Implementing Strip Layouts with GeoModelXml in Athena			
ATLAS Project Doc. No: **None**	*Institute Doc. No:* **None**	*Created:* **22/5/2014** *Modified:* **22/5/2014**	*Page:* **1 of 19** *Rev:* **B**

Note

Implementing Strip Layouts with GeoModelXml in Athena

This note summarises the steps needed to develop a geometry for the Strip sub-system (also known as SCT) of the ATLAS-Upgrade Inner-Tracker (ITk). The GeoModelXml package can in principle implement any ATLAS system or sub-system. Its documentation is therefore generic. This manual is specific to the strip sub-system of the ITk. It is aimed at bringing new-comers quickly up to speed, and to remind more experienced developers of certain details.

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

Distribution List:
Public

History of Changes

Rev. No.	Date	Pages	Made by	Description of changes
A	22/5/2014	All	N.P. Hessey	Original document
B	1/12/2016	13-16	N.P. Hessey	Updated athena run sequence

Contents

1 Introduction

First, someone or a group has to decide on a new layout to try. They work out precise dimensions and positions of all silicon elements. This is not covered here. Next the geometry has to be implemented in Athena to enable simulation and tracking. This note gives pointers on how to do that, and the steps needed to efficiently reach a good implementation.

The main part of the note describes how to build up the layout in an xml file for use with GeoModelXml [1]; then pointers are given on a sequence of Athena runs to test and debug the layout. Finally there is a very brief overview of several packages that are important or have been modified for the strip geometry.

2 Building up the gmx files

2.1 Splitting the input into separate files

The gmx input can all be in one file, or can be split over as many as are useful. A single file rapidly becomes unmanageable, and you are strongly recommended to split the input over several files. Consider the following when deciding what files to have:

- Re-usability: If a file is useful in different projects, make it a separate file. The most obvious example is the material file. Also files of dimensions can be useful for different projects, even if they need some modification.
- Multiple developers: If a team is developing a layout, e.g. a barrel specialist and an endcap specialist, then make separate barrel and endcap files.
- File size: Just keeping files to a manageble size, and collecting things together that are closely related, helps maintenance.

As an example, table 1 lists the files used to simulate the LoI layout [2].

2.2 Materials

There is a file Materials.gmx [3] which defines most commonly used chemical elements and materials. Try to find something suitable in there before creating a new material. It has all elements with atomic number from 1 to 32 and several of the remaining higher ones. For materials, it has most commonly used metals, some alloys, everything from the pocket Particle Data Group booklet, and a few more.

Conventions used: Elements have their (British spelling) full name. Metals have the element symbol followed by “Metal”, e.g. AlMetal for Aluminium.

2.3 Dimensions

Always use a variable name for a dimension. Collect them together near the start of the gmx.

File	Included by	Content
ITkStrip.gmx	Entry-level	All filenames; <positionindex>; envelopes; <addbranch>
Materials.gmx	ITkStrip.gmx	General-purpose material definitions
CommonDefines.gmx	ITkStrip.gmx	Dimensions that may be needed in barrel and endcap etc.
SensorDefines.gmx	ITkStrip.gmx	Sensor dimensions
Sensors.gmx	ITkStrip.gmx	Sensor geometries
ITk.gmx	ITkStrip.gmx	ITk-general items such as Outer Support Cylinder, polymoderator
Barrel.gmx	ITkStrip.gmx	Barrel geometry
BarrelDefines.gmx	Barrel.gmx	Dimensions specific to barrel
Endcaps.gmx	ITkStrip.gmx	Endcap geometry
EndcapDefines.gmx	Endcaps.gmx	Dimensions specific to Endcaps
Services.gmx	ITkStrip.gmx	Service volume geometry

Table 1: Gmx file-structure for the LoI layout. The files are collated in the order given.

2.4 Building a solid geometry from scratch

Start small. For example, first make one example of each sensor type and view in VP1. Then collect them together in groups, that make one side of a local support (“staveface”, “petalface”). View these before proceeding. Build them into staves and petals, then cylinders and wheels, and so on, viewing at each stage.

Write transformations in a way that is understandable: don’t combine consecutive transformations into a single one – that tends to obscure the purpose. And use $2\pi/n$ rather than using a calculator to work out the angle yourself. Use the name of the transformation to describe what it does.

2.4.1 Sensors

Barrel sensors are straightforward. Endcap sensors aren’t. I have a program “petalWafers” [5] that calculates optimised shapes for the sensors, and prints out the numbers in a format which can be included in a gmx file.

In the initial stages, don’t worry about position indexes or making them sensitive; that complicates life. Make the solid geometry correct first, then add sensitive detectors.

2.4.2 Assemblies of sensors

Use <multicopy> to place barrel sensors on a stave face. Usually you need two types of face: one with all strips parallel to the stave axis (“axial”), and one with them rotated by the stereo angle (“u” or “v”). At the moment, only one version (u) is needed. Then place an axial and a stereo face into a stave, adding a stave core. Note that an axial-u stave when turned over becomes a v-axial stave (ignoring some important details in real life, such as brackets which spoil this symmetry). This

reduces the number of petal-faces and stave-types needed (but complicates some other things like positional indexes).

For petals, place the sensors individually since very few items are repeated.

2.5 Avoiding touching volumes

Geant4 frequently recalculates the current containing-volume of particles. Rounding errors can lead to a wrong result when the particle is very close to the edge of a volume that touches another volume in the same mother volume. Such particles tend to get lost, invalidating the whole event. To avoid this, add space between any sibblings that in reality touch each other. The space needed by G4 is only 0.1 μm. But this is typically lost in the ascii to internal conversion of numbers in the xml. I find 2 μm sufficient; defined as EPS in CommonDefines.gmx. Shrink one or other of two touching volumes by this amount to avoid clashes.

2.6 Quick check of xml grammar and DTD compliance

VP1 takes quite a while to fire up in Athena, and Athena gives a lot of output burying any hints at where problems might originate.

One can copy the geomodel.dtd file to the directory with your .gmx files and run

```
xmllint --noent --valid --noout ITkStrip.gmx
```

to check the .gmx files are well-formed and valid for the given DTD.

2.7 VP1: Viewing the geometry

Section 6.1 summarises how to use VP1 [6] to check the solid geometry. Once your gmx code parses correctly, you can view it in VP1. Build up from small to big, checking as you go, rather than try to do everything at once.

To help pin-down problems in a large file, it can be useful to by-pass objects. Remember, only the things in the <addbranch> element are included; so you probably do not have to comment very much out. Find the highest level (typically the <addbranch> but could also be an <assembly> or <logvol>) in which you can comment something out without also removing what you want to debug.

Also check for volume clashes: see section 6.2.

3 Sensitive detectors

Once you are satisfied with the solid geometry, start making things sensitive. There are several stages to this.

3.1 SCT_GmxInterface

GeoModelXml calls several routines that the user should supply. These routines should be members of a class inheriting from the GmxInterface class. For the strips, the new class is called SCT_GmxInterface and is declared and defined in the SCT_SLHC_GeoModel package. There are four call-back routines, detailed in the GeoModelXml manual [1].

3.1.1 Detector Types

The gmx files define <readoutgeometry> elements. These contain <sensorclass> elements which are intended to map to a class inheriting from the SiDetectorDesign class in the InDetReadoutGeometry package. In turn, a <sensorclass> contains <sensortype> elements which are specific versions of such a <sensorclass>. For example, barrel short-strip, medium-strip, and long-strip sensors can be three distinct sensortypes differing in the number of rows of strips (4, 2, 1); they are all cuboid (box) shaped, so are contained in the SiStripBox <sensorclass>.

All these three elements can have <parameter> elements with a name and a value. The parameters are valid within the containing element and all its children. The user is free to add any uniquely named parameters, and is responsible for processing these in the SCT_GmxInterface::addSensorType() member. This is called each time a <sensortype> element is encountered, with the sensorclass-name, sensortype-name, and a map of all parameter name-value pairs valid at that time.

The <readoutgeometry> elements and all their contents are processed early on in the gmx processing, so they are all available before any sensitive logvols are encountered.

3.1.2 SensorId

At several points in Athena, sensitive volumes need identification. For example when dealing with G4 hits, digitization, and data analysis. Two schemes are in operation for historical reasons. Simulation Identifiers [11] are used for G4Hit processing. These identify only a volume, not the individual strips within a volume. (Later, Offline Identifiers are used with a helper-class declared in SCT_ID.h [12]). When a sensitive logvol is encountered, the GmxInterface::sensorId() method is called. This should return the Simulation Identifier for this volume. It has to be calculated from the position indexes that are passed as a map to the routine.

3.1.3 AddSensor

Each time a <logvol> is encountered with attribute `sensitive` given and set to a sensortype-name, the GmxInterface::addSensor() method is called. Here the user can add the sensitive element to the Detector Manager with a pointer to the SiDetectorDesign created for the sensortype.

3.2 Position Indexes

The are two main uses of position indexes. One is for the calculation of the Simulation Identifier [11] of a sensitive element, to be used in processing G4Hits; the other is for the calculation of

Field Name	LoI values	Describes
system	2	Inner Detector; assumed in SCT_ID.h; no need to set in gmx
part	2	SCT; no need to set in gmx
barrel_endcap	-2, 0, 2	Whether in Endcap-C (-2), Endcap-A (2), or Barrel (0)
layer_disc	Barrel: 0 - 5 Endcap: 0 - 6	Which cylinder; stub is 4 Which disk
eta_module	Barrel: -13 to -1; +1 to +13 Endcap: 0 - 5	Which sensor along a stave Which ring of sensors
phi_module	Barrel: 0 up to 71 (depending on which layer) Endcap: 0 to 31 or 63, depending on ring	Stave number within a cylinder Inner rings give petal number
side	0, 1	0 is side nearer the IP
strip	0 - big	strips within a row of a sensor; don't set in gmx
row	0 - 7, depending on sensor	Which row of strips within a sensor; don't set in gmx

Table 2: List of position index names, ranges for the LoI layout, and description. The first two and last two are not needed in the gmx files.

Offline Identifiers [12], used in the DetectorManager to identify sensors. The strips use the position indexes given in table 2 to build up both of these.

The values used must be allowed by the Offline Identifier scheme. The allowed identifiers must be set in an xml file (section 3.4) or you will get a lot of error messages. If your xml file is correct, and you still get error messages, then your position indexes in the gmx are probably wrong.

To make the correct formulae in the <index> elements, first write out the hierarchy of your layout. E.g. maybe your top assembly is the "tracker" containing a barrel and two endcaps; barrel contains cylinders; cylinders contain pairs of staves; staves contain petal faces; which contain sensors. Write this and the endcap equivalent out. At each level, write down the CNL_*n* variable relevant to it (CNL is "Copy Number Level", so CNL_2 is "copy number level 2").

Figure out how your desired position indexes relate to these CNL's. If you cannot do it with only CNLs, maybe you need to use other position indexes; these are available. But note they are evaluated in the order given in the <positionindex> element. Avoid using one before it is evaluated!

Then figure out at what position you need to evaluate the position index; and add an <index> element there, implementing the formula you thought you need. Check using the next section, and re-evaluate where you need to put the formula, and what it should be, until the Simulation Identifiers are correct.

3.3 sihit: debugging identifiers with VP1

Getting the position indexes, and hence the Simulation Identifiers correct needs some care. To help debug things, you can use VP1. Once you have the sensitive detectors visible, open the Interactions tab in the Geo tab. Select CopyNo. Click on a detector volume; the message area at the bottom gives you the CopyNo. This should be the Simulation Identifier. You can check if it is correct with a small routine called sihit, available from the author. Figure 1 is a screen shot of this process, with a selected sensor highlighted with red; the CopyNo has been cut and pasted into a terminal on the right as a parameter to the sihit program. Sihit then decodes it into the position-indexes; you can see if all is as you expect.

3.4 ATLAS Offline Identifier xml file

The ATLAS Offline Identifier scheme identifies each and every electronic channel in ATLAS uniquely. For the strips, an electronics channel corresponds to one preamplifier channel in an ABC chip. For the SCT, typically two strips joined end to end feed one such preamp. For the ITk strip detector, one strip feeds one preamp, so channels can also be seen as representing strips.

Whenever you change the arrangement of sensors (e.g. number of discs, number of staves in a barrel) the allowed Offline Identifiers will change. The set of allowed identifiers is built up in an xml file in the InDetSLHC_Example package [9]. Many, many versions are stored in the data directory of that package. It is definitely a useful check to be told early on that your position indexes from the gmx files lead to a disallowed Offline Identifier. So it is worth making an xml file with only the allowed Offline Identifiers. For some reason, most xml files contain a section defining more or less every identifier you could ever want (in a section called dummy region); avoid that.

Some background on identifiers: An Offline Identifier for the strips represents a set of integers, one

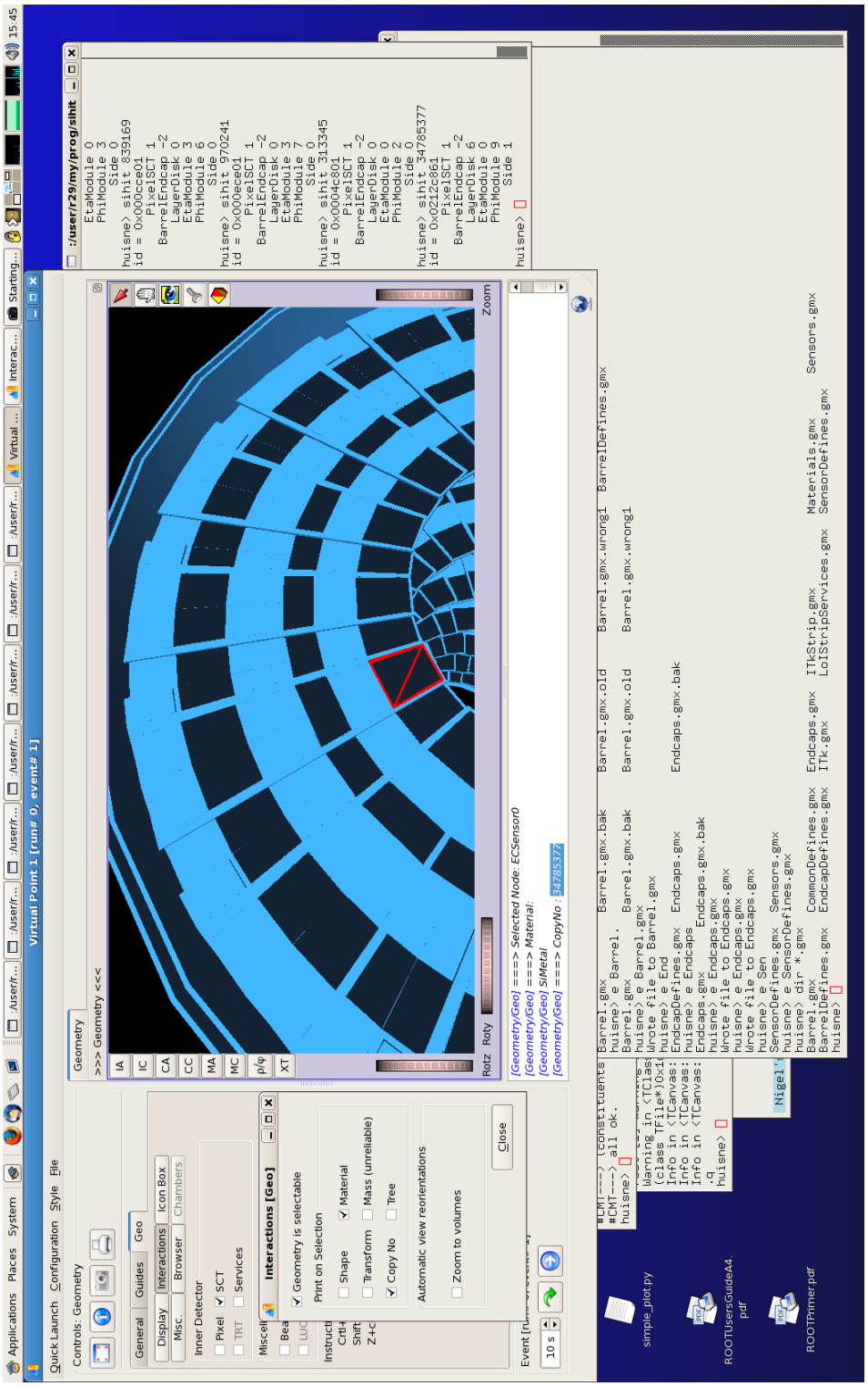

Figure 1: Screenshot illustrating debugging of position indexes via the copy number of sensitive volumes.

for each of the names listed in table 2. In the xml, these are called fields. There are three forms of representation:

- Expanded-Id: A vector of the integer-values of the fields
- Compact-Id: A single 64-bit integer, with each field encoded in a bit-range.
- Hash-Id: a partial representation, which represents an entire strip sensor. These ignore the system, part, row and strip fields. They are 32 bit integers from 0 up to the number-of-wafers-less-one. The hashing algorithm from which they are derived makes it easy (constant time) to go from hash-id to compact-id. But vice-versa involves a search and takes logarithmic time.

A field also has some concept of neighbours: the objects with a given integer value are assumed to be adjacent to objects with the next value allowed.

There are many elements allowed in the xml, and I have not understood what most of them are for. However a small subset is sufficient to define the allowed identifiers. The following gives a simpleton's guide to making an xml file. Read it in conjunction with an existing xml file and it should make some sense.

Allowed values of a field are set with a <range> element. Attribute `name` is set to the field name. For a run of consecutive allowed values, use the `minvalue` and `maxvalue` attributes. If you only want a single number, use the `value` attribute. For a list of non-consecutive numbers, use the `values` attribute. Where things wrap around, like staves in a barrel, you can explicitly state that the first stave is adjacent to the last with the `wraparound` attribute. Where a number is skipped, e.g. on staves where there is no eta_module 0, you can indicate that eta_module -1 is adjacent to eta_module +1 with the `next_value` and `prev_value` attributes.

A set of allowed identifiers is defined in a <region> element. These elements contain a complete list of <range> elements — i.e. one <range> element for each field of the identifiers. Several <region> elements are used to define the complete set of allowed identifiers. They must all have the <range> elements in the same order of field names. A <region> element adds all identifiers to the allowed list that can be made from the ranges contained: it is the Cartesian product set of the ranges given, i.e combine each member of the first field in turn with each in the second; combine each such pair with each of the third field; and so on. In general you need to give several <region> elements: otherwise either the Cartesian product will contain identifiers that do not represent any existing strip, or leave some strips without a valid identifier.

I think one could build up all needed strip identifier sets with just <range> and <region> elements. However, it can become very repetitive and hard to maintain. There are some more elements that are useful:

Values can be given a name; it is much clearer to use "negative_endcap" in place of -2. This is done with <label> elements.

If you are repeating a series of <range> elements in your <region> elements, you can put these in a <subregion> element. As the name suggests, this is less than a complete <region>. It can be used in a <region> element by use of a <reference> element. <subregion> elements can contain <reference> elements too.

4 Services

Cables, fibres, cooling pipes, and their supports are collectively known as services. They have a significant radiation length, and so must be modelled to get meaningful performance predictions. On the other hand, they are very numerous and implementing them individually is too much work. Instead they are modelled as material smeared over simple volumes when developing layouts (later much more detail will be needed). Various methods of calculating how much material to put in, and what its elemental mixture should be, exist. These include spreadsheets, a package InDetServMatGeoModel [15], and an xml based program ServMat which produces service volumes in gmx format.

Spreadsheets are hard to maintain, and need quite some effort to produce output in the form needed for Athena geometries. InDetServMatGeoModel works well, but does not go well with GeoModelXml (need to repeat and maintain dimensions in different places); it is difficult to find what is being modelled; and is undocumented.

ServMat is designed to work with GeoModelXml. It reads materials and dimensions in the same format as GeoModelXml, so you can use exactly the gmx files (or subsections of them) that you have for your layout. And it writes out materials, shapes and logvols in gmx format, for easy inclusion in your gmx. It could be fully automated with python scripts, but that is not done yet. The main documentation is in the DTD file and a note for a predecessor program, ServX0, which should at some stage be converted to a ServMat manual.

ServMat's main input is a collection of cross-sections of cables, cooling tubes, fibres etc. as (often concentric) tubes. The tubes have a material filling them. The number of tubes in a given volume is specified by the number of staves or petals whose services pass through that volume, and the number of such tubes that feed each stave or petal. Service volume shapes and positions are specified separately, and can be defined in terms of dimensions used in your gmx layout. ServMat calculates the weight of each element that is present in each service volume, creates a material with this elemental mix, and calculates a density for it such that when the service volume is filled with it, it has the correct mass. In this way, you get simultaneously the correct radiation and interaction lengths (on average). It also outputs gmx to create the service volume shapes and logvols, and transforms to position these correctly. The output can be incorporated in your gmx rather straightforwardly.

5 ATLAS Geometry Text-override file

The entire ATLAS geometry is stored in a data base. It is cumbersome to alter that (particularly adding and deleting parameters) and not everyone should be allowed to play with its contents. The text override-file is a way of developing layouts avoiding these problems. Essentially a layout is downloaded from the database at the start of Athena, but then some values are overwritten by parameters specified in this file. So, if you don't need to change anything with respect to the database, this file can be empty. Once a layout is ready for production use, it is migrated to the database and the file is obsolete. In principle, it need only contain lines with the things you need to change. In practice, it is useful to include the whole thing as a guide to parameter names that can be altered and to make it easy to check exactly what layout is being implemented.

This file is no longer needed for the strips when they are implemented with GeoModelXml. How-

ever, it also implements the pixels, beam-pipe and pixel services, and is used for those. It also has parameters for implementing the strip services, but these are ignored by SCT_SLHC_GeoModel should be stopped from doing so by setting a switch in it. Unfortunately, the polymoderator and outer support cylinder seem to be considered part of the pixel, and so are implemented from the text file (currently). Make sure your volumes do not clash with these.

There is a convention for the name of this file; see section 6. This is something of a nuisance: each time you make a new layout with different xml file, you have to also make a new copy of the text override file, even though its contents don't change (since we are not using it for the strips).

6 Athena run sequence

There are several stages in developing and testing a layout before releasing it for production and using large resources.

These are:

1. Prepare the GeoModelXml input files
2. Athena display with VP1: View the results
3. Athena/G4VolumeDebugger: Check layout for overlaps
4. Athena/G4 simulation: run single muons
5. Athena/SiHitAnalysis + ROOT: Check the G4 hit-maps
6. Athena/Digitization: Digitize the hits to make RDOs (Raw Data Objects)
7. Athena/SiDigiAnalysis + ROOT: Check the RDOs
8. Prepare a tracking geometry
9. Check tracking

The package InDetSLHC_Example [9] has example job-options files (python scripts) in the share directory. For each one you need to use, copy it to your run directory, and edit to reflect your needs. Since the set-up changes rapidly, instructions for this are found on the web and by word-of-mouth.

Currently the gmx files are treated differently. Give the pathname inside the requirements file in GmxLayouts/cmt.

Note that you can find out what parameters can be set in a python script by looking in the genConf directory of an installed package. The .py files in there contain the names that can be set. For example, after building SCT_SLHC_GeoModel, look in SCT_SLHC_GeoModel/genConf/SCT_-SLHC_GeoModel/SCT_SLHC_GeoModelConf.py. It lists 18 parameters that can be set, including the GmxFilename, in the SCT_SLHC_DetectorTool class.

6.1 VP1 Display

GeoModelXml produces a geometry in the GeoModel tree. The VP1 viewer [6] views this directly, and is a good debugging tool. Rotating objects 90 degrees the wrong way, easily done in new layouts, shows up instantly. However details are extremely hard to check, not helped by the fact that everything in the SCT has to have the same colour. You can set colours in a file called VisAttributes.cxx in the package VP1GeometrySystems. I can send you a good starting place for this file.

To run VP1, go to the InDetExample run directory. (InnerDetector/InDetExample/InDetSLHC_Example/run). Copy the files joboptions_display_GMX.py, geometry.py, ID_only.py from the ../share directory.

Run Athena with this job options file. Once VP1 appears:

- In the big window, click on Geometry Studies
- On the left panel, click the Geo tab
- Check the SCT box (Also the pixel and/or beampipe boxes if you want to see those parts too)
- You can move the view, zoom in etc. What you see is the Inner Tracker Envelope: currently a big boring cylinder. To look inside, Select the red arrow tool at the top right, and "Expand to child volumes": the panel on the left reminds you how to do this (Ctrl+Click). To go back, use Shift+click. To return to moving around the geometry, click the hand sign at the top-right.
- You can also make things disappear (Z+click); the disappeared can be brought back by double clicking on it in the Icon Box. You reach that by clicking on the Icon Box tab in the Geo tab.
- Assemblies: if you Ctrl+click enough, you typically end up with nothing to see, assuming your geometry uses Assemblies. Since Assemblies have no real shape, VP1 by default does not display them, nor their contents. Since there is nothing to click on, another mechanism is needed to make them appear. Select the Misc tab under Geo. Type in AssemblyLV (case sensitive!) in the “Vols. named” box (this is the name of all Assembly volumes in GeoModelXml). Then click on the Expand button next to it. Maybe you see some 1 mm cubes appear; these are Assembly contents of the assembly you just expanded. Keep clicking on the Expand button, and more and more detail appears. Be patient, especially if working remotely; it can take some time. I have not found a way to reverse this process, except to go right back to the start by pressing the Reset button in the same tab. Better solutions welcome.
- You can make pictures for presentations: The camera at the top left of the window should work, but hasn’t worked for me for many years now Instead, I go to moving around mode (hand sign at top right), then right click the background, scroll down to More... and find the eps option.
- Cut-away views: Select the Display tab under the Geo tab. Click on the red dots at the top left to open up views through a phi-sector.
- You can also change image quality in the Display tab: Show outlines, change line thickness, make curves more realistic.

- You can also change image quality by right-clicking on the picture. Anti-aliasing helps. In general, better quality pictures take longer to render.
- Saving settings: If you find yourself frequently going to the same view, with many clicks and manipulations, you can make a short cut to this. Under the Configuration tab, you can "Save current tab configuration". Then next time you start VP1 up, you can return to this view with the "Load tab configuration from file" button under the Configuration button.

6.2 Geomtest: Volume clash check

It is easy to create volume overlaps, or volumes sticking out of their mother volume. And it is not necessarily easy to see these in VP1. There is a tool to help you find such errors, which is well worth using – it will save time in the long run. The package is called /Simulation/G4Utilities/G4DebuggingTools/. Currently I need a private version and edit the default values directly in the header file src/VolumeDebugger.h and re-make it. The values you need are in the job options file, commented out. The job options file is called joboptions_geomtest_GMX.py. Run athena with it. Divert the output to a log file. When it has finished running, look for any error messages on stderr; any clashes found will be notified here, with more details in the log file (search for G4Exception).

6.3 Simulation: Single muon events

Next step is to generate some simple events, and see if we get G4 hits, and if they get the correct Simulation Identifiers. Amend the jobOptions_simulation_GMX.py file for your geometry files; also set the number of events (start with a small number like 10 and build up; 10,000 is good for most things and an overnight run of a million or so gives pretty pictures and a more stringent test.). And set the output file name athenaCommonFlags.PoolHitsOutput. Run athena with this job options file.

Look through the output for error messages and fix any issues. In particular, initially set the logging level to DEBUG. Then each hit will give a line of information about the position indexes and volume copy number; this can be very useful for debugging the position indexes. The package doing the work is SCT_G4_SD [16]. It has been modified to use the G4 copy number directly as the Simulation Identifier.

This stage creates an ATLAS persistant version of G4Hits in a ROOT file. That is difficult to view directly; instead, run the next stage which creates plots that help debug much better. The hits are stored in local hit coordinates, using the G4 transformations to go from the G4Hit position which is in global coordinates. The hits are stored in a condensed way, with all G4Hits in a contiguous region converted to one ATLAS hit, to save storage and processing time.

6.4 SiHitAnalysis: Drawing G4 hit positions

The ROOT file produced in the simulation stage above can be analysed by running athena with the jobOptions_SiHitAnalysis.py file. Check-out the package SiHitAnalysis. This processes the hit file and recalculates the hit positions in global coordinates, using the GeoModel transformations. The link to the transformation is made via the Simulation Identifier. This makes a very good check of

position indexes: wrong indexes or wrong transformations, give hits in the wrong place. Make very sure all your filenames and geometry versions are correct; you can have these quite wrong and still get plots, but with everything in the wrong place.

The transformed hits are made in another ROOT file, set in the job options. This can be processed into pictures using ROOT. A file stripHit.C, available in SiHitAnalysis, can be loaded into ROOT and will make several figures.

6.5 Digitisation

Once hits are looking good, you can digitize them into Raw Data Objects, RDOs. Use jobOptions_digitization_GMX.py.

6.6 SiDigiAnalysis: Checking RDOs

Check-out the package SiDigiAnalysis and use jobOptions_SiDigiAnalysis.py to produce a ROOT file of RDOs in a convenient form. Run ROOT and load stripDig.C to produce plots. Chenck the plots carefully...obviously, all RDO's should be where you have detectors, and all detectors should be seen.

6.7 Tracking Geometry

Ask an expert.

6.8 Tracking

Ask an expert.

6.9 Performance

There are packages ready made to analyse xAODs etc; no need to invent your own.

7 Packages used and how things work

A very brief summary of which packages are important for GeoModelXml implementation of the strips, especially those that have been modified for GeoModelXml.

7.1 GeoModelXml

A package intended to become part of GeoModel for building up the GeoModel tree under guidance from an xml file [1].

7.2 SCT_SLHC_GeoModel

This [8] is the main package that implements the strip geometry using GeoModelXml. (It could also form the basis for other parts of ATLAS to be implemented in GeoModelXml).

It is almost totally redone compared to the standard package. It is very much shorter and simpler than the standard package.

It also sets up the Detector Manager with sensitive detector lists, alignable item lists, some Athena tools, and some Offline Identifier things.

7.3 Geo2G4

A package to convert the GeoModel tree into a G4 geometry [7]. Slightly modified to use directly the GeoModel integer copy number directly as the G4 copy number, even for assemblies.

7.4 InDetSimEvent: Simulation Identifiers

This package [11] defines the Simulation Identifiers and provides methods to create them. SCT_-GmxInterface in SCT_SLHC_GeoModel uses this package to calculate copy numbers for sensitive volumes for GeoModelXml to add to the sensors.

7.5 SCT_ID.h: Offline Identifiers

This file [12] is an Offline Identifier helper-class for the strips. It uses the xml identifier definition file to define the allowed identifiers. It provides methods to create identifiers, to transform between representations, and to extract information from identifiers. Upgrade sensors have several rows of strips per sensor. This required a modification to SCT_ID.h to add the row field; this has been done in a backward compatible way so that all SCT software should still work.

7.6 SCT_G4_SD: G4 Hits

The package SCT_G4 [16] processes all strip hits from G4. It uses the G4 transformations to transform hits from the global coordinate system to the sensor local coordinate system. The hits are then compacted and written out as a pool file. GeoModelXml uses a modified (simplified and faster) version. The essential change is to use the G4 copy number directly as the Simulation Identifier. SCT_SLHC_GeoModel tells GeoModelXml to give sensitive detectors a copy number equal to the Simulation Identifier. Geo2G4 copies this to the G4 copy number. A modified version of Geo2G4 makes sure that this works also for assemblies, including assemblies of assemblies etc.

7.7 G4AtlasApps: Sensitive volume names in atlas_idet.py

This file [13] defines a set of names that will become sensitive volumes. Names of sensitive logvols in GeoModelXml must be the same as the names given here, or G4 simulation will not treat them as

sensitive and so not produce any hits in them. It is not sufficient just to use the `sensitive` attribute in the gmx.

8 Concluding Remarks

Please let me know of errors and improvements I could make.

References

[1] GeoModelXml package:
`https://svnweb.cern.ch/cern/wsvn/atlas-hessey/GeoModelXml/`

[2] GmxITkStripLoI package with .gmx files:
`https://svnweb.cern.ch/cern/wsvn/atlas-hessey/GmxITkStripLoI`

[3] Materials.gmx file: `https://svnweb.cern.ch/cern/wsvn/atlas-hessey/GmxITkStripLoI/trunk/data/Materials.gmx`

[4] gmx2geo package, a quick parser for .gmx files:
`https://svnweb.cern.ch/cern/wsvn/atlas-hessey/gmx2geo/trunk/`

[5] petalWafers: program to calculate strip-endcap sensor dimensions. Ask Nigel.

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

[7] GeoModel geometry to G4 geometry converter:
`https://svnweb.cern.ch/cern/wsvn/atlasoff/Simulation/G4Utilities/Geo2G4`

[8] SCT_SLHC_GeoModel package, which implements the strip geometry:
`https://svnweb.cern.ch/cern/wsvn/atlas-hessey/SCT_SLHC_GeoModel`

[9] InDetSLHC_Example package with several python scripts plus text files and identifier .xml files for running Athena: `https://svnweb.cern.ch/cern/wsvn/atlasoff/InnerDetector/InDetExample/InDetSLHC_Example/trunk`

[10] SiHitMapper package to histogram G4Hit results: `https://svnweb.cern.ch/cern/wsvn/atlas-usoldevi/usoldevi/Nsensor/17.3.12/SiHitMapper/run/`

[11] InDetSimEvent package to create and handle Simulation Identifiers: `https://svnweb.cern.ch/cern/wsvn/atlasoff/InnerDetector/InDetSimEvent/trunk/`

[12] SCT_ID package: ATLAS offline identifier helper package for the SCT:
`https://svnweb.cern.ch/cern/wsvn/atlasoff/InnerDetector/InDetDetDescr/InDetIdentifier/trunk/InDetIdentifier`

[13] Python script where names of sensitive volumes are picked up:
`https://svnweb.cern.ch/cern/wsvn/atlasoff/Simulation/G4Atlas/G4AtlasApps/trunk/python/atlas_idet.py`

[14] Helper scripts to get GeoModelXml running for strip geometries: `http://www.nikhef.nl/~r29/upgrade/gmx`

[15] InDetServMatGeoModel, a package to automate service volumes: `https://svnweb.cern.ch/cern/wsvn/atlasoff/InnerDetector/InDetDetDescr/InDetServMatGeoModel`

[16] Package SCT_G4_SD, which handles G4Hits in strip detectors: `https://svnweb.cern.ch/cern/wsvn/atlas-hessey/SCT_G4_SD`