LArHits 0.5 Documentation

LArHits is an implementation of the ATLAS liquid−argon calorimeter in Geant4. It is a work in progress; this is the documentation for version 0.5 (09−Jul−2001), available at <ftp://ftp.nevis.columbia.edu/pub/atlas/seligman/LArHits−0.5.0.tar.gz>.

This note focuses on the LArHits version developed by William Seligman (Nevis Labs) and tested until now with the tailored versions of the EM barrel and endcap geometry descriptions.

In its current form, LArHits illustrates three concepts:

- implementing the LAr calorimeter geometry efficiently within the Geant4 simulation;
- recording hits in the LAr EM cells using the Geant4 hit mechanism;
- saving simulated events to disk using an efficient ROOT−based persistency scheme.

The routines used to implement these concepts are described below. This document does not attempt to describe every coding trick used within the routines; the programs' comments already do this. The documentation for Geant4 and ROOT can be found at <http://wwwinfo.cern.ch/asd/geant4/G4UsersDocuments/Overview/html/index.html> and <http://root.cern.ch/root/RootDoc.html>, respectively.

LAr EM Calorimeter Geometry

Implementation notes

`LArDetectorConstruction` is the main routine for implementing the LAr EM geometry. This routine creates an "experimental hall" volume, then calls other methods to place their detectors within the volume. In LArHits 0.4 and higher, there are two such methods: `LArBarrelConstruction` and `LArEndcapConstruction`.

LAr Barrel Geometry

The geometry of the electrodes and absorbers in the LAr barrel is quite complex. While it is possible to specify the geometry of the detector using the volume primitives of Geant4, the resulting memory use is high (see <http://chep2000.pd.infn.it/abs/abs_a354.htm>). It turns out to be more efficient to create a custom"accordion−shaped volume" in a routine that inherits from `G4VSolid`.

This custom solid is implemented in routine `LArBarrelSolid` (in an earlier version, the name of this routine created by Stefan Simion was `G4Accordion`).

During the development of the Geant4 hit code (discussed below), it was observed that calculations contained in `G4Accordion` were also useful in determining the barrel cell in which energy was deposited. Therefore, the common routines required for both geometric projection and cell identification were put into a new class, `LArBarrelCalculator`.

`LArBarrelCalculator` uses the singleton pattern, that is, there can be only one object of this class. The class is used by passing a `G4ThreeVector` "point" to the

calculation methods `WhereIsHit` or `FindCell`. The user then "reads the results" of the calculator with accessor methods (also see "Calculator interface" below).

LAr Endcap Geometry

The development of the LAr endcap routines paralleled that of the LAr barrel. The initial geometric specification of the endcap electrodes and absorbers was developed separately by Andrei Soukharev as `LArWheelSolid` <http://cern.ch/asoukhar/LArWheelSolid–0.1.0.tar.gz. A separate calculator class was created as `LArWheelCalculator` (also see "Calculator interface" below).

In the development of `LArWheelCalculator` some common functionality with `LArBarrelCalculator` was identified. This functionality was "factored out" of the two calculator classes and put into a new class, `LArRootDetDescr`. As the name implies, this class is ROOT–compatible and some of its functions (e.g., calculating the eta and phi of a given cell) can be used in data analysis using ROOT.

The endcap geometry has a particular feature: It may be that the endcaps will have to be shifted from their nominal design position in order to make room for cables. This potential offset has been implemented in the LAr endcap geometry routines. The source of the "z–shift" is the `LArRootDetDescr` class. If the endcap z–shift is to be changed, or to be dependent on run number (for example), the change should be made only in this class.

Notes on future revisions

If a common "Materials Manager" is ever adopted by Geant4 or by the ATLAS development team, it could be used within `LArDetectorConstruction`.

The `LArDetectorConstruction` class will eventually have to be integrated into a global detector geometry definition for the entire ATLAS detector.

The constants and code within `LArBarrelSolid`, `LArBarrelCalculator`, `LArWheelSolid`, and `LArWheelCalculator` should eventually be replaced by a detector database. `LArRootDetDescr` is a step in this direction.

`LArWheelSolid` is not very efficient in terms of execution speed compared to `LArBarrelSolid`. There may be room for improvement.

LAr EM Calorimeter Hits

Implementation notes

In order to take advantage of the Geant4 hit mechanism, the following three steps are required:

1. A hit class that inherits from `G4VHit` must be defined by the user. This class defines what a "hit" means to the Geant4 simulation.
2. A sensitive–detector class that inherits from `G4VSensitiveDetector` must be defined by the user. This class describes how to take energy deposited by particle in

the simulation and convert it into hit information.

3. Geant4 must be told which logical volume(s) are to be associated with the sensitive–detector class defined in step (2). This is normally done in the detector geometry definition routine.

Note: Geant4 also includes a digitization mechanism, but this is not used in LArHits.

Hit class

The hit class in LArHits is `LArEMHit`. The essential items of information associated with a `LArEMHit` are the calorimeter cell in which the energy is deposited, the hit energy, and the time. The calorimeter cell identifier is stored in an object of class `LArRootCellID`.

Rather than storing the time, the quantity stored in the `LArEMHit` class is energy*time. The reason is that, for in–time hits in a cell, the total energy is the sum of all energy deposited, and the hit time is the energy–weighted average of the time of the energy depositions. It's easier to compute the time average if energy*time is stored and re–summed for each energy deposit in a cell.

Sensitive–detector class

In Geant4, the collections of hits associated with a given `G4Event` object are stored in a `G4HCofThisEvent` object. Since an event can have many kinds of hits (calorimeter hits, tracker hits, etc.), the hits are stored in a templated collection class; `G4THitsCollection<LArEMHit>` in this case (this is defined in the header file `LArEMHit.hh`).

Since the `ProcessHits` method of a sensitive–detector class is called each time a particle deposits energy in the sensitive detector, it is important that this routine be reasonably efficient.

The LAr EM sensitive–detector class also distinguishes between "in–time" and "out–of–time" hits; a hit is considered "out–of–time" if its energy is deposted at a time $(t - t_0) > 2.5$ *ns,* where t_0 is the time at the event vertex plus the time of travel to the point of energy deposition at the speed of light.

These requirements are implemented using the following algorithm:

- If an energy deposition is out–of–time, create a separate hit and add it to an "out–of–time" hit container. This container is implemented as an STL vector, which is as efficient as any other STL sequence for inserting new objects at the end.
- If an energy deposition is in–time, and in a cell in which no energy has yet been deposited, create a new hit and add it to an "in–time" hit container. This container is implemented as an STL map, which offers methods for binary searches to test if a particular cell already exists in the map.
- If an energy deposition is in–time, and in a cell in which there has been an energy deposit, add the energy to the existing hit in the hit collection.
- In the `EndOfEvent` method of `LArEMcalSD`, add the events in both the in–time

and out−of−time containers to the G4Event's hit containers.

Calculator interface

1. There is only one sensitive−detector class in LArHits: LArEMcalSD. This class is responsible for creating hits, and for adding each hit to a hit collection. This class does not depend on which detector receives the hits; there is no need to define a different hit−collection class for each detector.

The reason why there is only one sensitive−detector class, even though LArHits contains more than one type of detector, is that there is the only detector−specific algorithm required by the sensitive detector: FindCell(G4ThreeVector), which calculates a cell identifier given the position of the energy deposition. This algorithm is implemented in the appropriate calculator class, LArBarrelCalculator or LArWheelCalculator.

The abstract interface LArVEMCalculator defines the routines to be called; both LArBarrelCalculator and LArWheelCalculator implement this interface.

Logical volume

The logical volume of the barrel calorimeter is associated with a sensitive detector near the end of LArBarrelConstruction with the following statements:

```
G4String barrelSDname = "/LArCalorimeter/LArEM/barrel";
LArVEMCalculator* calculator =
   LArBarrelCalculator::GetCalculator();
LArEMcalSD* barrelSD =
   new LArEMcalSD(barrelSDname,calculator);
SDman->AddNewDetector(barrelSD);
Stac_log->SetSensitiveDetector(barrelSD);
```

As noted in the previous section, there is only one sensitive−detector routine in LArHits. In order for the routine to process the hits for a given detector, it needs to know two items of information: the name of the detector, in order to give a name to its hit collection; and which calculator to use in order to determine which cell was hit. The above code fragment (which has detailed comments in the source code) shows how this information is passed to the sensitive detector.

The logical volume Stac_log is defined to be the liquid−argon toroidal cylinder that contains the calorimeter. The volumes that define the lead absorbers and the electrodes are positioned within this cylinder.

If a particle deposits energy within a absorber or an electrode, the simulation treats the deposition as if it occurred in the associated absorber or electrode logical volume, not within the liquid−argon volume of Stac_log. The result is that the ProcessHits method of LArEMcalSD is only called if the energy deposition occurs inside the liquid−argon portion of the calorimeter.

The endcap sensitive detector is similarly defined in LArEndcapConstruction.

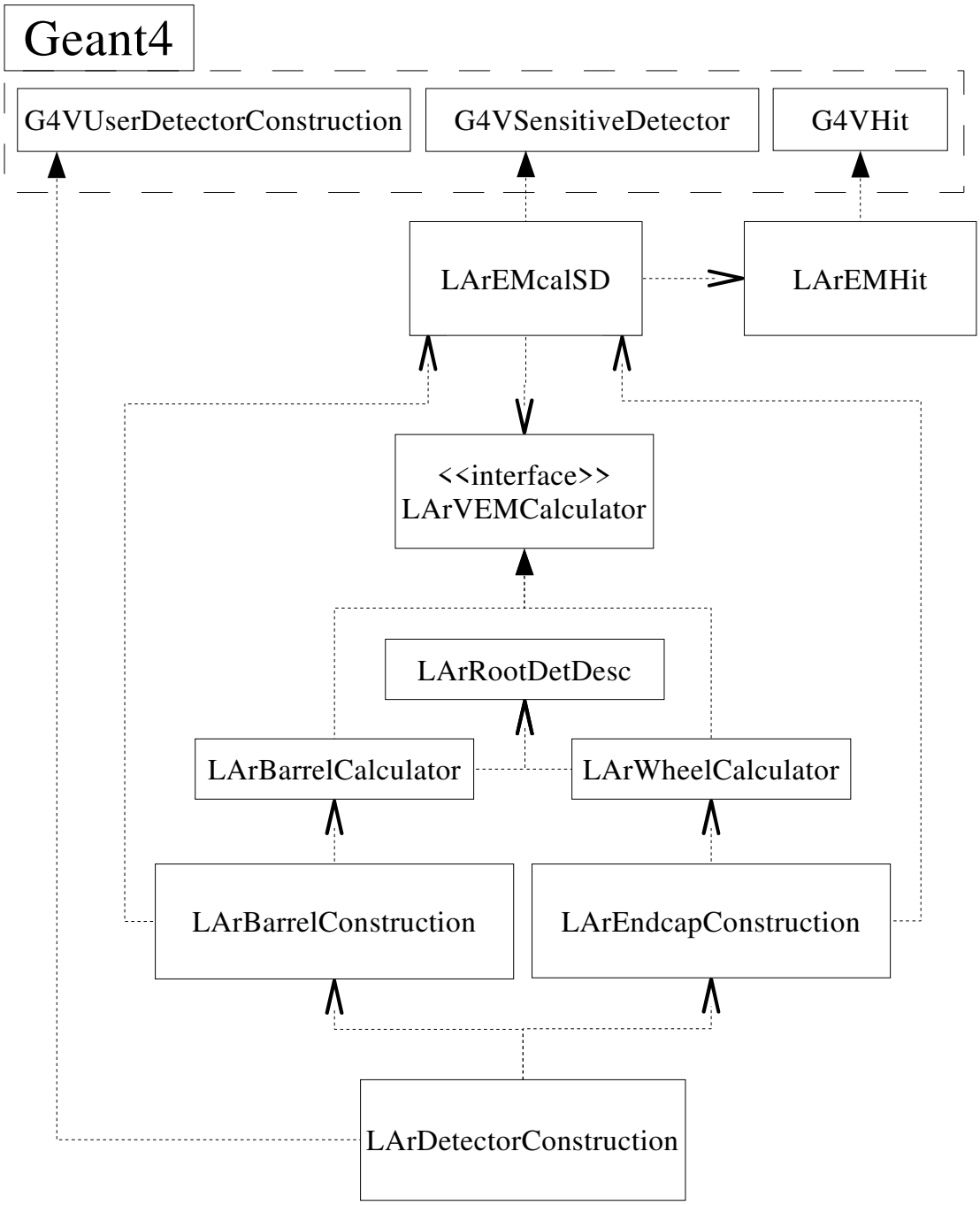

Figure 1: Class diagram for the geometry, hit, and calculator classe. For simplicity, the inheritance of the custom accordion solids (`LArBarrelSolid` and `LArWheelSolid`) from `G4VSolid` is not shown.

Notes on future revisions

The Geant4 hit mechanism is complex, but the benefit is flexibility in organizing and processing hits. If it were needed, the out−of−time calorimeter hits could be stored in a separate hit collection.

Discussions on the details of the endcap geometry indicate that the sensitive−detector mechanism within Geant4 may not be enough to accurately model the readout geometry of the ATLAS detector. Fortunately, Geant4 already has a readout geometry mechanism. The sensitive−detector model in LArHits should be revised to the readout−geometry model to enable better accuracy of the simulation.

ROOT−based Event Persistency

Implementation notes

The event persistency in LArHits makes use of the abstract base class `RecorderBase` described at <http://www.nevis.columbia.edu/~atlas/lar/>. Since LArHits uses the ROOT I/O scheme for its persistency, the name of the class that implements `RecorderBase` is `LArRootPersistency`.

The actual persistent classes written into a ROOT Tree are `LArRootEvent`, `LArRootPrimary`, `LArRootHit`, and `LArRootCellID` which contain event, primary particle, hit, and cell identifier data respectively. These classes are completely ROOT−based and contain no reference to any Geant4 data types. They correspond (roughly) to the Geant4−based classes `G4Event`, `G4PrimaryParticle`, `G4PrimaryVertex`, and `LArEMcalHit`.

The reason for defining completely ROOT−based classes is to take advantage of the features that ROOT makes available to such classes: automatic schema generation, automatic streamer I/O generation, automatic HTML documentation generation, etc.

The procedures for converting the Geant4 classes into the ROOT classes are contained in the conversion classes: `LArPersistentRootEvent`, `LArPersistentRootPrimary`, and `LArPersistentRootHit`. These are the "bridge" classes that connect Geant4 to ROOT.

There are several forms of I/O available in the ROOT Toolkit. LArHits uses the most efficient scheme identified in the study available at <http://www.nevis.columbia.edu/~atlas/lar/iostudy−20010404.html>: ROOT `TTrees` with all containers implemented as `TClonesArrays`, written with splitlevel=1 and compression=1.

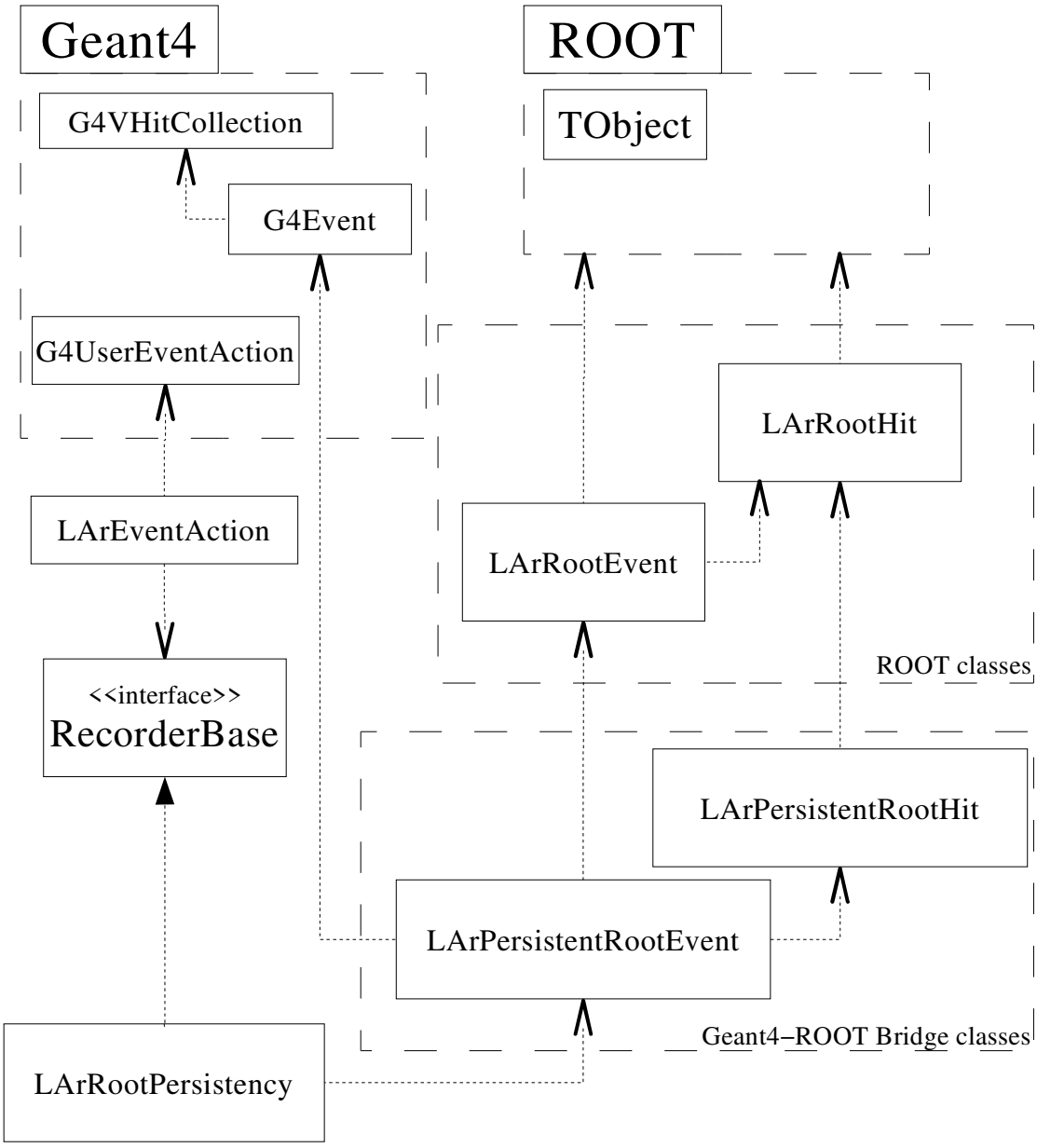

Figure 2: Class diagram for persistency scheme in LArHits. For simplicity, only the relationships for event and hit persistency are shown; the classes associated with primary particles and cell identifiers are omitted.

Notes on future revisions

The ROOT−based persistency mechanism should be adapted for StoreGate in the Athena framework.

The `LArRootCellID` class should be upgraded to the Athena `Identifier` class.

The class names should be changed to better fit with the Athena scheme.

Acknowledgments

The author wishes to thanks all those people working on the ATLAS LAr electromagnetic calorimeter in Geant4.