GEOMETRY DESCRIPTION MARKUP LANGUAGE AND ITS APPLICATION-SPECIFIC BINDINGS

W. Pokorski, R. Chytracek, CERN, Geneva, Switzerland
J. McCormick, SLAC, Stanford, USA
G. Santin, ESA, Noordwijk, Netherlands

Abstract

The Geometry Description Markup Language (GDML) is a specialized XML-based language designed as an application-independent persistent format for describing the detector geometries. It serves to implement 'geometry trees' which correspond to the hierarchy of volumes a detector geometry can be composed of, and to allow to identify the position of individual solids, as well as to describe the materials they are made of. Being pure XML, GDML can be universally used, and in particular it can be considered as the format for interchanging geometries among different applications. In this paper we will present the current status of the development of GDML. After having discussed the contents of the latest GDMLSchema, which is the basic definition of the format, we will concentrate on the GDML processors. We will present the latest implementation of the GDML 'writers' as well as 'readers' for either Geant4 or ROOT.

INTRODUCTION

The geometry description is the essential part of every simulation application. It is the most 'detector specific' element of the program and usually it is the most tedious to implement. In the same time it is often the case that the same geometry description needs to be used in several applications, either different simulation engines (for the purpose of physics validation and comparison) or other applications like visualization, reconstruction, analysis, etc. Those applications come with their native, different geometry description formats, which forces users to either reimplement their geometry for each of the applications or to come up with some automatic conversion mechanisms. It is indeed the case that users rarely implement their geometries in one of the simulation toolkits geometry formats. They usually come up with their own formats, often XML-based, which prove to be more flexible and allow reuse of the geometry in the different parts of the event processing chain. The drawback of such a solution, however is the fact that the geometry descriptions become strongly linked with the experiments' software frameworks and therefore cannot be easily exported and used in stand-alone applications.

It seems justified, therefore, to propose a geometry description language which would allow, from one side 'application-independent' way of implementing new geometries, and from the other side, would provide and exchange format for the already existing geometries.

Geometry Description Markup Language (GDML) has been developed as an application of XML. This has been motivated by a number of reasons, namely simple reading and writing mechanism, extensibility, widely used and application independent format. Moreover, as GDML is meant to describe geometry data, the choice of a markup rather then procedural language seems to be natural. The fact that a GDML file can be easily edited using any text editor, is also of a big advantage as it allows to easily introduce any changes in the particular geometry description.

The sections of this paper are organized as follows. We will first introduce the different components the GDML machinery is made of. We will then concentrate on discussing more in detail the structure of GDML schema as well as the GDML readers and writers. Finally, before concluding, we will demonstrate how GDML can be used, and illustrate it with a few examples.

GDML COMPONENTS

When talking about GDML, one can first of all distinguish on one side the XML definition part, or in other words a set of rules defining the structure of any GDML document as well as the list of the legal elements, and on the other side, the GDML generating and processing code.

The structure of a GDML document is defined through a set of XML Schema Definition (XSD) files which are an XML-based alternative to Document Type Definition (DTD) and which we call GDML schema. Any GDML geometry file must be valid with respect to the GDML schema.

The GDML file itself, can be either written by hand (in case GDML is used as the primary geometry source) or generated automatically out of the application specific 'in-memory' geometry tree using one of the GDML 'writers' called by the user application (when GDML is used as an exchange or persistency format). The GDML reader is responsible for parsing the GDML file and creating the in-memory representation of the geometry tree specific for the user application. The overall picture of different GDML components is shown in Figure 1.

GDML SCHEMA

The GDML schema is a set of XSD files which define the structure of the GDML document and its legal elements. The general structure of the GDML file can be seen on Figure 2. One can distinguish there five parts, each holding specific type of data.

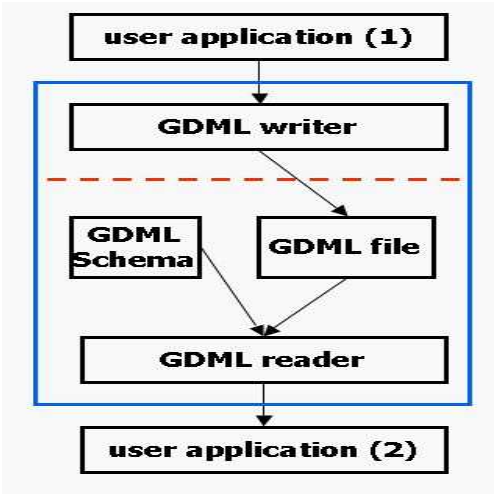

Figure 1: GDML components.

The first part 'define' contains numerical values of different constants, positions and rotations that will be used later on in the geometry construction.

The second part 'materials' contains definitions of all the materials used in the given geometry. The supported forms are simple materials (made from one element) as well as mixtures. Mixtures can be composed on the basis of fraction of mass or atom count.

The third part 'solids' is the collection of all solid definition which are used in the given geometry description. The presently supported solids are box, sphere, tube, cone, polycone, parallepiped, trapezoid, torus, polyhedra, hyperbolic tube, elliptical tube and ellipsoid. New solids are constantly being added to that list. Composite solids made using boolean operation (union, subtraction, intersection) are also supported.

The forth part of the file contains the actual implementation of the geometry tree together with the assignment of solids and materials. The hierarchy of volumes is defined by specifying the daughter volumes (physvol) positioned inside a volume. Constructions like assembly volumes, reflections, replicas and division are possible. There is also basic support for parameterized volume which will be extended in the future.

Finally, the last part of the file called 'setup' serves to specify the top volume of the geometry tree. It is possible to define several 'setups' within one file, allowing to test different subparts (or different configurations included in the same file) of the geometry tree without changing the GDML file.

GDML READER AND WRITER

The role of the GDML reader is to parse the GDML file, validate it against the GDML schema (unless the validation is switched off or not available in the given parser) and create the in-memory representation of the geometry specific for the given application. The design of the reader (see Figure 3) is such that most of the processing is done within an application-independent XML engine based on SAX parser. The instantiation of the actual geometry objects is done by a light application-specific binding. The user interacts only through a very simple interface which returns to him the pointer to the top volume of the geometry tree with all the XML processing hidden. The presently available bindings are for Geant4 [2] and ROOT [3] geometry models. Due to the modularity of the design, it would be straightforward to add any additional bindings if required in the future.

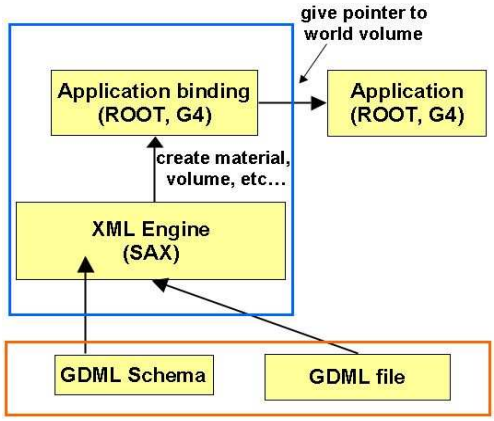

Figure 3: GDML reader.

The GDML writer functionality is to generate GDML files out of the 'in-memory' geometry trees. The application-specific binding (see Figure 4) scans the user defined geometry and prepares its representation in a application-independent way which can then be exported in the form of the GDML file. As in the case of the GDML reader, the application-specific binding is very light and most of the code is within the application-independent 'document builder'. The user interaction with the GDML writer is also very simple and consists of calling one method (WriteGeometry) passing the pointer to the top volume of the geometry tree as argument. User does not deal directly with the actual generation of XML file.

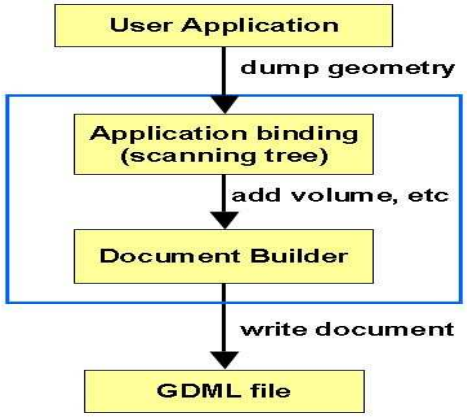

Figure 4: GDML writer.

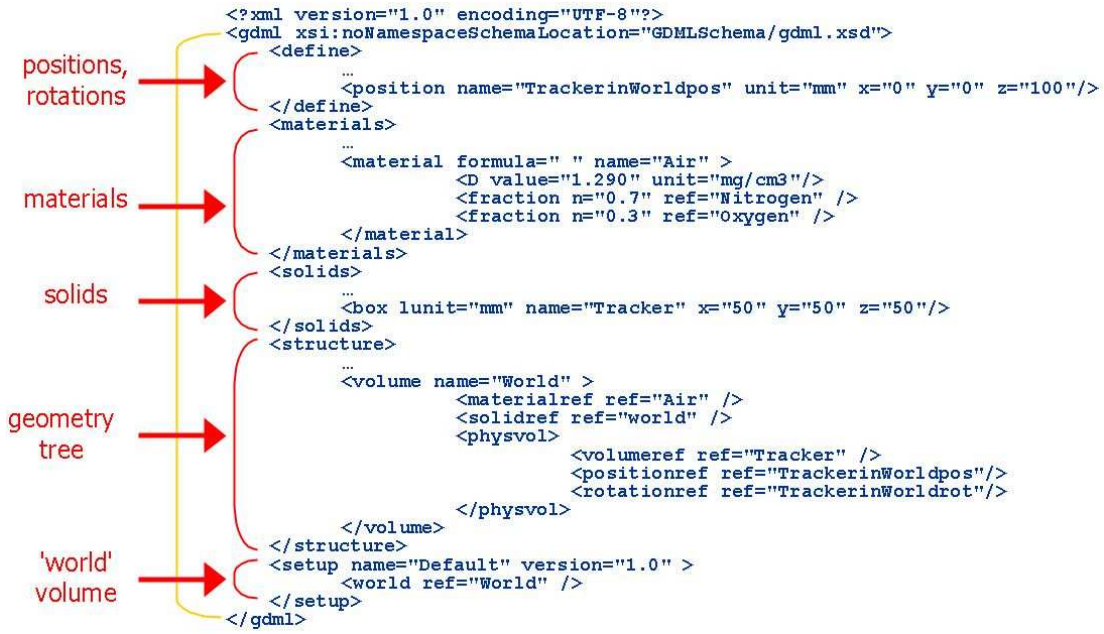

Figure 2: GDML file.

There have been two implementation approaches used for the GDML readers and writers. One was to implement it directly in C++ and second was to use Python together with specific Python bindings for particular C++ classes. The advantage of Python is that dealing with the XML parsing is more natural and easier than in C++ and therefore less code is needed. Moreover, Python turns out to be a very convenient tool for interfacing different applications together. It provides a nice environment for interactive simulation and analysis.

As far as the C++ implementation is concerned, the GDML reader and writer have been implemented for Geant4. These are used for Geant4 C++ applications where GDML is used as geometry source or when user wants to export the geometry from his native Geant4 application.

The GDML reader and writer for ROOT have been implemented in Python. It uses PyROOT [3] binding for ROOT classes. As mentioned in the previous paragraph, the choice of Python for that implementation significantly reduced the length of the processing code. Moreover, accessing ROOT from Python makes the integration with other applications easier and more interactive. Based on the same common structure, there has also been GDML reader for Geant4 implemented in Python. It uses the Reflex [3] tool for creating dictionary for the Geant4 classes which is then loaded into ROOT and allows interaction with those classes from the Python prompt. Such an approach allows to run interactively Geant4 and ROOT from Python using the same GDML file as geometry source.

USING GDML

In this section we will briefly describe the way to use GDML readers and writers. We start with the GDML writer in for Geant4.

The only thing needed to export the geometry in the form of the GDML file is the pointer to the top volume of the geometry tree which is then used as argument for calling the appropriete method of the writer. This can be done for instance in one of the Geant4 'user actions' once the geometry has been closed. User needs to instantiate the writer (giving the name of the schema file and the name of the output file as arguments) and call the 'DumpGeometryInfo' method:

```
G4GDMLWriter writer("gdml.xsd", "g4test.gdml");
writer.DumpGeometryInfo(g4worldvolume);
```

The GDML file is then generated and saved on the disk.

To use the GDML reader for Geant4, the procedure is the following. First, one has to instantiate, initialize and configure SAXProcessor (where 'mygeometry.gdml' file should correspond to the GDML file to be read):

```
SAXProcessor sxp;
sxp.Initialize();
ProcessingConfigurator config;
config.SetURI( "mygeometry.gdml" );
sxp.Configure( &config );
```

and then one has run the parsing and retrieve the pointer to the top geometry volume:

```
sxp.Run()
fWorld =  (G4VPhysicalVolume *)
GDMLProcessor::GetInstance()->GetWorldVolume();
```

This can, for instance, take place in the 'Construct()' method of the 'UserDetectorConstruction' class where the top volume is returned.

To use Python writers and readers, the procedure is very similar. We will briefly discuss here the use of GDML reader for ROOT and due to lack of space, skip the discussion of the GDML writer.

Assuming we have imported the ROOT module into Python and loaded the geometry library, we need just to import the generic XML parser and the GDML handler:

```
import xml.sax
import GDMLContentHandler
```

having done that, we can now instantiate the GDML handler and parse the file:

```
gdmlhandler =
  GDMLContentHandler.GDMLContentHandler()
xml.sax.parse('mygeometry.gdml', gdmlhandler)
```

The top volume of the geometry tree is now available from the handler:

```
topVolume = gdmlhandler.WorldVolume()
```

and can be passed to the ROOT geometry manager.

EXAMPLES

As mentioned in the Introduction, GDML can play two roles, it can be the language for the geometry implementation or it can be the geometry interchange format. It the first case, the use of GDML allows flexible geometry implementation which can be modified or exchanged without the need to recompile the application. GDML has been, for instance, the choice for the geometry implementation for Geant4 simulation application for Linear Collider experiments [4], space research [5] or medical physics [6].

As far as the geometry interchange aspect of GDML is concerned, it plays an important role for the LHC experiments, where physics validation of simulation toolkits is performed with the help of it. The original test-beam geometry is exported in the form of GDML files and then loaded into different stand-alone simulation applications allowing validation of particular aspects of simulation toolkits. GDML is also proving very useful in interchanging geometries between Geant4 and ROOT. An example of that can be seen on Figure 5 where the whole CMS detector has been visualized using ROOT 3D graphics with the geometry source being GDML file generated using the Geant4 GDML writer.

CONCLUSIONS

GDML is an XML-based, application-independent detector geometry description language designed for the purpose of simulation applications as well as analysis. It can be used as the principle geometry implementation format or it can play the role of an interchange format. The advantages of using GDML for geometry description implementation is the fact that it avoids hard-coding of the geometry,

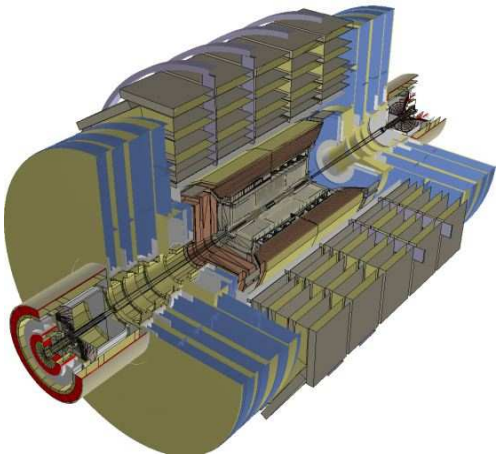

Figure 5: CMS detector visualized using ROOT from an automatically generated GDML file.

as well as it allows easy modifications and reconfiguration of the geometry used for the specific application. As far as geometry interchange format is concerned, GDML allows to extract geometry description from experiment-specific frameworks and use them in generic application for the purpose of physics validation and comparison. It also allows to move geometries between Geant4 and ROOT for the purpose or visualization using ROOT 3D graphics.

As far as the future developments are concerned, it is planned to further extend support for different solids available in Geant4, for instance the family of twisted solids recently introduced there. Work is also foreseen to enable modularization of GDML files in order to be able to load different subdetectors independently.

REFERENCES

[1] R. Chytracek, "The Geometry Description Markup Language", in proceedings of CHEP 2001, p. 473-476, Beijing, China. URL: http://cern.ch/gdml

[2] S. Agostinelli et al., "Geant4: a simulation toolkit", Nucl. Instrum. Meth A506, 2003, p. 250. URL: http://cern.ch/geant4

[3] URL: http://root.cern.ch/

[4] J. McCormick, "Full Detector Simulation Using SLIC and LCDD", SLAC-PUB-11418, Aug 18, 2005. 5pp. Contributed to 2005 International Linear Collider Workshop (LCWS 2005), Stanford, California, 18-22 Mar 2005. URL: http://www.lcsim.org/software/lcdd/

[5] G. Santin, V. Ivanchenko, H. Evans, P. Nieminen, E. Daly, "GRAS: A general-purpose 3-D modular simulation tool for space environment effects analysis" , IEEE Trans. Nucl. Sci. 52, Issue 6, 2005, pp. 2294 - 2299. URL: http://geant4.esa.int

[6] G. Guerrieri, Development of anthropomorphic models for radiation protection and radiotherapy, Thesis, University of Genova, 2005
R. Capra, S. Chauvie, Z. Francis, S. Guatelli, S. Incerti, B. Mascialino et al., Geant4 capabilities for microdosimetry simulation, Submitted to Radiation protection and Dosimetry (Proc. Suppl.), 2006