	User Manual and Guide to developing geometries with the GeoModelXml Package

ATLAS Project Doc. No: **None**	*Institute Doc. No:* **None**	*Created:* **8/8/2013** *Modified:* **16/5/2014**	*Page:* **1 of 21** *Rev:* **C**

Note

User Manual and Guide to developing geometries with the GeoModelXml Package

The package GeoModelXml simplifies the creation of ATLAS geometries in Athena. It allows the geometry to be defined in an xml file, which is parsed and used to create the GeoModel tree. This document is aimed as a user guide and to help future maintenance of the code.

Prepared by: **N.P. Hessey**	*Checked by:* **Nobody**	*Approved by:* **Nobody**

Distribution List:
Public

History of Changes

Rev. No.	*Date*	*Pages*	*Made by*	*Description of changes*
A	5/8/2013	All	N.P. Hessey	Original document
B	14/4/2014	Many	N.P. Hessey	Better installation instructions.
C	16/05/2014	Many	N.P. Hessey	Map not vector for position indexes; G4Hits in ITk explained; can use position index names in formulae.

Contents

1 Introduction

The ATLAS software framework Athena uses GeoModel [1] as the central geometry store. The geometry is stored in a graph (or tree), and a set of objects (materials, shapes etc.) which are managed by GeoModel. The graph has a collection of “physical volumes” of some shape with dimensions and filled with a material. These can contain other physical volumes. They have names and integer identifiers, as well as transformations giving their position. These transformations can be changed during program execution, to allow for movements of detector elements and for software alignment of components. This manual assumes familiarity with GeoModel, as well as a rudimentary knowledge of xml.

The objects and tree are built up by C++ code calling GeoModel methods. Most of ATLAS geometry is currently built up using C++ code directly, while the muon non-sensitive geometry (services, supports, magnets etc.) is built up in AGDD [2] which is an xml-based geometry description; C++ routines parse the xml and use generic code to build up the GeoModel geometry.

For studies of future Inner Tracker layouts, several geometries need to be explored. To speed these studies up, a package GeoModelXml has been written. This package contains a Document Type Definition (DTD) file, and hopefully later an xml schema. These create an xml geometry-definition-language called gmx ("GeoModel in Xml") and provides a C++ package GeoModelXml to convert the gmx file into a GeoModel tree and the required GeoModel objects. The recommended filename extension for gmx geometry description files is .gmx, which helps distinguish the files from other xml.

There are several needs in Athena which are very closely linked to the geometry. These are the ATLAS Identifier scheme, the digitization process for Geant hits (“Readout Geometry”), and detector software alignment. GeoModelXml only handles the solid geometry: materials, shapes, logical volumes and physical volumes. Identifiers, Readout Geometry, and Alignment, are factored out and handled separately in other packages. The link between these and the geometry are steered in the xml: whenever a volume that is declared “sensitive” is created, two call-back routines are called: one to create a detector identifier, the other to set-up the digitisation routines to handle this volume; and whenever a volume declared “alignable” is created, a call-back routine is invoked to store the relevant information in the geometry manager for the software alignment transformations.

After a brief back-ground on xml, this manual has three sections. The first is for people who just want to modify or develop a geometry where someone has already set-up Athena to use GeoModelXml. The next explains the C++ code to make a “Detector Manager” that uses gmx. The third is to aid code maintenance, giving an introductory overview of how the code is organised.

2 Why use xml

The ATLAS Inner Detector geometry in Athena is currently defined directly in C++ code, with most (but not all) dimensions and quantities stored in the data base. The Pixel and TRT geometries are each defined in one long (a few thousand lines of code) C++ file. The SCT geometry is defined in several smaller files, with one class per type of item. The former system is difficult to maintain; the latter requires quite some effort to code new geometries. The SCT method makes something very clear though: we basically do the same thing over and over in building up geometries, giving hope of automating this with some C++ code, steered by a control file.

Xml (extensible markup language) is very well suited to geometry description: it can store numbers, such as dimensions, as well as their meaning; it can also handle hierarchical structures (such as Inner Detector has a barrel has cylinders have staves...). It can be extended with whatever tags the user needs. These and their allowed content are defined in a “DTD” file or alternatively in a “schema” file. Standard freely available software such as Xerces-C [3] can parse the xml, checking it follows both the rules of xml and the rules given in the DTD. Insisting the xml description file follows the DTD greatly simplifies programming: the code only has to check for things that cannot be enforced in the DTD/schema. Furthermore any errors are found very early, and their position can be pin-pointed with error messages. This leads to fast development of a geometry.

If we can find some effort to also develop code to automate handling of ATLAS identifiers, digitization, and alignment, one could arrive at a highly flexible Athena geometry system: no need to compile-link-run to develop layouts; just modify the xml, run, and look at the results.

For geometry versioning, the xml file(s) can be stored in the conditions data-base. Some people have stated a preference for storing them and versioning them in SVN, which could also be made to work.

3 Guide to Developing a Geometry in the gmx file

3.1 Overview

The gmx file defines materials, shapes, logical volumes (logvols, which are a combination of a shape and a material filling it), and tranformations which place copies of the logvols in the desired position.

Dimensions can be defined in terms of parameters and arithmetic expressions involving those parameters. These parameters are dealt with first, creating them with their value. A few other details are also dealt with at the start: digitization-schemes and position index names (explained later).

The next step is very important to understand. Processing then skips everything in the file until the single <addbranch> tag at the end. The content of the addbranch is processed to build up the geometry in a branch of the GeoModel tree. If this content refers to other objects, those are found and processed. Similarly their children are processed, and so on iteratively until the addbranch contents are completed. So for example you can build up a large <materials> section and include it in all your projects. Only the materials and elements that you actually use in your geometry via the <addbranch> element will be created. You will not waste resources (neither memory nor cpu) during execution of Athena by having a large material file. Unused materials, shapes, logvols etc. will never be turned into GeoModel items - they can sit in the gmx ready for future use, though of course can clutter your gmx file. Note that while this means the file is processed in an apparently random order, the file must obey xml rules. These include the rule that if an attribute references another element, that element must have already been given. So the <addbranch> element has to come last. A logical order for the other tags would be defines, materials, shapes, logvols/assemblies, then the addbranch. However to allow flexibility in file inclusion and re-use, this is not imposed.

There are no “PhysVols” in the gmx file. Instead, each time a logvol is encountered during processing, a GeoPhysVol (or GeoFullPhysVol, if the logvol is alignable or sensitive) is created and added to the tree. For all tags that create GeoModel objects, the code is careful to only create one version, and use it many times. So the first time a logvol is encountered, a GeoLogVol is created followed by a Geo(Full)PhysVol which is added to the GeoModel tree. A pointer to the logvol is also stored in a list maintained by the logvol processor. This list (an STL map in C++ terms) is indexed by the logvol name. On a subsequent use of the logvol, it is found, via the name attribute, in the map and used in a Geo(Full)PhysVol, without recreating the logvol.

Several tags use this mechanism. This is the reason why many tags have mandatory names: even if the user does not need to refer to the name, it is needed for insertion in the map. Apart from this, the names allow the user to reference items, and make the gmx more readable.

Geometries in ATLAS can be large and complicated. Different parts may be managed by different people. Furthermore, some parts may be wanted in several geometries. To help keep things maintainable, and allow re-use (especially of materials) you are recommended to split the input in several files. Flexibility above the minimum strictly necessary has been built in to the DTD to allow sensible distribution of the gmx over several files. This can be done with the standard xml mechanism: one sets up the main file (first file called) with the usual xml header, giving the DTD file; then create local entities with the names of sub-files to be included. When the parser encounters this entity (surrounded by & and ;), it will switch to reading input from that file.

Look in the DTD file in the package data directory for all details of what is and is not allowed. The following sections do not attempt to cover all possibilities and restrictions, but rather give a quick start to developing good gmx files.

3.2 Debugging a gmx file; gmx2geo

There are various xml checkers around, which can give a very quick check of your gmx files. However, I have not found one that easily allows multiple input files, DTDs and schemas, and they do not necessarily pin down error positions as well as possible.

Once things are more or less set up, running VP1 in Athena is a powerful tool to see if you have created what you intended. But this is rather a slow way to find you misspelt an identifier or some other minor mistake. So I have developed a standalone way of checking gmx files: a program called gmx2geo. Once installed, it takes a second or so to find and pin-point the first error in the gmx file, which is very useful and saves a lot of development time. In the absence of xml errors, it goes on to build up the full GeoModel tree, and then traverse it, printing out a very verbose list of your geometry. Not generally useful, but can be helpful for difficult problems and for debugging GeoModelXml itself.

3.2.1 Installing gmx2geo

The program needs (in this order):

1. Xerces-C version 3.1: standard installation
2. CLHEP: standard installation
3. GeoModelKernel: Download with svn (it is in the ATLAS offline repositories, under DetectorDescription/GeoModel/GeoModelKernel). Also download the makefile at ???? into the

src directory, and do `make`.

4. GeoModelXml headers and a shared library of all .o files: You can use your standard Athena install, and run `make` in the src directory; or (for non-Athena computers) download using svn (from `svn+ssh://svn.cern.ch/reps/atlas-hessey/GeoModelXml`), edit the makefile in the src directory – especially (and hopefully only) the PROG variable – and do a `make` in the src directory.

5. gmx2geo: Download with svn (from `svn+ssh://svn.cern.ch/reps/atlas-hessey/gmx2geo`). In the src directory edit the makefile especially the PROG definition, and do `make`.

3.2.2 Running gmx2geo

In principle, copy the executable gmx2geo from the src directory to somewhere in your PATH, and just type

```
gmx2geo <filename>.gmx
```

In practice, you probably want to pipe it to less or send the output to a file for later browsing.

3.3 Basic gmx file contents

Example of the start and end of a gmx file, with use of including materials from another file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE geomodel SYSTEM "/path_to_GeoModelXml/trunk/src/geomodel.dtd" [
<!ENTITY materialFile SYSTEM "materials.gmx"> <!-- Internal DTD stuff-->
]>
<!-- The xmlns attribute says un-named namespace things are gmx: -->
<geomodel name="Example to illustrate basic structure of a gmx file"
  version="1.0" xmlns="http://www.nikhef.nl/\%7Er29/gmx">

<!-- Defines etc. here -->

&materialFile;

<!-- lots more stuff building up the logvol called SCT_logvol -->

<addbranch>
    <logvolref ref="SCT_logvol"/>
</addbranch>
</geomodel>
```

This assumes that a file called materials.gmx exists in this directory too. Usually xml files access the DTD on the web with http: access. However this is disabled in the Athena version of Xerces-C used for the xml processing. This restriction is probably a good idea: for versioning, we need to store the DTD and all xml files in the data base. Loading content from the web cannot be versioned properly. So you must use the DTD in the data directory in the package.

The DTD is stored in svn in the GeoModelXml package trunk/data directory. With cmt installation, there is no trunk directory.

3.4 Units

Lengths are in mm; angles in radians; densities in g/cm^3 (same as kg/m^3) and internally are converted to be consistent with Athena units.

3.5 Parameters and evaluation of numeric expressions

For maintainability and readability, it is much better to assign a meaningful name for a number, and use the name rather than the number directly. Furthermore it is very useful to be able to derive other dimensions from a basic given one. Gmx therefore allows the definition of named parameters with a numerical value. These parameters can be used in numerical expressions to derive other numerical values.

The expressions are CLHEP-Evaluator expressions [4]. Common mathematical functions are allowed; e.g. the following can be useful:

```
<var name="PI" value="acos(-1.0)"/>
```

The definitions are given inside <define> tags, and can be <var>, <vector>, or <matrix> values. You are recommended to make the defines all at the start of the gmx file in which they are used, or in a separate file or files to be included in the main file. To allow this, <define> sections can be distributed throughout the gmx file at the top level (i.e as direct children of the geomodel node).

All the parameters should be considered as constants (despite any suggestion from the tag name "var"). Their values are set in the order they appear in the gmx file. Thus an expression can only use previously declared parameters. <define>s are the first thing processed. This is the safest way! Xml is not a procedural language, and the xml file is not processed top to bottom (see overview section); this makes it unpredictable to use any truely "variable" parameter system. So gmx like the following is forbidden (the second <var> element attempts to re-use the unique xml-ID name-attribute), but in anycase would not have produced what the user presumably wanted:

For now, avoid white space after the last number in a matrix or vector list, or you will create an extra matrix or vector element. (To be fixed sometime).

```
<defines>
    <var name="InnerRad" value="100."/>
</defines>
<shapes>
    <tube name="HollowTube" rmin="InnerRad" rmax="200." zhalflength="400."/>
</shapes>
<defines>
```

```
    <var name="InnerRad" value="0."/>
    <!-- illegal! Second occurrence of InnerRad -->
</defines>
<shapes>
    <tube name="SolidTube" rmin="InnerRad" rmax="200." zhalflength="400."/>
</shapes>
```

would produce two *identical* (solid) tubes, one with a misleading name HollowTube. The last value given, 0.0 here, is the one that would be used everywhere for InnerRad, if this construct had been allowed in the DTD.

Three types of parameters can be defined: atomic constants; vectors; and matrices. Internally there are only atomic variables; a “vector” tag with name vecvar and dimension 3 creates three variables called vecvar_0, vecvar_1, vecvar_2. A matrix named m is converted to variable m_0_0 etc. with the first digit giving the row and the second giving the column; the user uses these extended names later in the gmx file to refer to the values.

Matrix and vector values must be pure numbers, separated by white space; the values are not passed through any evaluator. (Possibly in the future this could change, e.g. giving a list of semi-colon separated expressions to be evaluated.)

Example:

```
<defines>
    <var name="PI" value="acos(-1.0)"/>
    <var name="TWOPI" value="2 * PI"/>
    <vector name="UnitK" value="0 0 1"/>
    <matrix name="twoCols4Rows" coldim="2" value="0 1   2 3   4 5   6 7"/>
</defines>
```

You are recommended to implement some sort of namespace system for variable names, and avoid common names like xLen, to avoid clashes. E.g. precede all your variables with a three letter acronym followed by an underscore.

3.6 Adding geometry branches

The <addbranch> tag is where it all starts, as soon as the <defines> have been done. Typically the <addbranch> contains one or a few references to some object, say a <logvolref>. Processing starts at the first such reference, and creates it. If that contains children, they in turn are processed. When a logvol is first encountered, its material and shape are created (if not already created for some other logvol).

GeoModelXml allows you to chose any (valid xml ID entity) name you like for a logvol etc. in <addbranch>. However Athena and VP1 have conventions used with this name: they understand the names Pixel, SCT, TRT among others; and for children of SCT, Barrel, ForwardPlus, and ForwardMinus (Endcap-A = ForwardPlus) are understood.

3.7 Materials

These follow very much the GeoModel manual description. Elements are created with name and shortname and given Z and A (atomic number and atomic weight). Materials are given a name and density (in g/cm^3), and contain mass-fractions of elements and other materials. For example:

```
<materials> <!-- Fractions are by mass -->
  <element name="Hydrogen" shortname="H"  Z="1"  A="1.0079"/>
  <element name="Carbon"   shortname="C"  Z="6"  A="12.011"/>
  <element name="Nitrogen" shortname="N"  Z="7"  A="14.007"/>
  <element name="Oxygen"   shortname="O"  Z="8"  A="15.999"/>

  <material name="Graphite" density="2.267"> <!-- Wikipedia Carbon -->
    <elementref fraction="1.0" ref="Carbon"/>
  </material>
  <material name="Bisphenol-A" density="1.2">
    <elementref fraction="0.03" ref="Hydrogen"/>
    <elementref fraction="0.75" ref="Carbon"/>
    <elementref fraction="0.10" ref="Nitrogen"/>
    <elementref fraction="0.12" ref="Oxygen"/>
  </material>
  <material name="CyanateEster" density="1.2">
     <!-- It's a secret what they actually use -->
     <materialref fraction="1.0" ref="Bisphenol-A"/>
  </material>
  <material name="K13D2U" density="1.820">
    <materialref fraction="0.721" ref="Graphite"/>
    <materialref fraction="0.279" ref="CyanateEster"/>
  </material>
</materials>
```

3.8 Shapes

All shapes available in the GeoModel manual are implemented. The tag-name and parameter attribute names are all lower-case, but otherwise are identical to the shape name and parameter names used in the GeoModel manual. The generictrap shape is also implemented but not yet in the GeoModel manual. Refer to the GeoModel manual for further details of shapes.

Shapes are enclosed in a <shape> tag. This facilitates checking that a reference is indeed a reference to a shape item.

Shapes can be combined into "boolean" shapes with <union> (addition of two shapes), <subtraction> (removal of second shape from first), and <intersection> (only the volume common to both shapes) elements. The content of all these is always two shaperefs separated by a transformation applied to the second shaperef. If the transformation is empty, it acts as the identity transformation.

Example including a dice with a hole from corner to diagonally opposite corner:

```
<shapes>
  <box name="BoxShape" xhalflength="50."
   yhalflength="70." zhalflength="90."/>
  <box name="Cube" xhalflength="5." yhalflength="5." zhalflength="5."/>
  <tube name="Tube" rmin="0" rmax="1." zhalflength="10."/>
  <subtraction name="DiceForANecklace">
    <shaperef ref="Cube"/>
    <transformation name="Rotate45Deg">
      <rotation xcos="sqrt(2) / 2" ycos="sqrt(2) / 2" angle="PI/4"/>
    </transformation>
    <shaperef ref="Tube"/>
  </subtraction>
</shapes>
```

3.9 Logical volumes

The main stuff of gmx. Logical volumes (logvols) are a combination of a shape and a material, via attributes that give a reference to these things. They have a name so they can be referred to.

Unlike in GeoModel, logvols can contain other logvols. This works since the C++ GeoModelXml package positions and copies logvols into physvols, which can indeed contain other physvols.

In the following example, first a simple Box item is created. Then a large box is created to contain three copies of this box, spread along the z-axis, using <transform> elements. The Box is referenced via <logvolref> elements:

```
<box name="BoxShape" xhalflength="50" yhalflength="50"
  zhalflength="50"/>
<box name="BigBox" xhalflength="1000" yhalflength="1000"
  zhalflength="1000"/>
<logvol name="Box" material="K13D2U" shape="BoxShape"
  alignable="false"/>
<logvol name="ThreeBoxes" shape="BigBox" material="N2">

  <transform>
    <transformation>
      <translation z="-500"/>
    </transformation>
    <logvolref ref="Box"/>
  </transform>

  <logvolref ref="Box"/>
    <transform>
      <transformation>
        <translation z="+500"/>
      </transformation>
    <logvolref ref="Box"/>
  </transform>

</logvol>
```

Logvols by default are not alignable, which is generally suitable for services and support material, and lead to a GeoPhysVol element being created. Logvols that have sensitive detector children are often made alignable by setting the alignable attribute to true. Alignable logvols get turned into GeoFullPhysVols. These store their full position, allowing them to be moved around efficiently with software alignment of sensitive detector elements.

Another attribute of a logvol is “sensitive”. By default, logvols are insensitive material. If the sensitive attribute is given, it must reference a <digitizationscheme> element. These elements contain information such as number of strips and strip dimensions needed for digitization of Geant hits.

The child logvols of a logvol can be entered as new logvols, assemblies, transforms and ref-versions of these. See also the multicopy tag.

3.10 Assemblies

Assemblies are a way of collecting a group of logvols. The element content is the same as for logvols, and they can be alignable; however they cannot be sensitive. They can be manipulated as a whole (e.g. transformed) just like a logvol. The advantage of assemblies over logvols is that they need no material and dimensions specified. This avoids having to invent some non-existent shape with all its dimensions which is big enough to envelop all its content, yet without clashing with neighbouring volumes. They are converted by GEO2G4 into Geant4 G4Assembly items.

They can be included in logvols, and referred to by <assemblyref> tags.

In the following gmx snippet, two “PetalFaces” are combined to make a petal:

```
<assembly name="Petal" alignable="true">
  <transform name="NearFace">
    <transformation name="PlacePetalFaceNear">
      <translation z="-ISE_SensorZOffset"/>
    </transformation>
    <assemblyref ref="PetalFace"/>
  </transform>

  <transform name="FarFace">
    <transformation name="PlacePetalFaceFar">
      <translation z="ISE_SensorZOffset"/>
      <rotation xcos="1." angle="PI"/>
    </transformation>
    <assemblyref ref="PetalFace"/>
  </transform>
</assembly>
```

Internally, GeoModel recognises something as an assembly if its material is "special::Ether". GeoModelXml makes a single logvol called AssemblyLV containing special::Ether. All assemblies share this same logvol. This information can be useful when viewing with VP1, both to understand why such a logvol has been created without you asking for it, and to help manipulate viewing assemblies to make them visible or invisible. An unfortunate consequence of this is that many items in the tree end up called AssemblyLV making debugging harder.

3.11 Moving things around: Transformations and Transforms

Objects (logvols, assemblies, and shapes when combined into boolean shapes) need to be moved around. This is accomplished by applying transformations, which consist of a sequence of translations, rotations, and scalings in any order. First these are combined into a single HepGeom::-Transform3d, which is used to create a GeoTransform (or GeoAlignableTransform if the transform, transformation, and moved object are alignable).

The contents of a transformation are <translation>, <rotation>, and <scaling> elements. These have attributes to specify the transformation: *x*, *y*, *z* for translations; three direction-cosines specifying an axis of rotation through the origin *xcos*, *ycos*, *zcos* and the amount of the rotation in radians *angle* for rotations; and scale factors in each direction *x*, *y*, *z* for scalings (negative values give a parity transformation). All these attributes are set up to give an identity transformation: all 0 for translations and rotations; all 1 for scalings. So in general the user has to specify only a small subset of the attributes.

Generally supplying several simple successive transformations is easier to read and debug than trying to combine them yourself into a minimum number of transforms in the gmx file; they are combined internally into a single transformation, so there is no memory or cpu cost during Athena running by doing this.

A logvol or assembly can be moved by placing it in a <transform> element, along with the item to be moved (or a reference to such an item). See the Assembly section above for an example of this use. To avoid confusion between transform and transformation tags, think of transform! as a command to apply a transformation to an object.

The other uses of transformations are in building up boolean shapes (see the shapes section) and in <multicopy> elements (see next section).

3.12 Short-cuts for Multiple Copies

Xml is not a procedural language, and so does not have loop constructs, which tend to be very useful when making geometries in C++. It is not easy (nor wise) to implement loops in xml: look at the discussion above on why parameters are constants and think about what difficulties arise if you try to make a loop with a parameter depending on the loop counter, in an xml processor that does not process the xml serially from top to bottom. This lack of loop constructs tends to lead to quite some repetition in gmx files.

Some of this can be avoided using a <multicopy> element. Suppose you want several copies of a logvol to be added at regular translations (e.g. strip silicon sensors on a stave) or rotations (e.g. petals in a wheel). One could make an assembly with a transform object for each copy, incrementing the transformation in each transform. <multicopy> elements act as an abbreviation for this. One gives the transformation followed by the object to be transformed, and an attribute *n* giving the number of copies required. The first copy will be placed as-is (i.e. ”transformation to the power 0”); the second with the transformation applied once; and each subsequent copy has the transformation applied once more (“transformation to the power copy-number”).

Example to make a wheel of petals, assuming a “petal-pair” has been made consisting of two petals in a castellated layout:

```
  <assembly name="Wheel">
    <multicopy name="AddPetalPairsToWheel" n="16" >
      <transformation name="PlacePetalPairsToWheel">
        <rotation zcos="1." angle="2*PI/16"/>
      </transformation>
      <assemblyref ref="PetalPair"/>
    </multicopy>
  </assembly>
```

Note that the copy-number of each physvol added runs from 0 to n-1 every time the multicopy is encountered; i.e. in multicopy the copy number is automatically zeroed at the start of inserting the copies. This is almost always what is wanted. So there is no zeroid attribute.

Another common occurrence is to make several copies of an object at an irregular spacing, for example wheels in an endcap. multicopy has an attribute loopvar for this. If the loopvar attribute is given, its value must be the name of a vector defined in the defines section. Transformations can use the vector name as if it were a scalar value. For successive copies, the vector name will be set equal to successive values of the vector. E.g. to place wheels in an endcap:

```
  <defines>
     ...
     <vector name="WheelPos" value="1400. 1600. 2000. 2500. 3000"/>
     ...
  </defines>
...
  <assembly name="Endcap">
    <multicopy name="AddWheelsToEndcap" n="5" loopvar="WheelPos">
      <transformation name="PlaceWheel">
        <translation z="WheelPos"/> <!-- WheelPos is set to
                                         WheelPos_0, WheelPos_1,
                                         ...WheelPos_4 in turn -->
      </transformation>
      <assemblyref ref="Wheel"/>
    </multicopy>
  </assembly>
```

3.13 Positional Indices

There are several concepts of “neighbours” and “order” which are not strictly part of the solid geometry, but are closely related to it, and very useful. In principle one could create some sophisticated package which recognised from the juxtapositions of volumes which ones are neighbours, and what order a track passes through them. Such code would be difficult to write and probably would not cope well with all geometries. Especially as some of these concepts are subjective and approximate, and anyway need to be backward compatible with previous Athena software, in particular with the ATLAS identifier scheme. Hence in practice this needs steering by the user.

GeoModelXml generically allows for such schemes. The user can create as many positional indices as desired, with whatever names seem useful. In practice, these should mesh with the detector manager: so for the strip inner tracker, should provide barrel_endcap, layer_wheel, eta_module, phi_module, and side indices so an SCT_ID can be produced.

The user creates a list of indices in a <positionindex> element, using the <addindex> element for each:

```
<positionindex>
  <addindex name="section"/>
  ...
  <addindex name="sensor_phi"/>
</positionindex>
```

The names and values of the indexes are passed to GmxInterfaces in a map, with name as key and the index-value as value.

Each index is associated to a formula in an <index> element. These formulae can appear in logvols, assemblies, and multicopies. They are dynamic: each time a formula is encountered, any previous stored string is over-written. It is changed as soon as the <index> element is encountered. Evaluation of all formulae occurs only and always when an alignable transform or a sensitive logvol is encountered. The resulting values are passed to the user via the addAlignable() or addSensor() methods of the GmxInterface class. The user is thus able to calculate ATLAS Identifiers etc. and store them in a Detector Manager for example.

The formulae are CLHEP evaluator expressions. They can use the defined var, vector, and matrix elements (see <defines>). In addition, and in general much more usefully, they can use the copy-numbers of all their parent Geo(Full)PhysVols. Each time such a physvol is made, it is stored with a copy number (GeoIdentifierTag) in the GeoModel graph. This integer is incremented each time a logvol is copied into a physvol. The user can reset this copy-number to zero whenever appropriate by setting logvolref and assemblyref elements attribute zeroid="true". For example, suppose you have a stave volume used in three successive barrel cylinders. Each time you start a new cylinder, you probably want to add the zeroid="true" to the first stave in the cylinder.

During processing of the <addbranch> element and its daughters, one descends through several levels of physvols. E.g. the strip barrel maybe be the top level (level 0); the cylinders maybe be the next down (level 1); then staves, stave-faces, modules, sensors (level = 5). To access these copy-numbers, the user must create a special vector called CNL ("Copy-Number Level") in a <defines> element and give it a length at least as big as the biggest depth (so 6 elements in the example given). Typically just set each element to 0. There is no choice about the name; it is hard-wired in the code:

```
<defines>
  ...
  <vector name="CNL" value="0 0 0 0 0 0"/>
  ...
</defines>
```

The formulae for position indexes can then use the variables CNL_0, ..., CNL_5 (or more if you make the vector longer). These have dynamic values, i.e. they are an exception to the rule. Internally, they are given the appropriate value at the time of use, by looking at the copy-numbers of the alignable/sensitive volume and those of all its ancestors.

Suppose you want the phi-index of a sensitive-detector element to be equal to its stave copy-number. Then one would use something like:

```
...
<digitizationscheme name="barrelSSDetector"
 params="nRows = 4; nStrips = 1240; ..."/>
...
<positionindex>
  ...
  <addindex name="phi_module"/>
  ...
</positionindex>
...
<logvol name="barrelShortStripSensor" sensitive="barrelSSDetector">
  <index name="phi_module" value="CNL_2"/>
  <!-- CNL_2 is the stave copy number-->
</logvol>
...
```

This example has a very simple formula; in practice sometimes the formula may need to be much more complicated. The formula has to be entered in one line, unlike in C++. Sometimes the conditional operators available in the CLHEP evaluator are useful. If you want different formulae according to whether CNL_0 is 0 or 1, try something like

```
(CNL_0 == 0) * CNL_5 + (CNL_0 == 1) * (64 - CNL_5)
```

to have an index increasing in one case and decreasing in the other.

Other indexes can be used in the formula, by using the positionalindex name. The values are always calculated in the order that the names are given in the position index element. So if you want to use the value of *layer_disk* when calculating the *side* position-index, make sure the <addindex> for *layer_disk* precedes that for *side*.

One can also use dimensions such as a barrel radius to decide a layer index.

Yes, some things are much easier in C++ than xml.

3.14 Sensitive Volumes

Silicon sensors, muon tubes etc. are sensitive to the passage of high energy particles. They need special treatment in Geant4 so that energy deposits ("Geant Hits") can be registered, and then these deposits processed through electronics simulations to be converted into digital output ("digitization"). To allow this, these sensitive volumes are built up into the "Readout Geometry". These things are not handled in GeoModelXml, but they are allowed for with an attribute "sensitive". This can only be given for a logvol (i.e. not for assemblies or multicopies). The value of the attribute is a reference to a <sensortype>.

Each time a logvol is encountered with a 'sensitive' attribute, a GeoFullPhysVol (as opposed to a GeoPhysVol) will be created. First a call-back is made to the user's sensitiveId function which has to return a 32-bit integer. This integer is stored as the GeoIdentifierTag of the GeoFullPhysVol. Then a call-back is made to the addSensor() method of the GmxInterface, with parameters including the digitization-scheme name, a map of positionalindex names and values, and a pointer to the GeoFullPhysVol. The user's addSensor() usually should register the physvol as sensitive in the Detector Manager, storing pointers to it and associating a unique identifier (the upper part of a full ATLAS-identifier) to it.

In detail: for the inner tracker simulation, the sensitiveId should be the Simulation Identifier. These are defined in InDetSimEvent/SiHit.h and SiHitIdHelper.h. Use the position index values and the GetHelper()->buildHitId method. This then becomes the G4 volume copy number via Geo2G4. A (currently private) copy of Geo2G4 is needed if you want to use assemblies; the standard Geo2G4 recalculates copy numbers in the case assemblies are used. G4Hits are processed in the package InnerDetector/InDetG4/SCT_G4_SD. A private copy of this uses the hit-volume's copy number directly, assuming it is the required ATLUAS Simulation Identifier (Not the ATLAS Offline Identifier!).

See the section on position indexes for example code.

3.15 Alignment

The actual ATLAS detector has its sensitive detector elements positioned slightly displaced from the ideal locations, and furthermore these elements move around with thermal expansion etc. To account for these changing positions, the data base stores the actual positions as corrections to the ideal positions. These corrections are often derived by looking at the data in a process known as software alignment. The GeoModel geometry needs to allow for these displacements for analysis of real events.

Essentially in a running detector, each sensitive element is alignable at level-0. Then each support structure (e.g. module, cyclinder, barrel-section) is also made alignable at subsequent levels. This hierarchy allows displacements of large components to be followed rapidly and accurately, while the individual sensitive detectors can be aligned less frequently, once enough statistics have been built up.

Note that simulation is carried out with detectors in their ideal positions. It has proved difficult to apply displacements in Geant4 because they tend to lead to clashes between volumes. So for developing layout designs for future detectors, alignment is not really needed. But for long term use, including once any detector has been built, we need the option to include alignment. Hence alignment is allowed for in GeoModelXml (but not tested, and unlikely to be tested for some time), with a similar procedure involving a call-back routine addAlignable() which is a member of the GmxInterface class. The parameters include the level as described in the previous paragraph, pointers to both the transformation and physvol, and the same map of index-names and values as passed to the addSensor() routines.

The alignment system requires a transformation to place the alignable physvol, even if the transformation is the identity (which will be modified by the actual small offsets). Hence it is convenient to make a transform alignable with an alignable attribute, thus ensuring both elements are present. By default (in the absence of this attribute), the transform will not produce anything alignable. Each time an alignable transform is encountered, the addAlignable() method is called. The alignment level is set to the value of the alignable attribute, which must evaluate to zero or a positive integer.

To be alignable, the transformation must be a GeoAlignableTransform and the physvol must be a GeoFullPhysVol as opposed to the simpler GeoTransform and GeoPhysVol versions that are produced by default. Since these can be defined outside a transform and could (at least in principle) first be encountered in a non-alignable case, they also need to be flagged as alignable. The alignable attribute in this case though only needs to be set to "true", which is sufficient to trigger the creation of the correct type of GeoModel object. Transformations, logvols, assemblies, and multicopies all have an alignable attribute, default value false, which can be set true to make an alignable version.

To summarise: For now, you can ignore alignable. But when the time comes to use alignment, all three of transform, transformation, and object (logvol, assembly or multicopy) need the alignable attribute set. The transform sets the level, the other two are set to true.

4 Guide to setting up a GeoModel detector-tool to use a gmx file

In order to make use of GeoModelXml, you need to make a GeoModel detector tool (i.e. GeoModelXml is not itself an Athena tool, but becomes part of the GeoModel Detector Tool when you use it). The following sections give the current method, with for an example the SCT_SLHC_GeoModel package as the GeoModel Detector Tool. These instructions were valid round about December 2013 and work with Athena version 17.3.4.

4.1 Packages needed

The following packages from the ATLAS SVN repositories are required:

- Standard version InnerDetector/InDetExample/InDetSLHC_Example
- Version 00-00-74 DetectorDescription/GeoModel/GeoModelKernel
- Version 00-05-07 graphics/VP1/VP1Utils
- Version 00-06-12 graphics/VP1/VP1HEPVis
- Version 00-01-05 Simulation/G4Utilities/Geo2G4

Currently (April 2014) the above Geo2G4 version needs a small but important change to give hits correctly; and SCT_G4_SD needs major changes. Ask the author for the fixes. The following packages from the atlas-hessey user repositories are also required:

- GeoModelXml

- SCT_SLHC_GeoModel
- Version 00-03-20 InnerDetector/InDetG4/SCT_G4_SD
- GmxITkStripLoI
- SiHitMapper

The last four are only needed for LoI strip layout, but are a useful starting point for any other layout or detector. The intention is eventually to move these to the standard ATLAS SVN region, simplifying life.

4.2 Helper Scripts and Preparation of Athena

To help users get going, the scripts I use to install the packages are made available at Instructions:

1. Create a new work area (e.g. `mkdir athena/17.3.12; cd athena/17.3.12`).
2. Copy the three files initialise, getPackages, and buildAll from my web pages at [6] into it (e.g. `wget www.nikhef.nl/ r29/upgrade/gmx/initialise`)
3. The file initialise is used to setup your Athena environment. Amend it to suit your local environment. It might need quite some changes depending on your local Athena installation. Then source it (don't execute it).
4. Getting the packages: make the getPackages file user-executable (`chmod u+x getPackages`). Edit it to reflect your directory structure and your lxplus user-name. Use `kinit <lxplus-user-name>` to avoid having to type in your password over and over again. Then execute it. Watch for errors; it should download all needed Athena packages from both the atlas-offline repository (atlasoff) and the atlas-hessey repository. It also downloads some further small files from my web pages at [6] and moves these to the appropriate directories in packages downloaded from the atlasoff repository.
5. Building the packages: make the file buildAll user-executable. Execute it, saving the output into a log file. Check the log-file: each package should build successfully, with some message like "`All OK`".

The initialise file is then sourced each time you start a new terminal in which you want to work on your geometry. But do not run getPackages again: it may overwrite any fixes you made, and you might lose them.

I am interested to hear suggestions for improving these scripts so they work on many platforms.

4.3 Running VP1 to view and debug your geometry

Use the InDetSLHC_Example package to run Athena with VP1. Descend into InnerDetector/InDetExample/InDetSLHC_Example/run. Edit jobOptions_display_SLHC and amend the top-level .gmx file path to suit your environment.

From then on, you can run VP1 with (remembering to "source initialise" if you have a new xterm):

```
athena -l ERROR jobOptions_display_SLHC.py > athena.log
```

If all is well, VP1 should fire up. Select "Geometry Studies" in the main window. Select the "Geo" tab on the left. Check "Other unrecognised volumes" or your detector name. Voila: your geometry is displayed.

4.4 Implementing a GeoModel Detector Factory with GeoModelXml

Included in the above installation is the package SCT_SLHC_GeoModel from my SVN area. This already has everything you need to get running for ITk strip geometries.

However, if you want to make a new subsystem, or modify some other subsystem (pixels, beam-pipe, beam monitor, ...) here is a very brief guide.

You need to build a GeoModel detector tool. Look at my version of the SCT_SLHC_GeoModel package tool. It has a class called SCT_SLHC_DetectorTool in a .h and .cxx file. You need something like this as the starting point for your geometry. You also need the SCT_ComponentFactory, SCT_GeoModelAthenaComps, and components directory stuff (these are all standard things for Athena tools).

Then you need to make a GeoModel detector factory, as described in the GeoModel manual [1]. In this factory, you need the following:

```
...
#include "GeoModelXml/Gmx2Geo.h"
#include "GeoModelXml/GmxInterface.h"
...
// In the create method of your Detector Factory, add the following
// at the appropriate place
void XXX_DetectorFactory::create(GeoPhysVol *world){
  string filename("myGeometry.gmx") // Change name to your top-level .gmx
                                    // file, or of course better still
                                    // pass it as a python parameter

  ... //  All sorts as required in the GeoModel manual

  GmxInterface gmxInterface;
  Gmx2Geo gmx2Geo(filename, world, gmxInterface);

  ... // More stuff as required in the GeoModel manual

//
//   Add the tree-top to the detector manager. This also makes it
//   appear as SCT in VP1. It is probably the last (most recently
//   added) thing in the world PV so loop from the back looking for
//   our subdetector name.
//
    unsigned int nChildren = world->getNChildVols();
```

```
for (int iChild = nChildren - 1; iChild; --iChild) {
    if (world->getNameOfChildVol(iChild) == "SCT") {
        // The * converts from a ConstPVLink to a reference to
        // a GeoVPhysVol; the & takes its address.
        m_detectorManager->addTreeTop(&*world->getChildVol(iChild));
        break;
    }
}
```

This will create an instance of a Gmx2Geo object which will process the file (in the constructor), and add volumes to "world" as instructed. It also adds your detector to the detector manager. There are some naming conventions to follow, which allow VP1 to pick-up your detector.

4.4.1 GmxInterface

The GmxInterface class is a base class implementing four call-back routines. It interfaces the GeoModelXml package to your detector factory tool. The methods are addSensorType, sensorId, addSensor, and addAlignable. The first is called each time a <digitizationscheme> element is encountered. The second two are called each time a <logvol> with its 'sensitive' attribute set is encountered. The last is called each time a <transform> with alignable attribute is encountered. The parameter list includes pointers to GeoFullPhysVols, GeoAlignableTransforms, and a map of positional index names and their values. The intention is that the user over-rides the default methods with methods using the detector manager to build up the readout geometry and alignment schemes. The default methods simply print out debugging information.

Probably it is best to first set up a geometry without any alignable or sensitive parameters, and develop it using VP1. Then deal with sensitive volumes. Then finally deal with alignable volumes. This can be done using the default GmxInterface methods to check the input; then finally switch to your over-ridden versions of these methods.

References

[1] Joe Boudreau, GeoModel Manual, `http://atlas.web.cern.ch/Atlas/GROUPS/DATABASE/detector_description/Geometry%20Kernel%20Classes.doc`

[2] AGDD web pages `Cannotfindthem`

[3] Xerces-C web pages `https://xerces.apache.org/xerces-c/`

[4] CLHEP web pages `http://wwwasd.web.cern.ch/wwwasd/lhc++/clhep/index.html#docu`

[5] Virtual Point-1 display package web pages `http://atlas-vp1.web.cern.ch/atlas-vp1/index.php`

[6] `http://www.nikhef.nl/~r29/upgrade/gmx`