-like” interface to access XML files from user application
5. At the same time ROOT can access data from XML files, using classes dictionary

Generated streamers provide functionality to serialize/deserialize object data. Only following data members in user classes are supported:

- basic data types, arrays, const char*
- objects, object pointers
- array of objects, array of objects pointers
- STL string, vector, list, deque, set, map, multimap

No any dictionaries or TClass objects are used inside streamers. The streamer resolves class inheritance tree using only standard C++ type_info information.

Streamer function does not appear as method of appropriate class, therefore they have special treatment for private and protected data members. This requires knowledge how object data is represented in memory, therefore such code cannot be directly ported on other platform. User should have different code to read XML files under Windows or Linux or Solaris, but the format of XML file is preserved for any platform.

An appropriate example is avaliable at [5]. It provides a makefile for generation of streamers and includes code to access XML files from user application. Example includes several classes which demonstrate most of supported data members.

Export data from ROOT application

This situation supposes that the user has a running ROOT-based program with developed I/O for its own classes. Most probably, these classes include or inherit ROOT classes like TObject, TString, TObjArray, etc. The problem is to access this data from an application which doesn’t know any ROOT classes and doesn’t include any ROOT I/O libraries.

One probable solution is avoids the usage of ROOT classes inside user classes. This leads us to previous case, but this cannot be considered as general solution. One needs a possibility to deal with data derived from ROOT classes outside ROOT.

Currently there is no solution for such case, but some ideas can be considered. One can generate class definitions for user code, where dependency from ROOT is excluded and ROOT containers (TString, TArray, TList) are replaced by correspondent STL containers (stl::string, stl::vector, stl::list). Of course, one cannot support all ROOT classes, especially like TTree, but for a finite classes subset implementation can be done. Investigation of real user requirements should be done before any code can be implemented.

Access to existing non-ROOT XML files

Frequently one needs to access data in existing XML files, which are not created by ROOT and imposes another structure of XML tags and attributes. This is the most demanding problem, while there is no general solution for that case. One can support some of typical and simple cases only, e.g. the XML file contains structures like telephone book or library catalogue.

In such case one can try to declare a class which corresponds to the structure of XML file. Probably one needs a certain level of “customisation” of XML-file structures to allow assignment between data members and names of XML tag where these members are stored. Also required is a functionality to read/write such structures directly, without unnecessary ROOT overhead like list of keys, class version number and so on. Further investigations and user suggestions are still required.

CONCLUSION

Generic XML I/O was developed for ROOT. It includes support of data exchange with other non-ROOT applications. Further developments are necessary to fulfill all user requirements.

REFERENCES

[1] http://expat.sourceforge.net
[2] http://xmlsoft.org
[3] http://xml.apache.org/xerces-c/
[4] http://xmlbench.sourceforge.net
[5] http://www-linux.gsi.de/~linev/reader.tar.gz