Geant4 User's Guide for Toolkit Developers

Version: geant4 10.2

Publication date 4 December 2015

Geant4 Collaboration

Geant4 User's Guide for Toolkit Developers

by Geant4 Collaboration

Version: geant4 10.2

Publication date 4 December 2015

Table of Contents

Chapter 1. Introduction

1.1. Scope of this manual

The User's Guide for Toolkit Developers provides detailed information about the design of Geant4 classes as well as the information required to extend the current functionality of the Geant4 toolkit. This manual is designed to:

- provide a repository of information for those who want to understand or refer to the detailed design of the toolkit, and
- provide details and procedures for extending the functionality of the toolkit so that experienced users may contribute code which is consistent with the overall design of Geant4.

This manual is intended for developers and experienced users of Geant4. It is assumed that the reader is already familiar with functionality of the Geant4 toolkit as explained in the "User's Guide For Application Developers", and also has a working knowledge of programming using C++. A knowledge of object-oriented analysis and design will also be useful in understanding this manual. It is also useful to consult the ``Software Reference Manual" which provides a list of Geant4 classes and their major methods.

Detailed discussions of the physics included in Geant4 are provided in the ``Physics Reference Manual".

1.2. How to use this manual

Part I: to understand the goal of the software design of Geant4, it is useful to begin by reading the User Requirements Document referred to in the next section.

Part II: ``Design and Function of the Geant4 Categories" provides detailed information about the design of each class category and the classes in it. Before considering an extension of one of the toolkit categories, a detailed understanding of that category is required.

Part III: ``Extending Toolkit Functionality" explains in some detail how to extend the functionality of Geant4. Most of the class categories are covered and some, which are especially useful to most users, are covered in greater detail.

It is not necessary to understand the entire manual before adding a new functionality. To add a new physics process, for example, only the following items must be read and understood:

- the design principle described in the ``Physics processes" chapter of Part II
- techniques explained in the ``Physics processes" chapter of Part III.

1.3. User Requirements Document

At the beginning of Geant4 development, a set of user requirements was collected in order to inform the object-oriented analysis and design of the toolkit. The User Requirements Document follows the PSS-05 software engineering standards and is available at

http://cern.ch/geant4/OOAandD/URD.pdf .

This document provides a general description of the main capabilities and constraints of the toolkit. It also defines three types of users characterized by their level of interaction with the system. Specific requirements are also listed and classified.

Chapter 2. Design and Function of Geant4 Categories

2.1. Introduction

Geant4 exploits advanced software engineering techniques based on the Booch/UML Object Oriented Methodology and follows the evolution of the ESA Software Engineering Standards for the development process. The "spiral", or iterative, approach has been adopted. User requirements were collected in the initial phase and problem domain decomposition, object-oriented methods, and CASE tools were used for analysis and design. This produced a clear hierarchical structure of sub-domains linked by a uni-directional flow of dependencies. This led to a software product which is modular and flexible (a toolkit) and in which the physics implementation is transparent and open to user validation of physics predictions. It allows the user to understand, customize and extend the toolkit in all domains. At the same time the modular architecture allows the user to load only needed components.

2.2. Run

2.2.1. Design Philosophy

The run category manages collections of events (*runs*). In a single run the events share the detector implementation, physics conditions and primary generation.

The classes associated with the run category can be considered as the main and higher level application programming interface (*API*) used in a Geant4 application. A simple applicaiton will use concrete classes provided with the toolkit, the developer will provide the detector description a primary genertor (possibly using one of the general purpose ones provided with the toolkit), define the physics for the application (the *physics list*, possibly one of the few provided with the toolkit) and optional *user actions* to interact with the simulation itself.

In few cases it is however necessary to modify the default behavior of one or more classes in this category to allow for a user-customization. As an example the class **G4MTRunManager** extends the basic run-manager class to take into account event level parallelism via multi-threading.

During a run some states of the application are invariant and cannot be modified: the physics list (i.e. the list of processes attached to each particle) and the detector layout (not that some geometry primitives allow for changing parameters during the event run: parametrizations. However technically the class instances representing the detector layout do not change during a run).

2.2.2. Class Design

The relevant classes for the run category are shown here. This show, in particular, the relation between classes for the case of a multi-threaded application. For a sequential applicaiton the diagram is simplified since no **G4WorkerRunManager** class exist and **G4MTRunManager** is replaced by **G4RunManager**:

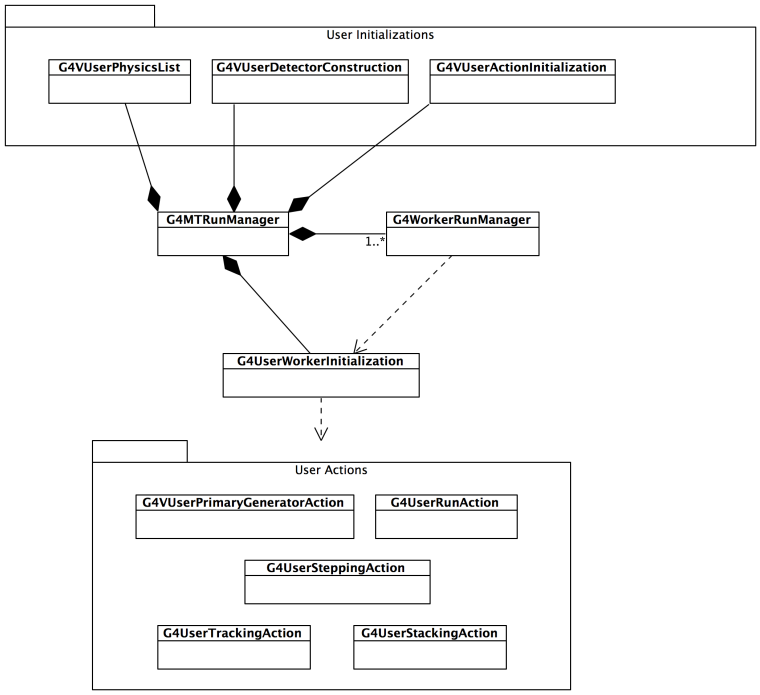

Figure 2.1. Class diagram for main run category classes

For a descriptiont of multi-threading functionality refer to "Parallelism in Geant4: multi-threading capabilities" chapter.

One of the main functions of the run category is to control the life-cycle of a Geant4 application, again with reference to the case of a multi-threaded application the following schema describes it:

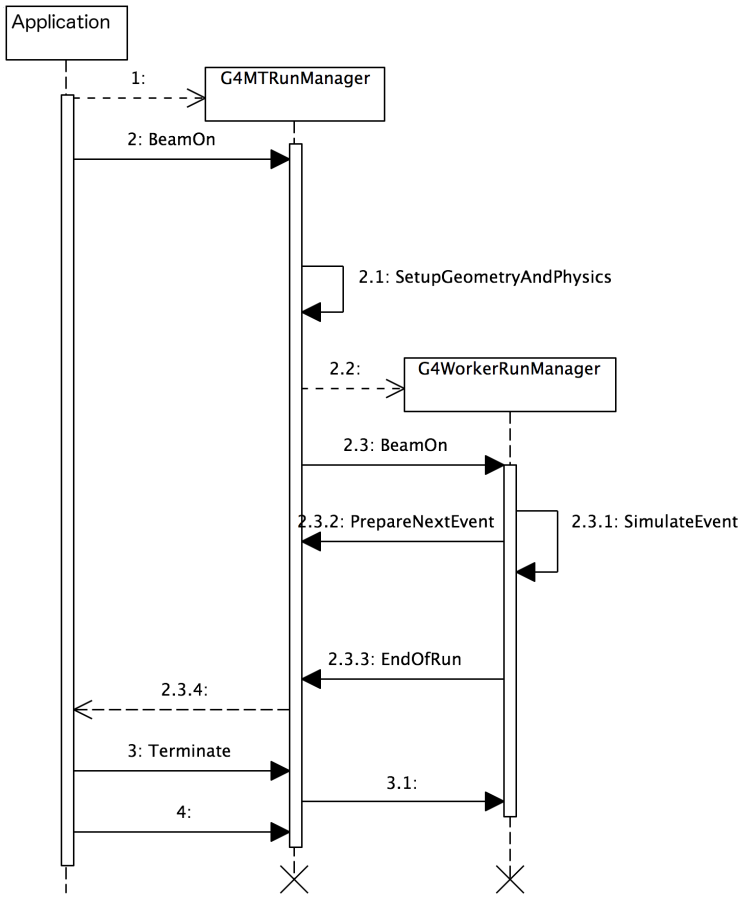

Figure 2.2. Life cycle of a Geant4 application and main run category classes

A list of the main classes for the category is provided:

- **G4Run** - This class represents a run. An object of this class is constructed and deleted by G4RunManager.
- **G4RunManager** - the run controller class. Users must register detector construction, physics list and primary generator action classes to it. G4RunManager or a derived class must be a singleton. This class provides several virtual methods that can be used to define user-specific behavior for a Geant4 application.
- **G4RunManagerKernel** - provides control of the Geant4 kernel. This class is constructed by G4RunManager. This class does not provide virtual methods and user should not sub-class from it. The applicaiton G4RunManager shold own an instance of a G4RunManagerKernel singleton.
- **G4{MT,Worker}RunManager[Kernel]** - specialized versions to provide a multi-threading enabled application. Refer to chapter "Parallelism in Geant4: multi-threading capabilities" for additional information.
- **G4VUserDetectorConstruction** - pure virtual base class that represents the simulation setup.
- **G4VUserParallelWorld** - pure virtual base class of the user's parallel world.
- **G4VUserPhysicsList** - pure virtual base class for a physcis list.
- **G4VUserPrimaryGeneratorAction** - pure virtual class used by user to define the primary generation.
- **G4VModularPhysicsList** - Pure virtual class to construct a physics list from **G4VPhysicsConstructor**. More modern and modular approach preferred in current versions of pre-packaged physics lists.
- **G4UserRunAction** - user action class for run. Instantiate user-derived G4Run and provides user-hooks for begin and end of run.
- **G4UserWorkerInitialization** and **G4UserWorkerThreadInitialization** - define here the concrete behavior for threading model. Both classes provide several virtual methods that can be modified in derived classes.
- **G4VUserActionInitialization** - pure virtual class used by user to instantiate concrete instances of the user-actions.
- **G4WorkerThread** - this class encapsulates thread-specific data.
- **G4RNGHelper** - helper class to register and use RNG seeds. Used by MT applicaitons to guarantee reproducibility.

2.3. Event

2.3.1. Design Philosophy

In high energy physics the primary unit of an experimental run is an event. The same concept is also known in other fields as *history*. We retain the name from the HEP community. An event consists of a set of primary particles, and a set of detector responses to these particles.

In Geant4, objects of the G4Event class are the primary units of a simulation run. Before the event is processed, it contains primary vertices and primary particles produced by a generator (a concrete implementation of a G4VPrimaryGenerator). After the event is processed, it may also contain hits, digitizations ,and optionally, trajectories generated by the simulation and additional user information (a sub-class of G4VUserEventInformation). The event category manages events and provides an abstract interface to the external generators.

G4Event and its content vertices and particles are independent of other classes. This isolation allows Geant4-based simulation programs to be independent of specific choices for physics generators and of specific solutions for storing the ``Monte Carlo truth''. G4Event avoids keeping any transient information which is not meaningful after event processing is complete. Thus the user can store objects of this class for processing further down the program chain. For performance reasons, G4Event and its content classes are not persistent. Instead the user must provide the transient-to-persistent conversion.

The current event being simulated is managed by G4EventManager, a singleton responsible of handling the simulation of the event. The tracks being followed for the current event are stored in a stack managed by G4StackManager. Different stacks allow for fine control of the simulation (urgent, waiting and postponed stacks).

User hooks allow for a customization of the simulation behavior via G4UserEventAction, G4UserStackingAction and G4VUserEventInformation.

Event generation is performed via a concrete implementation of a G4VPrimaryGenerator class. This is usually instantiated by the user in the user-defined concrete implementation of G4VUserPrimaryGeneratorAction (belonging to the run category). Geant4 provides three concrete implementation of G4VPrimaryGenerator: G4ParticleGun, a simple generator that can shoot one or more primaries; G4HEPEvtInterface, specifically desinged for HEP experiments to read /HEPEVT/ common block format; and the G4GeneralParticleSource able to generate primaries distributed according to complex and configurable distributions. This last possibility is described in detail in the Application Developers Guide.

2.3.2. Class Design

- **G4Event** - this class represents an event. It is constructed and deleted by G4RunManager or its derived class.
- **G4EventManager** - this class controls an event. It must be a singleton and should be constructed by G4RunManager.
- **G4TrajectoryContainer** - this class can contain the concret G4VTrajectory objects defined by user or used to display the current event.
- **G4UserEventAction** - the abstract base class to allow for a user to inject code at the beginning and end of an event.
- **G4UserStackingAction** - the abstract base class to allow for the user to control and tune the stacking of particles. See documentation in class and Geant4 examples.
- **G4StackManager** - controls the stacks of tracks belonging to the event currently being processed. The three stacks are: urgent, waiting and postponed. The first is of type G4SmatrTrackStack while the other two are of the simples G4TrackStack type.
- **G4VPrimaryGenerator** - the abstract base class of all of primary generators. This class has only one pure virtual method, GeneratePrimaryVertex(), which takes a G4Event object, generates a primary vertex and associates primary particles with the vertex.

UML class diagrams for classes related to the event and event generator classes are shown in Figure 2.3.

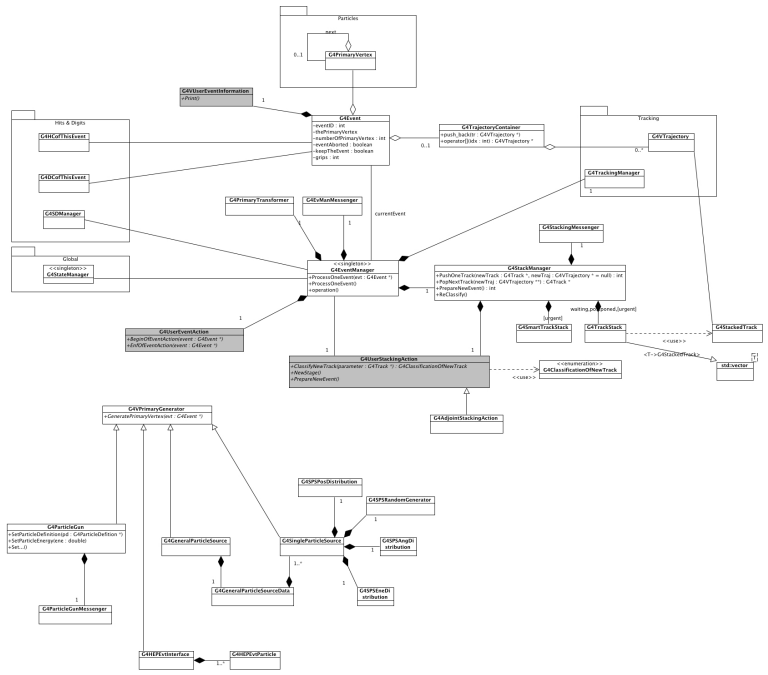

Figure 2.3. Event Category UML Diagram. Classes in grey are to be sub-classed by user.

2.4. Tracking

The tracking category manages the contribution of the processes to the evolution of a track's state and provides information in sensitive volumes for hits and digitization.

2.4.1. Design Philosophy

It is well known that the overall performance of a detector simulation depends critically on the CPU time spent propagating the particle through one step. The most important consideration in the object design of the tracking category is maintaining high execution speed in the Geant4 simulation while utilizing the power of the object-oriented approach.

An extreme approach to the particle tracking design would be to integrate all functionalities required for the propagation of a particle into a single class. This design approach looks object-oriented because a particle in the real world propagates by itself while interacting with the material surrounding it. However, in terms of data hiding, which is one of the most important ingredients in the object-oriented approach, the design can be improved.

Combining all the necessary functionalities into a single class exposes all the data attributes to a large number of methods in the class. This is basically equivalent to using a common block in Fortran.

Instead of the 'big-class' approach, a hierarchical design was employed by Geant4. The hierarchical approach, which includes inheritance and aggregation, enables large, complex software systems to be designed in a structured way. The simulation of a particle passing through matter is a complex task involving particles, detector geometry, physics interactions and hits in the detector. It is well-suited to the hierarchical approach. The hierarchical design manages the complexity of the tracking category by separating the system into layers. Each layer may then be designed independently of the others.

In order to maintain high-performance tracking, use of the inheritance ('is-a' relation) hierarchy in the tracking category was avoided as much as possible. For example, `track` and `particle` classes might have been designed so that a `track` 'is a' `particle`. In this scheme, however, whenever a `track` object is used, time is spent copying the data from the `particle` object into the `track` object. Adopting the aggregation ('has-a' relation) hierarchy requires only pointers to be copied, thus providing a performance advantage.

2.4.2. Class Design

Figure 2.4 shows a general overview of the tracking design in Unified Modelling Language Notation.

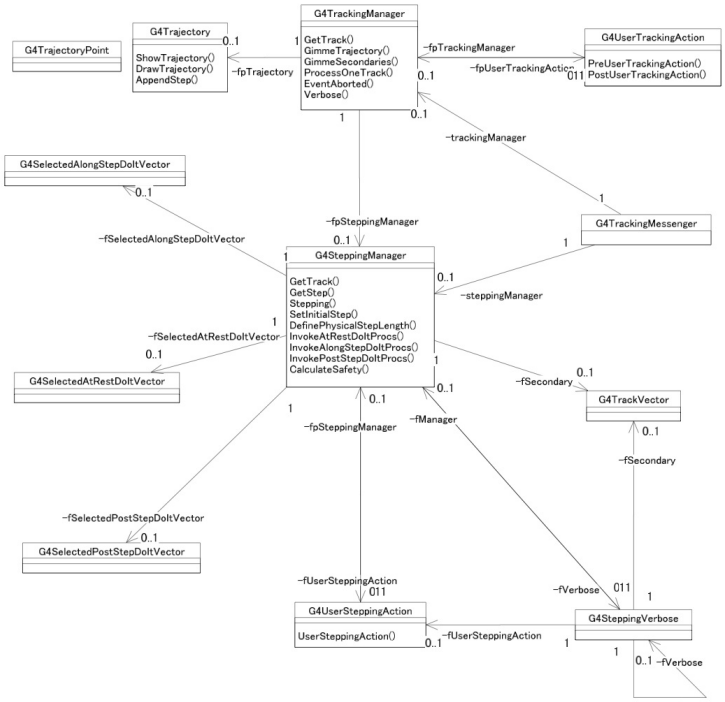

Figure 2.4. Tracking design

- **G4TrackingManager** is an interface between the event and track categories and the tracking category. It handles the message passing between the upper hierarchical object, which is the event manager (`G4EventManagerz`), and lower hierarchical objects in the tracking category. `G4TrackingManager` is responsible for processing one track which it receives from the event manager.

  `G4TrackingManager` aggregates the pointers to `G4SteppingManager`, `G4Trajectory` and `G4UserTrackingAction`. It also has a 'use' relation to `G4Track`.
- **G4SteppingManager** plays an essential role in particle tracking. It performs message passing to objects in all categories related to particle transport, such as geometry and physics processes. Its public method `Stepping()` steers the stepping of the particle. The algorithm employed in this method is basically the same as that in Geant3. The Geant4 implementation, however, relies on the inheritance hierarchy of the physics interactions. The hierarchical design of the physics interactions enables the stepping manager to handle them as abstract objects. Hence, the manager is not concerned with concrete interaction objects such as bremsstrahlung or pair creation. The actual invocations of various interactions during the stepping are done through a dynamic binding mechanism. This mechanism shields the tracking category from any change in the design of the physics process classes, including the addition or subtraction of new processes.

  `G4SteppingManager` also aggregates
  - the pointers to `G4Navigator` from the geometry category, to the current `G4Track`, and
  - the list of secondaries from the current track (through a `G4TrackVector`) to `G4UserSteppingAction` and to `G4VSteppingVerbose`.

  It also has a 'use' relation to `G4ProcessManager` and `G4ParticleChange` in the physics processes class category.
- **G4Track** - the class `G4Track` represents a particle which is pushed by `G4SteppingManager`. It holds information required for stepping a particle, for example, the current position, the time since the start of stepping, the identification of the geometrical volume which contains the particle, etc. Dynamic information, such as particle momentum and energy, is held in the class through a pointer to the `G4DynamicParticle` class. Static information, such as the particle mass and charge is stored in the `G4DynamicParticle` class through the pointer to the `G4ParticleDefinition` class. Here the aggregation hierarchical design is extensively employed to maintain high tracking performance.

- **G4TrajectoryPoint and G4Trajectory** - the class `G4TrajectoryPoint` holds the state of the particle after propagating one step. Among other things, it includes information on space-time, energy-momentum and geometrical volumes.

  `G4Trajectory` aggregates all `G4TrajectoryPoint` objects which belong to the particle being propagated. `G4TrackingManager` takes care of adding the `G4TrajectoryPoint` to a `G4Trajectory` object if the user requested it (see Geant4 User's Guide - For Application Developers. The life of a `G4Trajectory` object spans an event, contrary to `G4Track` objects, which are deleted from memory after being processed.
- **G4UserTrackingAction and G4UserSteppingAction** - `G4UserTrackingAction` is a base class from which user actions at the beginning or end of tracking may be derived. Similarly, `G4UserSteppingAction` is a base class from which user actions at the beginning or end of each step may be derived.

2.4.3. Tracking Algorithm

The key classes for tracking in Geant4 are `G4TrackingManager` and `G4SteppingManager`. The singleton object "TrackingManager" from `G4TrackingManager` keeps all information related to a particular track, and it also manages all actions necessary to complete the tracking. The tracking proceeds by pushing a particle by a step, the length of which is defined by one of the active processes. The "TrackingManager" object delegates management of each of the steps to the "SteppingManager" object. This object keeps all information related to a particular step.

The public method `ProcessOneTrack()` in `G4TrackingManager` is the key to managing the tracking, while the public method `Stepping()` is the key to managing one step. The algorithms used in these methods are explained below.

ProcessOneTrack() in G4TrackingManager

1. Actions before tracking the particle: Clear secondary particle vector
2. Pre tracking user intervention process.
3. Construct a trajectory if it is requested
4. Give SteppingManager the pointer to the track which will be tracked
5. Inform beginning of tracking to physics processes
6. Track the particle Step-by-Step while it is alive
   - Call Stepping method of G4SteppingManager
   - Append a trajectory point to the trajectory object if it is requested
7. Post tracking user intervention process.
8. Destroy the trajectory if it was created

Stepping() in G4SteppingManager

1. Initialize current step
2. If particle is stopped, get the minimum life time from all the at rest processes and invoke InvokeAtRestDoIt-Procs for the selected AtRest processes
3. If particle is not stopped:
   - Invoke DefinePhysicalStepLength, that finds the minimum step length demanded by the active processes
   - Invoke InvokeAlongStepDoItProcs
   - Update current track properties by taking into account all changes by AlongStepDoIt
   - Update the **safety**
   - Invoke PostStepDoIt of the active discrete process.
   - Update the track length
   - Send G4Step information to Hit/Dig if the volume is sensitive
   - Invoke the user intervention process.
   - Return the value of the StepStatus.

2.4.4. Interaction with Physics Processes

The interaction of the tracking category with the physics processes is done in two ways. First each process can limit the step length through one of its three `GetPhysicalInteractionLength()` methods, AtRest, AlongStep, or PostStep. Second, for the selected processes the DoIt (AtRest, AlongStep or PostStep) methods are invoked.

All this interaction is managed by the Stepping method of `G4SteppingManager`. To calculate the step length, the `DefinePhysicalStepLength()` method is called. The flow of this method is the following:

- Obtain maximum allowed Step in the volume define by the user through G4UserLimits.
- The PostStepGetPhysicalInteractionLength of all active processes is called. Each process returns a step length and the minimum one is chosen. This method also returns a G4ForceCondition flag, to indicate if the process is forced or not:
  - Forced : Corresponding PostStepDoIt is forced.
  - NotForced : Corresponding PostStepDoIt is not forced unless this process limits the step.
  - Conditionally : Only when AlongStepDoIt limits the step, corresponding PoststepDoIt is invoked.
  - ExclusivelyForced : Corresponding PostStepDoIt is exclusively forced.

  All other DoIt including AlongStepDoIts are ignored.
- The AlongStepGetPhysicalInteractionLength method of all active processes is called. Each process returns a step length and the minimum of these is chosen. This method also returns a fGPILSelection flag, to indicate if the process is the selected one can be is forced or not:
  - CandidateForSelection: this process can be the winner. If its step length is the smallest, it will be the process defining the step (the process
  - NotCandidateForSelection: this process cannot be the winner. Even if its step length is taken as the smallest, it will not be the process defining the step

The method `G4SteppingManager::InvokeAlongStepDoIts()` is in charge of calling the AlongStepDoIt methods of the different processes:

- If the current step is defined by a 'ExclusivelyForced' PostStepGetPhysicalInteractionLength, no AlongStepDoIt method will be invoked
- Else, all the active continuous processes will be invoked, and they return the ParticleChange. After it for each process the following is executed:
  - Update the G4Step information by using final state information of the track given by a physics process. This is done through the UpdateStepForAlongStep method of the ParticleChange
  - Then for each secondary:
    - It is checked if its kinetic energy is smaller than the energy threshold for the material. In this case the particle is assigned a 0. kinetic energy and its energy is added as deposited energy of the parent track. This check is only done if the flag ApplyCutFlag is set for the particle (by default it is set to 'false' for all particles, user may change it in its G4VUserPhysicsList). If the track has the flag IsGoodForTracking 'true' this check will have no effect (used mainly to track particles below threshold)
    - The parentID and the process pointer which created this track are set
    - The secondary track is added to the list of secondaries. If it has 0. kinetic energy, it is only added if it it invokes a rest process at the beginning of the tracking
  - The track status is set according to what the process defined

The method `G4SteppingManager::InvokePostStepDoIts` is on charge of calling the PostStepDoIt methods of the different processes.

- Invoke the PostStepDoIt methods of the specified discrete process (the one selected by the PostStepGetPhysicalInteractionLength, and they return the ParticleChange. The order of invocation of processes is inverse to the order used for the GPIL methods. After it for each process the following is executed:
  - Update PostStepPoint of Step according to ParticleChange
  - Update G4Track according to ParticleChange after each PostStepDoIt
  - Update safety after each invocation of PostStepDoIts
  - The secondaries from ParticleChange are stored to SecondaryList
  - Then for each secondary:
    - It is checked if its kinetic energy is smaller than the energy threshold for the material. In this case the particle is assigned a 0. kinetic energy and its energy is added as deposited energy of the parent track. This check is only done if the flag ApplyCutFlag is set for the particle (by default it is set to 'false' for all particles, user may change it in its G4VUserPhysicsList). If the track has the flag IsGoodForTracking 'true' this check will have no effect (used mainly to track particles below threshold)
    - The parentID and the process pointer which created this track are set
    - The secondary track is added to the list of secondaries. If it has 0. kinetic energy, it is only added if it it invokes a rest process at the beginning of the tracking

- The track status is set according to what the process defined

The method `G4SteppingManager::InvokeAtRestDoIts` is called instead of the three above methods in case the track status is **fStopAndALive**. It is on charge of selecting the rest process which has the shortest time before and then invoke it:

- To select the process with shortest tiem, the AtRestGPIL method of all active processes is called. Each process returns an lifetime and the minimum one is chosen. This method return also a G4ForceCondition flag, to indicate if the process is forced or not: = Forced : Corresponding AtRestDoIt is forced. = NotForced : Corresponding AtRestDoIt is not forced unless this process limits the step.
- Set the step length of current track and step to 0.
- Invoke the AtRestDoIt methods of the specified at rest process, and they return the ParticleChange. The order of invocation of processes is inverse to the order used for the GPIL methods.

  After it for each process the following is executed:
  - Set the current process as a process which defined this Step length.
  - Update the G4Step information by using final state information of the track given by a physics process. This is done through the UpdateStepForAtRest method of the ParticleChange.
  - The secondaries from ParticleChange are stored to SecondaryList
  - Then for each secondary:
    - It is checked if its kinetic energy is smaller than the energy threshold for the material. In this case the particle is assigned a 0. kinetic energy and its energy is added as deposited energy of the parent track. This check is only done if the flag ApplyCutFlag is set for the particle (by default it is set to 'false' for all particles, user may change it in its G4VUserPhysicsList). If the track has the flag IsGoodForTracking 'true' this check will have no effect (used mainly to track particles below threshold)
    - The parentID and the process pointer which created this track are set
    - The secondary track is added to the list of secondaries. If it has 0. kinetic energy, it is only added if it it invokes a rest process at the beginning of the tracking
  - The track is updated and its status is set according to what the process defined

2.4.5. Ordering of Methods of Physics Processes

The ProcessManager of a particle is responsible for providing the correct ordering of process invocations. `G4SteppingManager` invokes the processes at each phase just following the order given by the ProcessManager of the corresponding particle.

For some processes the order is important. Geant4 provides by default the right ordering. It is always possible for the user to choose the order of process invocations at the initial set up phase of Geant4. This default ordering is the following:

1. Ordering of GetPhysicalInteractionLength
   - In the loop of GetPhysicalInteractionLength of AlongStepDoIt, the Transportation process has to be invoked at the end.
   - In the loop of GetPhysicalInteractionLength of AlongStepDoIt, the Multiple Scattering process has to be invoked just before the Transportation process.
2. Ordering of DoIts
   - There is only some special cases. For example, the Cherenkov process needs the energy loss information of the current step for its DoIt invocation. Therefore, the EnergyLoss process has to be invoked before the Cherenkov process. This ordering is provided by the process manager. Energy loss information necessary for the Cherenkov process is passed using G4Step (or the static dE/dX table is used together with the step length information in G4Step to obtain the energy loss information). Any other?

2.5. Physics Processes

2.5.1. Design Philosophy

The processes category contains the implementations of particle transportation and physical interactions. All physics process conform to the basic interface `G4VProcess`, but different approaches have been developed for the detailed design of each sub-category.

For the decay sub-category, the decays of all long-lived, unstable particles are handled by a single process. This process gets the step length from the mean life of the particle. The generation of decay products requires a knowledge of the branching ratios and/or data distributions stored in the particle class.

The electromagnetic sub-category is divided further into the following packages:

- `standard`: handling basic properties for electron, positron, photon and hadron interactions,
- `lowenergy`: providing alternative models extended down to lower energies than the standard package,
- `dna`: providing DNA physics and chemistry simulation,
- `highenergy`: providing models for rare high energy processes,
- `muons`: handling muon interactions and energy loss propagator,
- `adjoint`: implementing reverse Monte Carlo approach,
- `xrays`: providing specific code for x-ray physics,
- `optical`: providing specific code for optical photons,
- `utils`: collecting utility classes used by the above packages.

It provides the features of openness and extensibilty resulting from the use of object-oriented technology; alternative physics models, obeying the same process abstract interface, are often available for a given type of interaction.

For hadronic physics, an additional set of implementation frameworks was added to accommodate the large number of possible modeling approaches. The top-level framework provides the basic interface to other Geant4 categories. It satisfies the most general use-case for hadronic shower simulations, namely to provide inclusive cross sections and final state generation. The frameworks are then refined for increasingly specific use-cases, building a hierarchy in which each level implements the interface specified by the level above it. A given hadronic process may be implemented at any one of these levels. For example, the process may be implemented by one of several models, and each of the models may in turn be implemented by several sub-models at the lower framework levels.

The hadronic sub-category is divided into the following packages:

- `management`: providing the top level hadronic process classes;
- `cross_sections`: providing inelastic and elastic cross sections for hadron-nucleon, hadron-nucleus and nucleus-nucleus interactions; it also contains inelastic cross sections for gamma- and lepto-nuclear interactions;
- `models`: providing hadronic final-state models; there is a further sub-level, corresponding to each model (`abla, abrasion binary_cascade, cascade, coherent_elastic, de_excitation, em_dissociation, fission, im_r_matrix, inclxx, lend, lepto_nuclear, management, parton_string, pre_equilibrium, qmd, quasi_elastic, radioactive_decay, rpg, theo_high_energy, util`);
- `processes`: providing the in-flight hadronic physics processes: inelastic, elastic, capture and fission;
- `stopping`: providing the nuclear capture of hadrons and muon at rest;
- `util`: collecting utility classes used by the above packages.

2.5.2. Class Design

2.5.2.1. General

The object-oriented design of the generic physics process `G4VProcess` and its relation to the process manager is shown in Figure 2.5. Figure 2.6 shows how specific physics processes are related to `G4VProcess`.

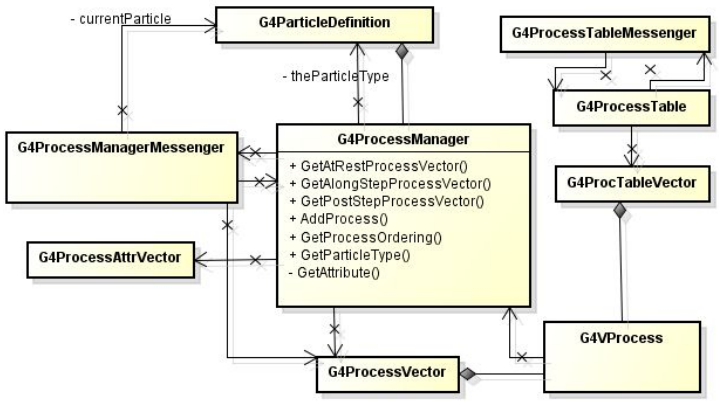

Figure 2.5. Management of Physics Processes

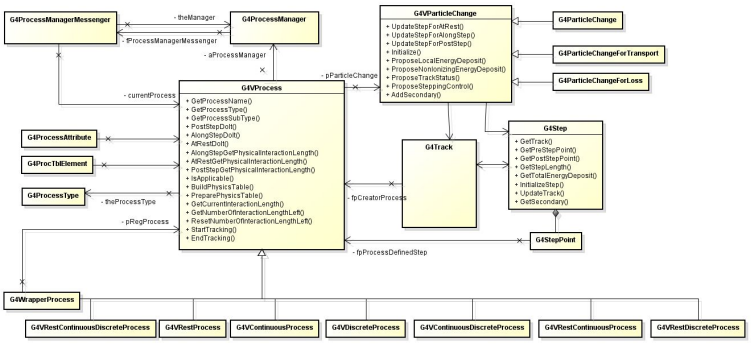

Figure 2.6. Management of Physics Processes

2.5.3. Electromagnetic

The electromagnetic (EM) processes of Geant4 follow the basic interfaces:

- `G4VEnergyLossProcess`;
- `G4VEmProcess`;
- `G4VMultipleScattering`.

The class diagram is shown in Figure 2.7.

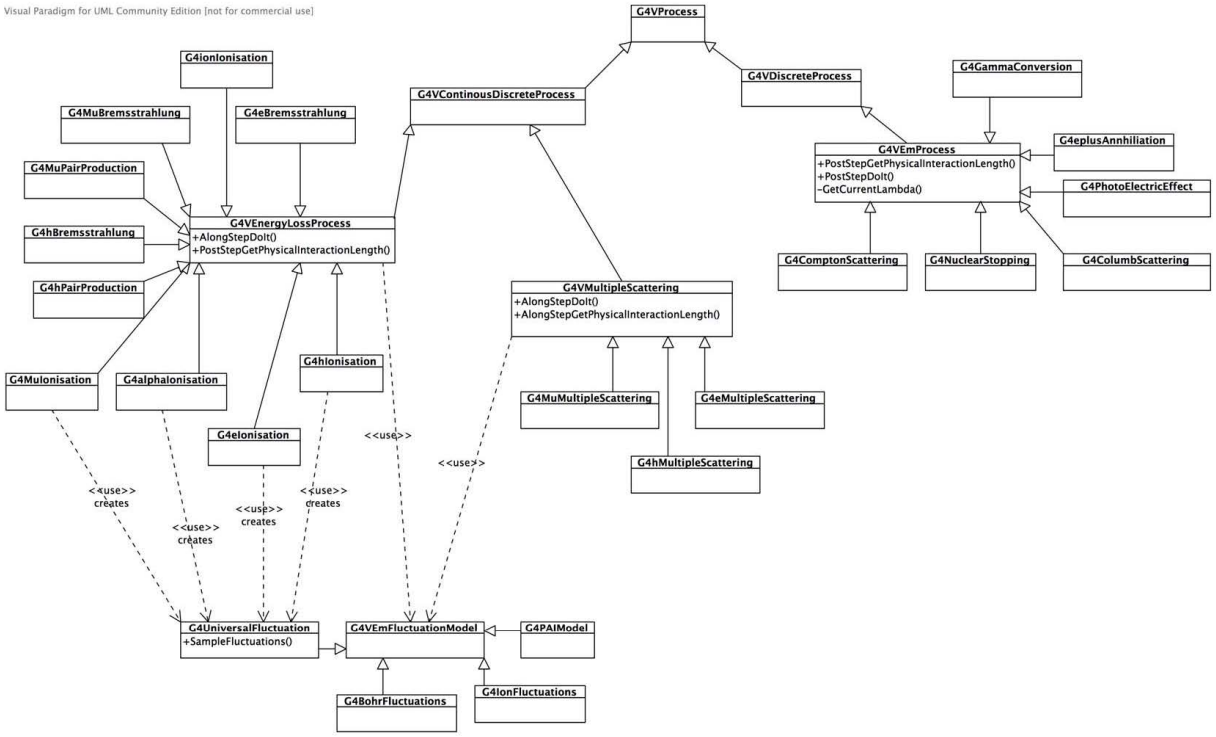

Figure 2.7. Design of EM physics processes.

These base classes provide all management work of initialisation of processes, creation and filling of physics tables, and generic run-time actions. Concrete process classes are responsible for the initialisation of parameters and defining the set of models for the process. In some specific cases these interfaces are not applicable and the high level interface `G4VProcess` is used.

Concrete physics models are implemented via EM model interfaces:

- `G4VEmModel`;
- `G4VMscModel`.

In the majority of use-cases when new EM physics is needed, it is enough to create only a new model class and use it in the existing EM process class. A new model may be added to an existing process using `AddEmModel(G4int, G4VEmModel*, G4Region*)` method. The class diagram is shown in Figure 2.8.

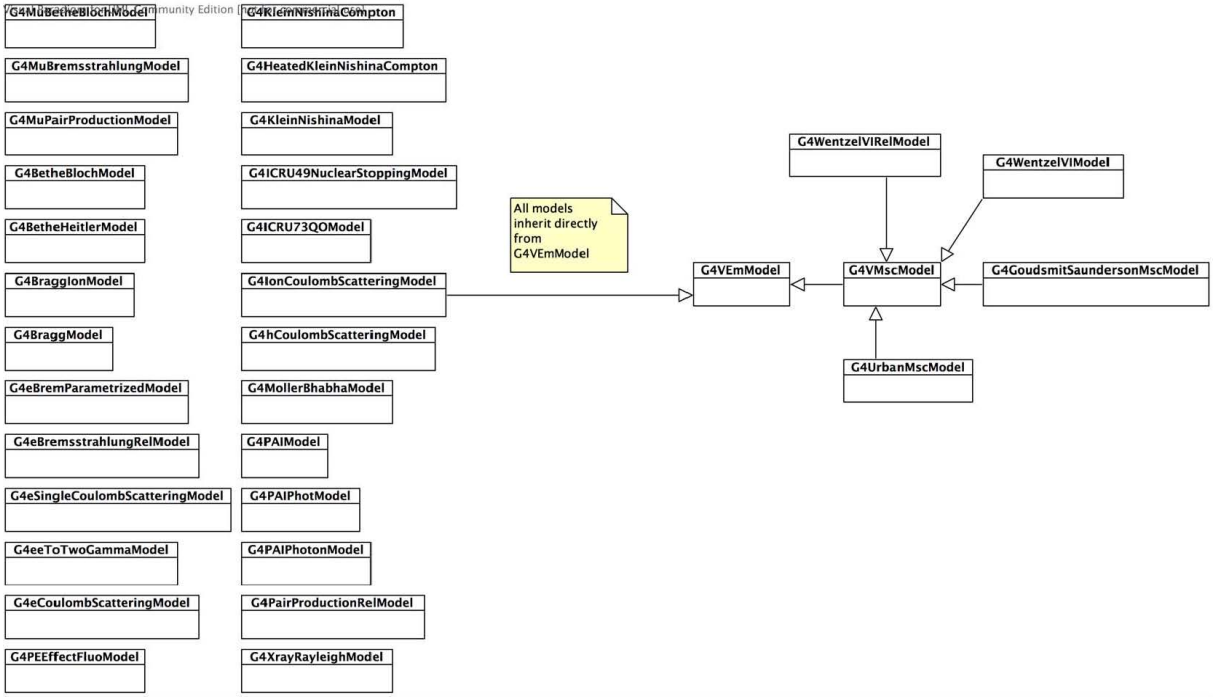

Figure 2.8. Design of EM physics models.

2.5.4. Hadronic

The hadronic physics of Geant4 has been designed to allow three basic types of modeling: data-driven, parametrisation-driven, and theory-driven. Five implementation frameworks have been used to allow great flexibility in these modeling approaches. An overview of the first two framework levels will be given here (for a wider and more detailed coverage please refer to the next Chapter).

The top-level framework defines the hadronic processes, and provides the basic interface to other Geant4 categories. All processes have a common base-class `G4VProcess`, from which a set of specialised classes are derived. Two of them are used as base classes for hadronic processes for particles (`G4VDiscreteProcess`), and for processes like radioactive decay that can be both in-flight or at-rest (`G4VRestDiscreteProcess`). Each of these classes declares two types of methods: one for calculating the time to interaction (for at-rest processes) or the physical interaction length (for in-flight processes), allowing tracking to request the information necessary to decide on the process responsible for final-state production.

Note on at-rest processes: starting with Geant4 version `9.6` - when the Bertini and Fritiof final-state models have been extended down to zero kinetic energy and used also for simulating the nuclear capture at-rest - the at-rest processes derive from `G4HadronicProcess`, hence from `G4VDiscreteProcess`, instead than from `G4VRestProcess` as in the initial design of at-rest processes. This requires some adaptation a discrete process to handle an at-rest one using top level interface `G4VProcess`. A different solution, under consideration but not yet implemented, would be instead to have `G4HadronicProcess` inheriting from `G4VRestDiscreteProcess`: in this way, `G4HadronicProcess`, and therefore any theory-driven final-state model, could be deployed for any kind of hadronic process, including capture-at-rest processes and radioactive decays.

The class diagram is shown in Figure 2.9.

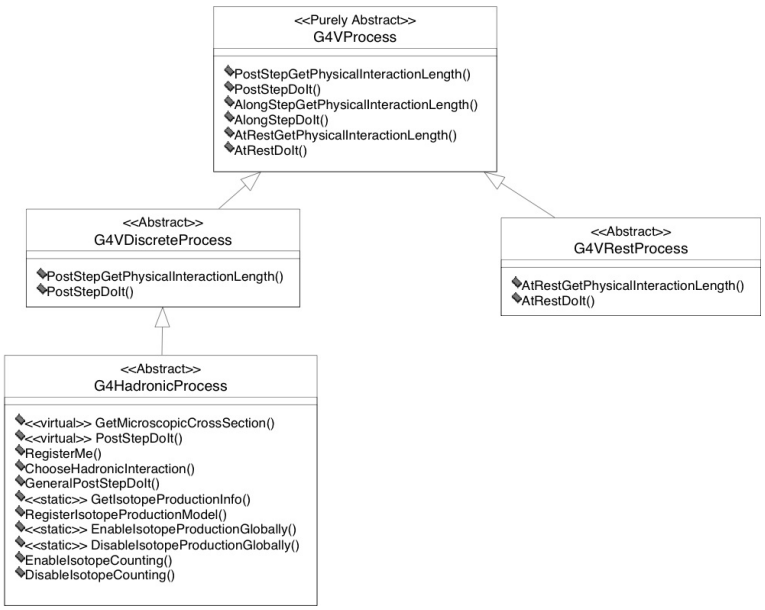

Figure 2.9. First level implementation framework of the hadronic category of Geant4.

Whenever possible, it is preferable to add any new hadronic physics to Geant4 in terms of a model, and assign the model to an existing process, rather than develop a new, specific process. However, in some cases, a directly implemented hadronic process may be necessary. In these cases, the new process must derive from `G4HadronicProcess` and the following three methods must be implemented:

```
virtual G4VParticleChange* PostStepDoIt(const G4Track&, const G4Step&) ,
virtual G4bool IsApplicable(const G4ParticleDefinition&) , and
G4double GetMeanFreePath(const G4Track& aTrack, G4double, G4ForceCondition*) .
```

More details on these methods will be provided in the next Chapter.

At the next level of abstraction, only processes that occur for particles in flight are considered. For these, the main design requirement is to treat cross sections and the final-state models (i.e. the models responsible for the production of the secondaries) independently, so that it is possible to change cross section while keeping a particular final-state model, or, vice versa, to keep a given cross section while replacing the final-state model. Moroever, a set of cross sections can be used for a single hadronic process to cover a wide kinematical range (e.g. from thermal energies up to several tera-electronvolts of the projectile kinetic energy). Similarly, for the same reason, a set of different final-state models can be used for a single hadronic process, allowing the overlapping between two models in an interval of the projectile kinetic energy, to insure a smooth transition between these models.

The class diagram for hadronic cross-sections is shown in Figure 2.10 and in Figure 2.11 for final-state models.

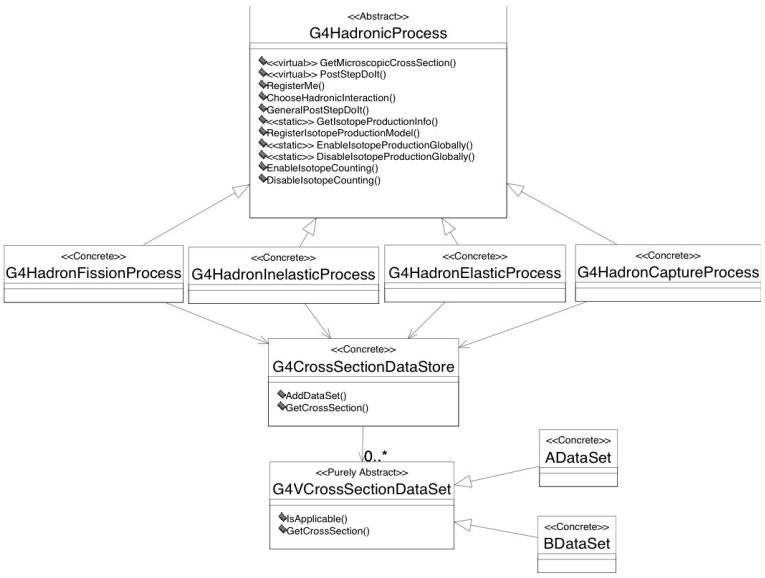

Figure 2.10. Second level implementation framework of the hadronic category of Geant4: cross-section aspect.

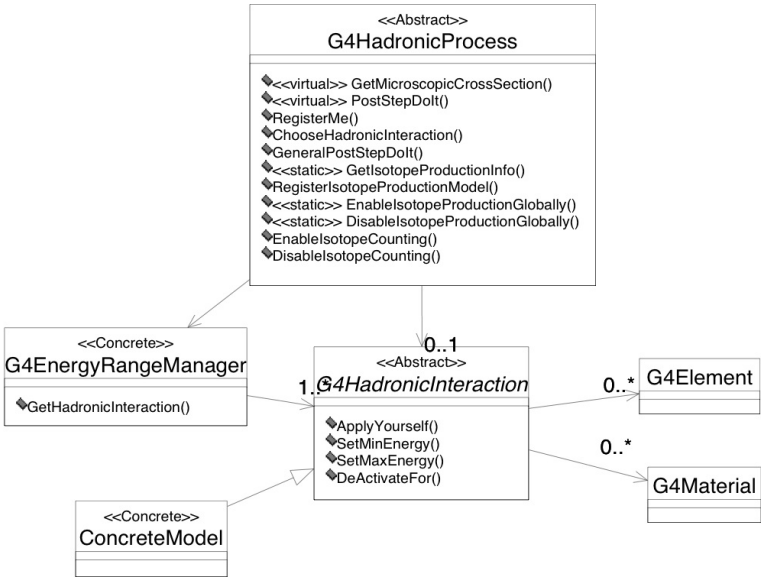

Figure 2.11. Second level implementation framework of the hadronic category of Geant4: final-state production aspect.

Each hadronic process is derived from `G4HadronicProcess`, which holds a list of ``cross section data sets''. All cross section data set classes are derived from the abstract class `G4VCrossSectionDataSet`. The process stores and retrieves the data sets through a data store that acts like a FILO (First-In-Last-Out) stack. Details on how to write a new hadronic cross section set will be provided in the next Chapter.

The `G4HadronicProcess` class provides a registration service for classes deriving from `G4HadronicInteraction`, and delegates final-state production to the applicable model. Models inheriting from `G4HadronicInteraction` can be restricted in applicability in projectile type and energy, and can be activated/deactivated for individual materials and elements. Details on how to write a new hadronic final-state model will be provided in the next Chapter.

2.6. Hits and Digitization

2.6.1. Design Philosophy

In Geant4 a *hit* is a snapshot of a physical interaction or an accumulation of interactions of a track or tracks in a ``sensitive'' detector component. A digitization, or *digit*, represents a detector output, such as an ADC/TDC count or a trigger signal. A *digit* is created from one or more hits and/or other *digits*.

Given the wide variety of Geant4 applications, ways of describing detector sensitivity and the quantities to be stored in the *hits* and *digits* vary greatly. This category therefore provides only abstract classes for both detector sensitivity and *hits*/*digits*. It also provides tools for organizing the *hits*/*digits* into collections.

2.6.2. Class Design

- **G4VHit** - this class has all the information about a particular hit caused by a single step.
- **G4VHitsCollection** - base class for a collection of hits.
- **G4THitsCollection** - template class for a collection of hits of the (template) type. Implements **G4VHitsCollection** interface providing efficient storage of hits via allocators.
- **G4HCofThisEvent** - container class for collections of hits for the current event.
- **G4VSensitiveDetector** - pure virtual class representing a sensitive element responsible for creating and managing associated hits. The user should implement the method ProcessHits a filter and readout geometry (optional) are also allowed.
- **G4SDManager** - singleton managing sensitive detectors.
- **G4SDMessengerr** - SD manager associated messenger.
- **G4SDStructure** - used exclusively used by G4SDManager for handling the tree structure of the user's sensitive detector names. Each branch represents the hits in given sub-detector. For example, the first level of branches may consist of a tracker, ECAL, and HCAL, while the second level, in HCAL, consists of the barrel and endcaps. Finally the barrel may have phi-slices, Z-slices, etc.

For digitization features a similar design as for hits is applied:

- **G4VDigi** - an abstract (base) class for all G4 digitizations. This could be data as simple as a singe byte, or as complex as an Ntuple.
- **G4VDigiCollection** - base class for a collection of digits.
- **G4TDigiCollection** - template class for a collection of digits of the (template) type. Implements **G4VDigiCollection** interface providing efficient storage of digits via allocators.
- **G4DCofThisEvent** - container class for collections of digits for the current event.
- **G4VDigitizerModule** - the class of objects which transform the hits deposited by particles into digitizations.
- **G4DigiManager** - singleton managing digitiser modules.
- **G4DigiMessengerr** - Digi manager associated messenger.

The UML class diagram for sensitivity related classes is shown in the following class diagram. Figure 2.12 shows the general management of hit classes.

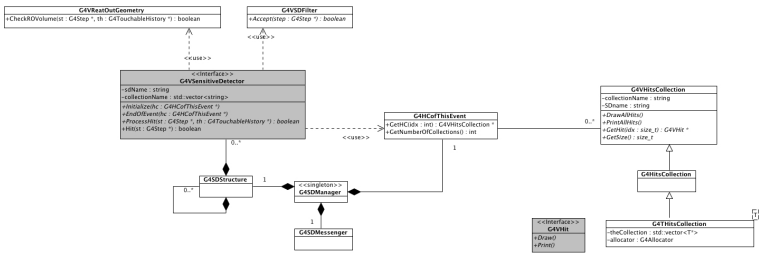

Figure 2.12. Overview of hit classes management. Classes in grey represent the main components that a user must subclass to implement a sensitive detector. User is also responsible of creating the binding between G4THitsCollection and its hit class.

2.7. Geometry

2.7.1. Design Philosopy

The geometry category provides the ability to describe a geometrical structure and propagate particles efficiently through it. This is done in part with the aid of two central concepts, the *logical* and *physical* volumes. A logical volume represents a detector element of a given shape which may contain other volumes, and which may have other attributes. It has access to other information which is independent of its phyisical location in the detector, such as material and sensitive detector behavior. A physical volume represents the spatial positioning or placement of the logical volume with respect to an enclosing mother (logical) volume. Thus a hierarchical tree structure of volumes can be built with each volume containing smaller volumes (which may not overlap). Repetitive structures can be represented by specialized physical volumes, such as replicas and parameterized placements, sometimes resulting in a large savings in memory.

In Geant4 the logical volume has been refined by defining the shape as a separate entity, called a *solid*. Solids with simple shapes, like rectilinear boxes, trapezoids, spherical or cylindrical sections or shells, each have their properties coded separately, in accord with the concept of *Constructed Solid Geometry (CSG)*. More complex solids are defined for specific use, or having their surfaces approximated by facets (tessellated solids).

Another way to build solids is by Boolean combination - union, intersection and subtraction. The elemental solids should be CSGs.

Although a detector is naturally and best described as by a hierarchy of volumes, efficiency is not critically dependent on this. An optimization technique, called voxelization, allows efficient navigation even in ``flat'' geometries, typical of those produced by CAD systems.

2.7.2. Class Design

- **G4GeometryManager** - responsible for managing ``high level'' objects in the geometry subdomain, notably including opening and closing (``locking'') the geometry, and creating/deleting optimization information for G4Navigator. The class is a "singleton".
- **G4LogicalVolumeStore** - a container for optionally storing created logical volumes. It enables traversal of all logical volumes by the UI/user/etc.
- **G4LogicalVolume** - represents a leaf node or unpositioned subtree in the geometry hierarchy. It may have daughters ascribed to it, and is also responsible for retrieval of the physical and tracking attributes of the physical volume that it represents. These attributes include solid, material, magnetic field, and optionally user limits, sensitive detectors, etc. Logical volumes are optionally entered into the G4LogicalVolumeStore.
- **G4MagneticField** - a class responsible for the magnetic field in each volume, including the calculation of particle trajectories along curved paths. In cases where the geometry step limits the particle's step, the distance calculated is guaranteed to be the distance to a volume boundary.
- **G4Navigator** - a class used by the tracking management, able to obtain/calculate tracking-time geometrical information such as distance to the next volume, or to find the physical volume containing a given point in the world reference system. The navigator maintains a transformation history and other information used to optimize the tracking time performance.
- **G4NavigationHistory** - responsible for maintenance of the history of the path taken through the geometrical hierarchy. It is principally a utility class for use by G4Navigator.
- **G4NormalNavigation** - a utility class for navigation in volumes containing only G4PVPlacement daughter volumes.
- **G4ParameterisedNavigation** - a utility class for navigation in volumes containing a single G4PVParameterised volume for which voxels for the replicated volumes have been constructed.
- **G4VoxelNavigation** - a utility class for navigation in volumes containing only G4PVPlacement daughter volumes for which voxels have been constructed.
- **G4ReplicaNavigation** - a utility class for navigation in volumes containing a single G4PVParameterised volume for which voxels for the replicated volumes have been constructed.
- **G4PhysicalVolumeStore** - a container for optionally storing created physical volumes. It enables traversal of all physical volumes by the UI/user/etc. All solids should be registered with G4PhysicalVolumeStore, and removed on their destruction. It is intended principally for the UI browser.

- **G4VPhysicalVolume** - a volume positioned within and relative to a given mother volume, and also represented by a given logical volume. They are optionally entered into the G4PhysicalVolumeStore.
- **G4PVPlacement** - a physical volume corresponding to a single touchable detector element, positioned within and relative to a mother volume.
- **G4PVReplica** - a physical volume representing many identically formed touchable detector elements, differing only in their positioning. The elements' positions are determined by means of a simple formula, and the elements completely fill the containing mother volume.
- **G4PVParameterised** - a physical volume representing many touchable detector elements differing in their positioning and dimensions. Both are calculated by means of a G4VParameterisation object. Each element's position is calculated as per G4PVReplica, and each element's shape can be modified by means of a user supplied formula.
- **G4VPVParameterisation** - a parameterisation class able to compute the transformation and, indirectly, the dimensions of parameterised volumes, given a replication number.
- **G4SmartVoxelProxy** - a class for proxying smart voxels. The class represents either a header (in turn refering to more VoxelProxies) or a node. If created as a node, calls to GetHeader cause an exception, and likewise GetNode when a header.
- **G4SmartVoxelHeader** - represents a single axis of virtual division. Contains the individual divisions which are potentially further divided along different axes.
- **G4SmartVoxelNode** - a single virtual division, containing the physical volumes inside its boundaries and those of its parents.
- **G4VoxelLimits** - represents limitation/restrictions of space, where restrictions are only made perpendicular to the cartesian axes.
- **G4SolidStore** - a container for optionally storing created solids. It enables traversal of all/any solids by the UI/user/etc. The class is a "singleton".
- **G4VSolid** - position independent geometrical entities. They have only `shape', and encompass both CSG and boundary representations. They are optionally entered into the G4SolidStore. This class defines, but does not implement, functions to compute distances to/from the shape. Functions are also defined to check whether a point is inside the shape, to return the surface normal of the shape at a given point, and to compute the extent of the shape.
- **G4VTouchable** - a class that maintains a ``reference'' on a given touchable element of the detector - a kind of bookmark. It enables a given detector element to be saved during tracking (in case of booleans/user code/etc.) and the corresponding G4PhysicalVolume retrieved later, with its ``state'' information (path through the tree) optionally restored so that navigation can be restarted. G4Touchables provide fast access to the transformation from the global reference system to that of the saved detector element.
- **G4TouchableHistory** - object representing a touchable detector element, and its history in the geomtrical hierarchy, including its net resultant local->global transform.
- **G4GRSSolid**} - object representing a touchable solid. It maintains the association between a solid and its net resultant local-to-global transform.
- **G4GRSVolume** - object representing a touchable detector element. It maintains associations between a physical volume and its net resultant local-to-global transform.
- **G4AffineTransform** - a class for geometric affine transformations. It supports efficient arbitrary rotation and transformation of vectors and the computation of compound and inverse transformations. A ``rotation flag'' is maintained internally for greater computational efficiency for transforms that do not involve rotation.
- **G4UserLimits** - responsible for user limits on step size, ascribable to individual volumes.

Figure 2.13 shows a general overview, in UML notation, of the geometry design. A detailed collection of class diagrams from the geometry category is found in the Appendix.

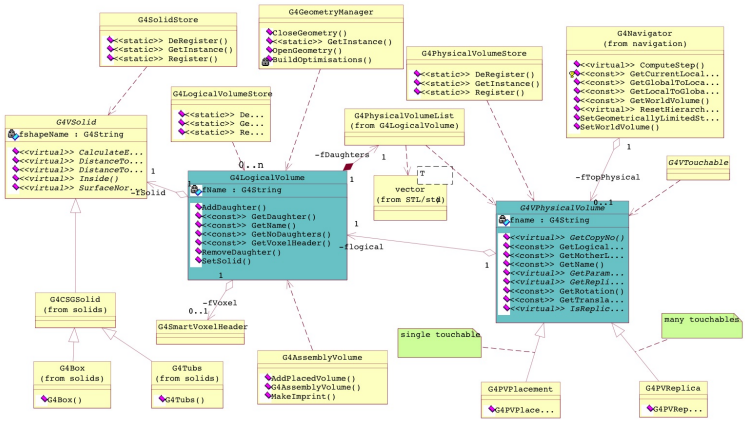

Figure 2.13. Overview of the geometry

2.7.3. Additional Geometry Diagrams

Additional diagrams for the object-oriented design of the 'geometry' related classes are included here. Figure 2.14 shows the class diagram for smart voxels. Figure 2.15 shows the class diagram for the navigator.

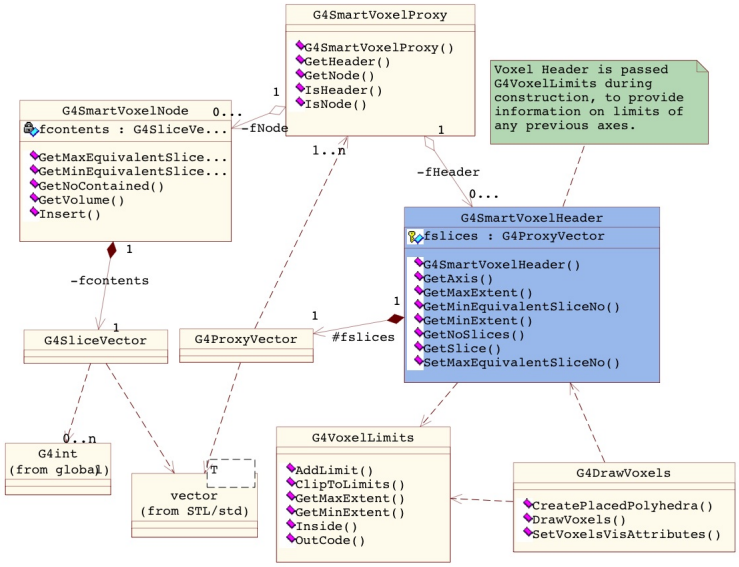

Figure 2.14. Class diagram for smart voxels

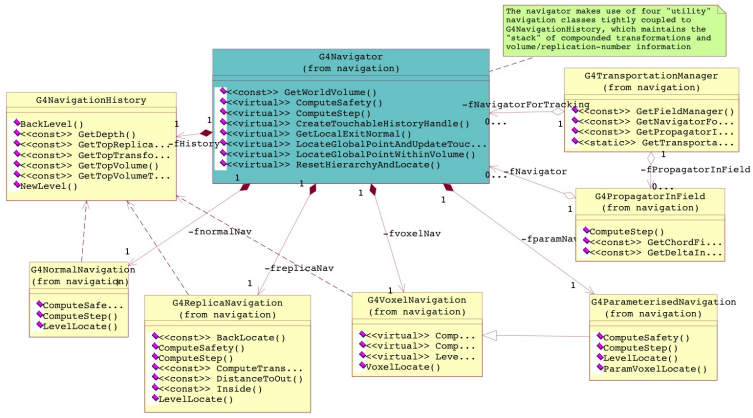

Figure 2.15. Class diagram for the navigator

2.8. Electromagnetic Fields

2.8.1. Class Design

- **G4TransportationManager** - singleton class storing the (volume) navigator used by the transportation process to do the geometrical tracking. It also stores a pointer to the propagator used in a (magnetic) field and to the field manager. The class instance is created before main() is called, and in turn creates the navigator and the rest.
- **G4PropagatorInField** - class performing the navigation/propagation of a particle/track in a magnetic field. The field is in general non-uniform. For the calculation of the path, it relies on the class G4ChordFinder.
- **G4FieldManager** - class managing (storing) a pointer to the Field subclass that describes the field of a detector (magnetic, electric or other). Also stores a reference to the chord finder. The G4FieldManager class exists to allow the user program to specify the electric, magnetic and/or other field(s) of the detector. A field manager can be set to a logical volume (or to more than one), in order to vary its field from that of the world. In this manner a zero or constant field can override a global field, a more or less exact version can override the external approximation, lower or higher precision for tracking can be specified, a different stepper can be chosen for different volumes, ... It also stores a pointer to the G4ChordFinder object that can do the propagation in this field. G4FieldManager allows the other classes/object (of the MagneticField and other class categories) to find out whether a detector field object exists and what that object is. A default G4FieldManager is created by the singleton class G4NavigatorForTracking and exists before main is called. However a new one can be created and given to G4NavigatorForTracking. Our current design envisions that one G4FieldManager is valid for each region detector.
- **G4ChordFinder** - class providing the integration of motion using the ODE solver's driver to find the end-point that satisfies the chord length criterion. It also returns an Approximate point on the curve near to a (chord) point. The G4ChordFinder handles all geometrical track "advancements" in the field. The G4ChordFinder must be created either by calling CreateChordFinder() for a Magnetic Field from G4FieldManageror by the user creating a G4ChordFinder object "manually" and setting the pointer.

The object-oriented design of the classes related to the electromagnetic field is shown in the class diagram of Figure 2.16. The diagram is described in UML notation.

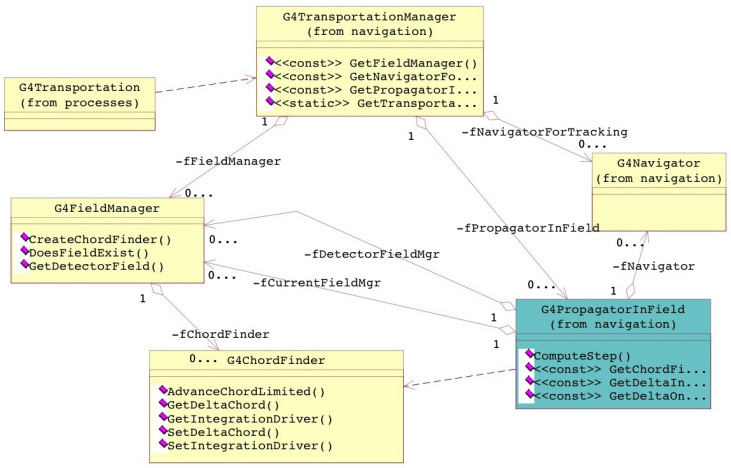

Figure 2.16. Electromagnetic Field

2.9. Particles

2.9.1. Design Philosophy

The particles category implements the facilities necessary to describe the physical properties of particles for the simulation of particle-matter interactions. All particles are based on the G4ParticleDefinition class, which de-

scribes basic properties such as mass, charge, etc., and also allows the particle to carry a list of processes to which it is sensitive. A first-level extension of this class defines the interface for particles that carry cuts information, for example range cut versus energy cut equivalence. A set of virtual, intermediate classes for leptons, bosons, mesons, baryons, etc., allows the implementation of concrete particle classes which define the actual particle properties and, in particular, implement the actual range versus energy cuts equivalence. All concrete particle classes are instantiated as singletons to ensure that all physics processes refer to the same particle properties.

2.9.2. Class Design

The object-oriented design of the 'particles' related classes is shown in the following class diagrams. The diagrams are described in the Booch notation. Figure 2.17 shows a general overview of the particle classes. Figure 2.19 shows classes related to the particle table and Figure 2.18 shows the classes related to the ion table in the particle table. Figure 2.20 shows the classes related to the particle decay table in the G4ParticleDefinition. This decay tabele is used by decay process and Figure 2.21 shows the classes related to the decay process and decay channels

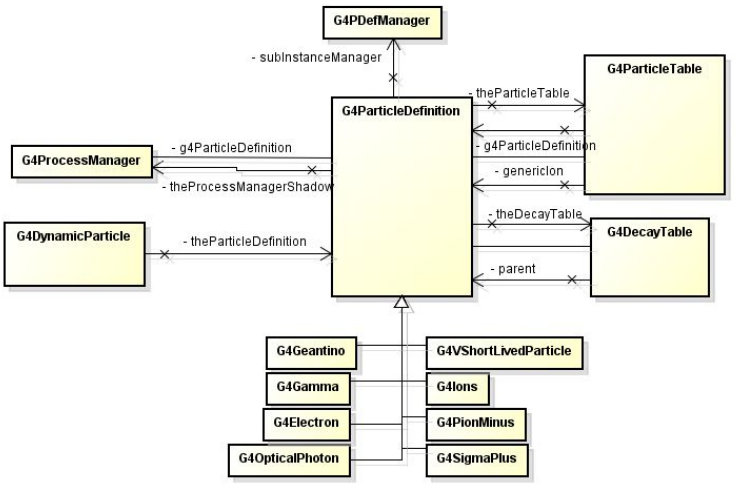

Figure 2.17. Particle classes

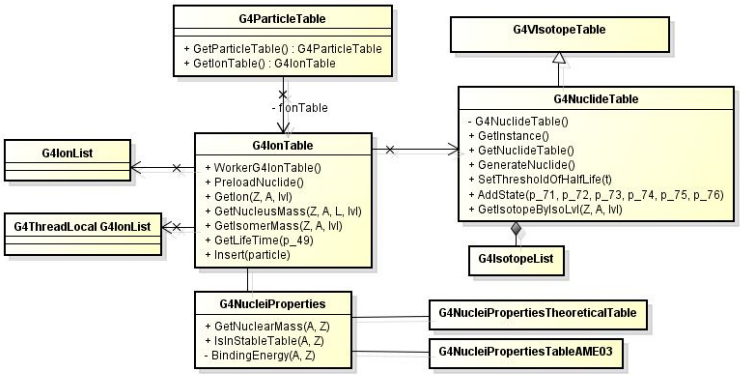

Figure 2.18. Particle Table

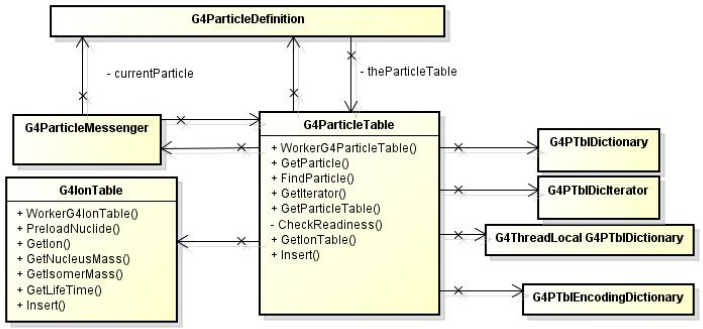

Figure 2.19. Particle Table

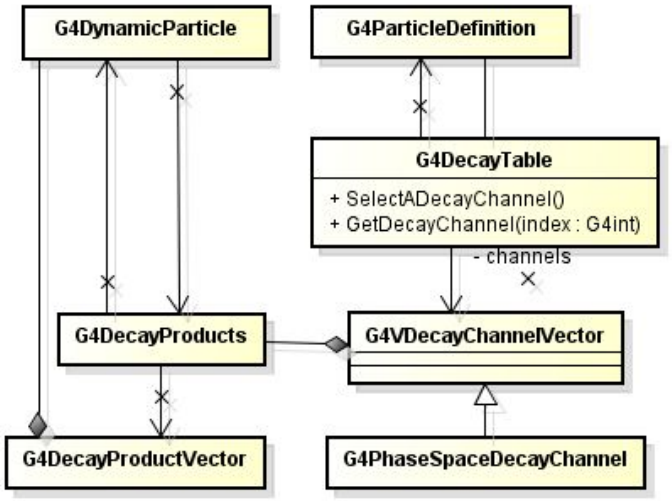

Figure 2.20. Decay Table

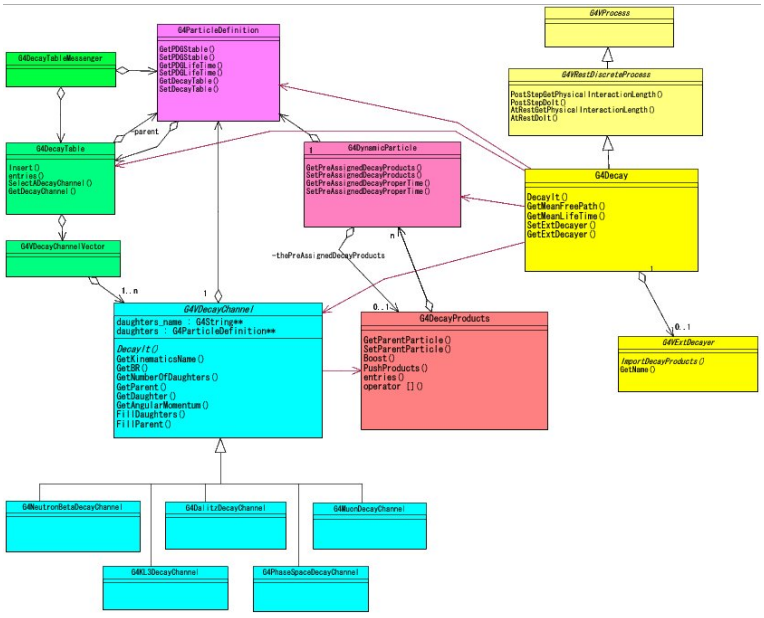

Figure 2.21. Particle Decay

2.10. Materials

2.10.1. Design Philosophy

The design of the materials category reflects what exists in nature: materials are made of a single element or a mixture of elements, and elements are made of a single isotope or a mixture of isotopes. Because the physical properties of materials can be described in a generic way by quantities which can be specified directly, such as density, or derived from the element composition, only concrete classes are necessary in this category.

The material category implements the facilities necessary to describe the physical properties of materials for the simulation of particle-matter interactions. Characteristics like radiation and interaction length, excitation energy loss, coefficients in the Bethe-Bloch formula, shell correction factors, etc., are computed from the element, and if necessary, the isotope composition.

The material category also implements facilities to describe surface properties used in the tracking of optical photons.

2.10.2. Class Design

The object-oriented design of the 'materials' related classes is shown in the class diagram: Figure 2.22. The diagram is described in the Booch notation.

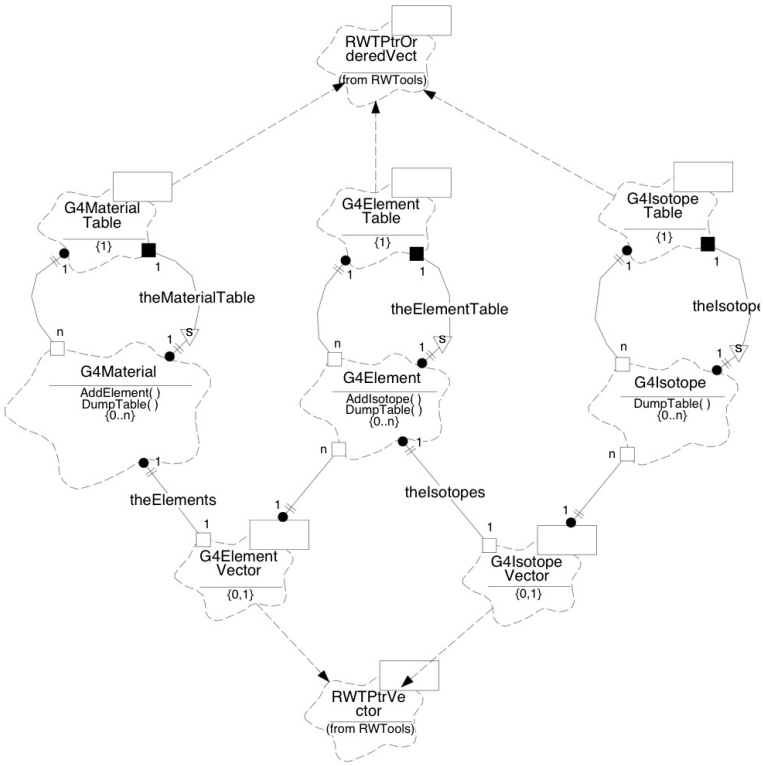

Figure 2.22.

2.11. Global Usage

2.11.1. Design Philosophy

The global category covers the system of units, constants, numerics and random number handling. It can be considered a place-holder for "general purpose" classes used by all categories defined in Geant4. No back-dependencies to other Geant4 categories affect the "global" domain. There are direct dependencies of the global category on external packages, such as CLHEP, STL, and miscellaneous system utilities.

Within the management sub-category are ``utility'' classes generally used within the Geant4 kernel. They are, for the most part, uncorrelated with one another and include:

- *G4Allocator*
- *G4FastVector*
- *G4ReferenceCountedHandle*
- *G4PhysicsVector, G4LPhysicsFreeVector, G4PhysicsOrderedFreeVector*
- *G4Timer*
- *G4UserLimits*
- *G4UnitsTable*

A general description of these classes is given in section 3.2 of the Geant4 User's Guide for Application Developers.

The module includes wrappers to most CLHEP classes used in Geant4, and tools for memory management (`G4Cache, G4AutoDelete`) and for threading (`G4AutoLock, G4Threading, G4ThreadLocalSingleton, G4TWorkspacePool`). It also provides specialised fast implementations for some heavily used mathematical functions, like `G4Exp, G4Log, G4Pow`.

In applications where it is necessary to generate random numbers (normally from the same engine) in many different methods and parts of the program, it is highly desirable not to rely on or require knowledge of the global objects instantiated. By using static methods via a unique generator, the randomness of a sequence of numbers is best assured. Hence the use of a static generator has been introduced in the original design of HEPRandom as a project requirement in Geant4.

2.11.2. Class Design

Analysis and design of the HEPRandom module have been achieved following the Booch Object-Oriented methodology. Some of the original design diagrams in Booch notation are reported below. Figure 2.23 is a general picture of the static class diagram.

- **HepRandomEngine** - abstract class defining the interface for each Random engine. Its pure virtual methods must be defined by its subclasses representing the concrete Random engines.
- **HepJamesRandom** - class inheriting from HepRandomEngine and defining a flat random number generator according to the algorithm described in "F.James, Comp.Phys.Comm. 60 (1990) 329". This class is instantiated by default as the default random engine.
- **DRand48Engine** - class inheriting from HepRandomEngine and defining a flat random number generator according to the drand48() and srand48() system functions from the C standard library.
- **RandEngine** - class inheriting from HepRandomEngine and defining a flat random number generator according to the rand() and srand() system functions from the C standard library.
- **RanluxEngine** - class inheriting from HepRandomEngine and defining a flat random number generator according to the algorithm described in "F.James, Comp.Phys.Comm. 60 (1990) 329-344" and originally implemented in FORTRAN 77 as part of the MATHLIB HEP library. It provides 5 different "luxury" levels [0..4].
- **RanecuEngine** - class inheriting from HepRandomEngine and defining a flat random number generator according to the algorithm RANECU originally written in FORTRAN 77 as part of the MATHLIB HEP library. It uses a table of seeds which provides uncorrelated couples of seed values.
- **MixMaxRng** - class inheriting from HepRandomEngine and interfacing the MixMax Matrix PseudoRandom Number Generator described in J.Comput.Phys. 97, 573 (1991).
- **HepRandom** - the main class collecting all the methods defining the different random generators applied to HepRandomEngine. It is a singleton class which all the distribution classes derive from. This singleton is instantiated by default.
- **RandFlat** - distribution class for flat random number generation. It also provides methods to fill an array of flat random values, given its size or shoot bits.
- **RandExponential** - distribution class defining exponential random number distribution, given a mean. It also provides a method to fill an array of flat random values, given its size.
- **RandGauss** - distribution class defining Gauss random number distribution, given a mean or specifying also a deviation. It also provides a method to fill an array of flat random values, given its size.

- **RandBreitWigner** - distribution class defining the Breit-Wigner random number distribution. It also provides a method to fill an array of flat random values, given its size.
- **RandPoisson** - distribution class defining Poisson random number distribution, given a mean. It also provides a method to fill an array of flat random values, given its size.

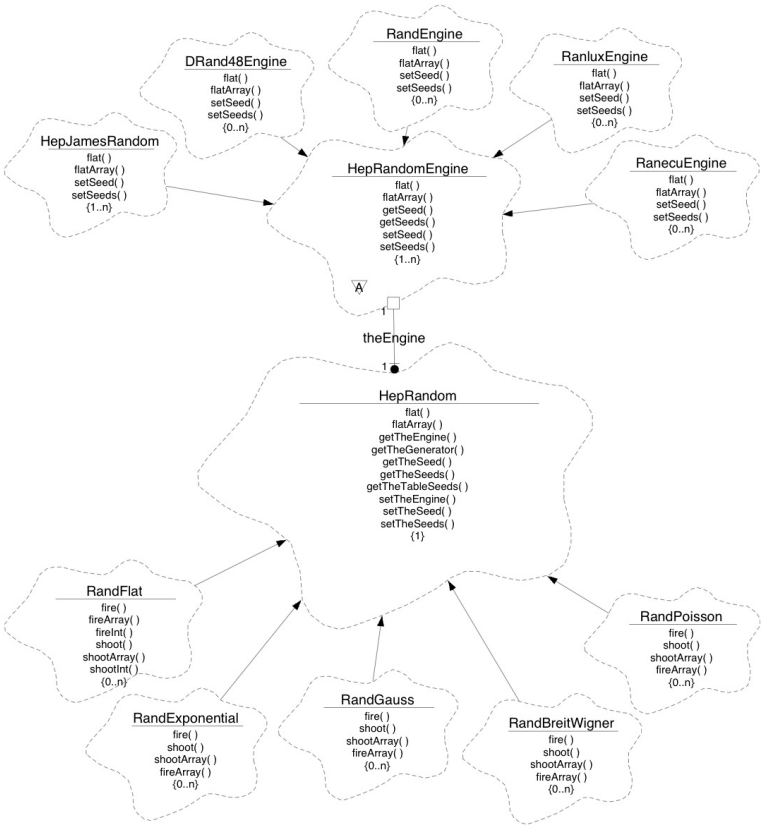

Figure 2.23. HEPRandom module

For detailed documentation about the HEPRandom classes see the CLHEP Reference Guide or the CLHEP User Manual.

Informations written in this manual are extracted from the original manifesto distributed with the HEPRandom package.

HEPNumerics

The HEPNumerics module includes a set of classes which implement numerical algorithms for general use in Geant4. The User's Guide for Application Developers contains a description of each class. Most of the algorithms were implemented using methods from the following books:

- B.H. Flowers, "An introduction to Numerical Methods In C++", Claredon Press, Oxford 1995.
- M. Abramowitz, I. Stegun, "Handbook of mathematical functions", DOVER Publications INC, New York 1965 ; chapters 9, 10, and 22.

The HEPNUmerics module provides general mathematical methods supporting Geant4 Monte-Carlo simulation processes. Among these, there are methods for function and array interpolations using known special functions, class method integration solving polynomial equation (up to 4th order) and some others.

Of particular interest is the templated class `G4Integrator` which consists of methods allowing to integrate class methods. Since the type whose method should be integrated is not known in advance, `G4Integrator` uses templated signatures and pointers to functions in its API. It provides both usual numerical methods like adaptive Gauss or Simpson integration, and more sofisticated (faster and at the same accurate) methods based on the orthogonal polynomials.

Among the different integration methods involving orthogonal polynomials there are: Gauss-Legendre, Gauss-Chebyshev, Gauss-Hermite and Gauss-Jacobi methods:

```
template <class T, class F>
G4double G4Integrator<T,F>::Legendre( T& typeT, F f, G4double a,
                                      G4double b, G4int nLegendre )
//
// The value nLegendre set the accuracy required, i.e the number of points
// where the function pFunction will be evaluated during integration.
// The function creates the arrays for abscissas and weights that used
// in Gauss-Legendre quadrature method.
// The values a and b are the limits of integration of the function  f.
// nLegendre MUST BE EVEN !!!
// Returns the integral of the function f between a and b, by 2*fNumber point
// Gauss-Legendre integration: the function is evaluated exactly
// 2*fNumber times at interior points in the range of integration.
// Since the weights and abscissas are, in this case, symmetric around
// the midpoint of the range of integration, there are actually only
// fNumber distinct values of each.
// Convenient for using with some class object dataT

template <class T, class F>
G4double G4Integrator<T,F>::Legendre10( T& typeT, F f,G4double a,
                                      G4double b )
//
// Returns the integral of the function to be pointed by T::f between a and b,
// by ten point Gauss-Legendre integration: the function is evaluated exactly
// ten times at interior points in the range of integration. Since the weights
// and abscissas are, in this case, symmetric around the midpoint of the
// range of integration, there are actually only five distinct values of each
// Convenient for using with class object typeT. The method is very fast and accurate enough.
// Roots and weights are from Abramowitz M., Stegan I.A. 1964 , Handbook of Math... , p. 916

template <class T, class F>
G4double G4Integrator<T,F>::Legendre96( T& typeT, F f,G4double a,
                                      G4double b )
//
// Returns the integral of the function to be pointed by T::f between a and b,
// by 96 point Gauss-Legendre integration: the function is evaluated exactly
// ten Times at interior points in the range of integration. Since the weights
// and abscissas are, in this case, symmetric around the midpoint of the
// range of integration, there are actually only five distinct values of each
// Convenient for using with some class object typeT. The method is very accurate and fast enough.
// Roots and weights are from Abramowitz M., Stegan I.A. 1964 , Handbook of Math... , p. 919

template <class T, class F>
G4double G4Integrator<T,F>::Chebyshev( T& typeT, F f, G4double a,
                                      G4double b, G4int nChebyshev )
//
// Integrates function pointed by T::f from a to b by Gauss-Chebyshev
// quadrature method.
// Convenient for using with class object typeT

template <class T, class F>
G4double G4Integrator<T,F>::Laguerre( T& typeT, F f, G4double alpha,
                                      G4int nLaguerre )
//
// Integral from zero to infinity of std::pow(x,alpha)*std::exp(-x)*f(x).
// The value of nLaguerre sets the accuracy.
// The function creates arrays fAbscissa[0,..,nLaguerre-1] and
// fWeight[0,..,nLaguerre-1] .
// Convenient for using with class object 'typeT' and (typeT.*f) function
// (T::f)

template <class T, class F>
G4double G4Integrator<T,F>::Hermite( T& typeT, F f, G4int nHermite )
//
// Gauss-Hermite method for integration of std::exp(-x*x)*f(x)
// from minus infinity to plus infinity.

template <class T, class F>
G4double G4Integrator<T,F>::Jacobi( T& typeT, F f, G4double alpha,
                                      G4double beta, G4int nJacobi )
//
// Gauss-Jacobi method for integration of ((1-x)^alpha)*((1+x)^beta)*f(x)
// from minus unit to plus unit (-1,+1).
```

HEPGeometry

Documentation for the HEPGeometry module is provided in the CLHEP Reference Guide or the CLHEP User Manual.

2.12. Visualisation

2.12.1. Design Philosophy

The visualisation category consists of the classes required to display detector geometry, particle trajectories, tracking steps, and hits. It also provides visualisation drivers, which are interfaces to external graphics systems.

A wide variety of user requirements went into the design of the visualisation category, for example:

- very quick response in surveying successive events,
- high-quality output for presentation and documentation,
- flexible camera control for debugging detector geometry and physics,
- selection of visualisable objects,
- interactive picking of graphical objects for attribute editing or feedback to the associated data,
- highlighting incorrect intersections of physical volumes,
- co-working with graphical user interfaces.

Because it is very difficult to respond to all of these requirements with only one built-in visualiser, an abstract interface was developed which supports several complementary graphics systems. Here the term **graphics system** means either an application running as a process independent of Geant4 or a graphics library to be compiled with Geant4. A concrete implementation of the interface is called a **visualisation driver**, which can use a graphics library directly, communicate with an independent process via pipe or socket, or simply write an intermediate file for a separate viewer.

2.12.2. The Graphics Interfaces

- G4VVisManager: All user code writes to the graphics systems through this pure abstract interface. It contains Draw methods for all the graphics primitives in the graphics_reps category (G4Polyline, G4Circle, etc.), geometry objects (through their base classes, G4VSolid, G4PhysicalVolume and G4LogicalVolume) and hits and trajectories (through their base classes, G4VHit and G4VTrajectory).

  Since this is an abstract interface, all user code must check that there exists a concrete instantiation of it. A static method is provided, so a typical user code fragment is:

```
G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
if(pVVisManager) {
  pVVisManager->Draw(G4Circle...
  ...
```

  Note that this allows the building an application without a concrete implementation, for example for a batch job, even if some code, like the above, is still included. Most of the novice examples can be built this way if G4VIS_NONE is specified.

  The concrete implementation of this interface is hereafter referred to as the **visualisation manager**.
- **G4VGraphicsScene**: The visualisation manager must also provide a concrete implementation of the subsidiary interface, G4VGraphicsScene. It is only for use by the kernel and the modeling category. It offers direct access to a ``scene handler'' through a reference provided by the visualisation manager. It is described in more detail in the section on extending the toolkit functionality.

The Geant4 distribution includes implementations of the above interfaces, namely **G4VisManager** and **G4VSceneHandler** respectively, and their associated classes. These define further abstract base classes for visu-

alisation drivers. Together they form the **Geant4 Visualisation System**. A variety of concrete visualisation drivers are also included in the distribution. Details of how to implement a visualisation driver are given in Section 3.7. Of course, it is always possible for a user to implement his or her own concrete implementations of G4VVisManager and G4VGraphicsScene replacing the Geant4 Visualisation System altogether.

2.12.3. The Geant4 Visualisation System

The Geant4 Visualisation System consists of

- **G4VisManager**: An implementation of the G4VVisManager interface. It manages multiple graphics systems and defines three more concepts -- the **scene** (G4Scene), the **scene handler** (base class G4VSceneHandler, itself a sub-class of G4VGraphicsScene) and the **viewer** (base class G4VViewer) -- see below. G4VisManager is a singleton and an abstract class, requiring the user to derive from it a concrete visualisation manager (G4VisExecutive is provided -- see below). Roles and structure of the visualisation manager are described in Chapter 8 of the User's Guide for Application Developers.
- **G4VisExecutive**: A concrete visualisation manager that implements the virtual functions RegisterGraphicsSystems and RegisterModelFactories. These functions must be in the users' domain, since the graphics systems and models that are instantiated by them are, in many cases, provided by the user (graphics libraries, etc.). It is therefore implemented as a .hh-.icc combination that is designed to be included in the users' code. Of course, the user may write his or her own.
- **G4Scene** The scene is a list if models for physical volumes, axes, hits, trajectories, etc. - see Section Section 2.12.4. They are distinguished according to their lifetime -- ``run-duration'' for physical volumes, etc., ``end-of-event'' for hits and trajectories, etc. The end-of-event models are only to be used when the Geant4 state indicates the end of event has been reached. The scene has an **extent** (G4VisExtent), which is updated by the scene when a new model is added (each model itself has an extent), and a ``standard'' target point; these are used to define the standard view -- see below. In addition, the scene keeps flags which indicate whether end-of-event objects should be accumulated or refreshed for each event or run.
- **G4VGraphicsSystem**: This is an abstract base class for scene handler and viewer factories. It is used by the visualisation manager to create scene handlers and viewers on request.
- **G4VSceneHandler**: A sub-class of G4VGraphicsScene, itself an abstract base class for specific scene handlers, whose job is to convert the scene into graphics-system-specific code for the viewer. For example, the scene handler may create a graphical database, taking care to separate run-duration (persistent) and end-of-event (transient) information (this is described further in Section 3.7.1.6.
- **G4VViewer**: An abstract base class for specific viewers. Its job is to create windows or files and identify where and how the final view should be rendered. It has **view parameters** (G4ViewParameters) which specify viewpoint direction, type of rendering (wireframe or surface), etc. It is the view's responsibility, noting the scene's extent and target point, to choose a camera position and magnification that ensures that the scene is automatically and comfortably rendered in the viewing window. This is then the **standard view**, and any further operations requested by the user - zoom, pan, etc. - are relative to this standard view. The class G4ViewParameters has utility routines to assist this procedure; it is strongly advised that toolkit developers writing a viewer should study the G4ViewParameters class, whose header file contains much useful information (also preserved in the Software Reference Manual).

  The viewer is messaged by the vis manager when the user issues commands, such as `/vis/viewer/refresh`. This invokes methods such as SetView, ClearView and DrawView. A detailed description of the call sequences is given in Section 3.7.1.2- Section 3.7.1.5.

Note there is no restriction on the number or type of scene handlers or viewers. There may be several scene handlers processing the same or different scenes, each with several viewers (for example, the same scene from differing viewpoints).

By defining a set of three C++ classes inheriting from the virtual base classes - G4VGraphicsSystem, G4VSceneHandler and G4VViewer - an arbitrary graphics system can easily be plugged in to Geant4. The plugged-in graphics system is then available for visualising detector simulations. Together, this set of three concrete classes is called a "visualisation driver". The DAWN-File driver, for example, is the interface to the Fukui Renderer DAWN, and is implemented by the following set of classes:

1. G4DAWNFILE : public G4VGraphicsSystem for creation of the scene handlers and viewers

2. G4DAWNFILESceneHandler : public G4VSceneHandler for modeling 3D scenes
3. G4DAWNFILEView : public G4VView for rendering 3D scenes

Several visualisation drivers are distributed with Geant4. They are complementary to each other in many aspects. For details, see Chapter 8 of the User's Guide for Application Developers.

2.12.4. Modeling sub-category

- **G4VModel** - a base class for visualisation models. A model is a graphics-system-independent description of a Geant4 component.

  The sub-category visualisation/modeling defines how to model a 3D scene for visualisation. The term "3D scene" indicates a set of visualisable component objects put in a 3D world. A concrete class inheriting from the abstract base class G4VModel defines a "model", which describes how to visualise the corresponding component object belonging to a 3D scene. G4ModelingParameters defines various associated parameters.

  For example, G4PhysicalVolumeModel knows how to visualise a physical volume. It describes a physical volume and its daughters to any desired depth. G4HitsModel knows how to visualise hits. G4TrajectoriesModel knows how to visualise trajectories.

  The main task of a model is to describe itself to a 3D scene by giving a concrete implementation of the following virtual method of G4VModel:

  ```
  virtual void DescribeYourselfTo (G4VGraphicsScene&) = 0;
  ```

  The argument class G4VGraphicsScene is a minimal abstract interface of a 3D scene for the Geant4 kernel defined in the graphics_reps category. Since G4VSceneHandler and its concrete descendants inherit from G4VGraphicsScene, the method DescribeYourselfTo() can pass information of a 3D scene to a visualisation driver.

  It is easy for a toolkit developer of Geant4 to add a new kind of visualisable component object. It is done by implementing a new class inheriting from G4VModel.
- **G4VTrajectoryModel** - an abstract base class for trajectory drawing models.

  A trajectory model governs how an individual trajectory is drawn. Concrete models inheriting from G4VTrajectoryModel must implement two pure virtual functions:

  ```
  virtual void Draw(const G4VTrajectory&, G4int i_mode = 0) const = 0;
  virtual void Print(std::ostream& ostr) const = 0;
  ```

  See for example G4TrajectoryDrawByParticleID.
- **G4VModelFactory** - an abstract base class for factories creating models and associated messengers.

  It is not necessary to generate messengers for a trajectory model that will be constructed and configured directly in compiled code. If the user requires model creation and configuration features through interactive commands, however, there must be a mechanism to generate both models and their associated messengers. This is the role of G4VModelFactory. Concrete factories inheriting from G4VModelFactory are responsible for creating a concrete model and concrete messengers. To help ensure a type safe messenger to model interaction on the command line, the messengers should inherit from G4VModelCommand.

  Concrete factories must implement one pure virtual function:

  ```
  virtual ModelAndMessengers
  Create(const G4String& placement, const G4String& modelName) = 0;
  ```

  where placement indicates which directory space the commands should occupy. See for example G4TrajectoryDrawByParticleIDFactory.

2.12.5. View parameters

View parameters such as camera parameters, drawing styles (wireframe/surface etc) are held by G4ViewParameters. Each viewer holds a view parameters object which can be changed interactively and a default object (for use in the `/vis/viewer/reset` command).

If a toolkit developer of Geant4 wants to add entries of view parameters, he should add fields and methods to G4ViewParameters.

2.12.6. Visualisation Attributes

All drawable objects (should) have a method:

```
const G4VisAttributes* GetVisAttributes() const;
```

A drawable object might be:

- a "visible" (i.e., inheriting G4Visible), such as a polyhedron, polyline, circle, etc. (note that text is a slightly special case - see below) or
- a solid whose vis attributes are held in its logical volume.

2.12.6.1. Finding the applicable vis attributes

This is an issue for all scene handlers. The scene handler is where the colour, style, auxiliary edge visibility, marker size, etc., of individual drawable objects are needed.

2.12.6.1.1. Visibles

If the vis attributes pointer is zero, drivers should pick up the default vis attributes from the viewer:

```
const G4VisAttributes* pVisAtts = visible.GetVisAttributes();
if (!pVisAtts)
  pVisAtts = fpViewer->GetViewParameters().GetDefaultVisAttributes();
```

where visible denotes any visible object (polyhedron, circle, etc.).

There is a utility function G4VViewer::GetApplicableVisAttributes which does this, so an alternative is:

```
const G4VisAttributes* pVisAtts =
  fpViewer->GetApplicableVisAttributes(visible.GetVisAttributes());
```

Confusingly, there is a utility function G4VSceneHandler::GetColour which also does this, so if it's only colour you need, the following suffices:

```
const G4Colour& colour GetColour(visible);
```

but equally well:

```
const G4VisAttributes* pVisAtts =
  fpViewer->GetApplicableVisAttributes(visible.GetVisAttributes());
const G4Colour& colour pVisAtts->GetColour();
```

or even:

```
const G4VisAttributes* pVisAtts = visible.GetVisAttributes();
if (!pVisAtts)
```

```
  pVisAtts = fpViewer->GetViewParameters().GetDefaultVisAttributes();
const G4Colour& colour pVisAtts->GetColour();
```

2.12.6.1.2. Text

Text is a special case because it has its own default vis attributes:

```
const G4VisAttributes* pVisAtts = text.GetVisAttributes();
if (!pVisAtts)
  pVisAtts = fpViewer->GetViewParameters().GetDefaultTextVisAttributes();
const G4Colour& colour pVisAtts->GetColour();
```

and there is a utility function G4VSceneHandler::GetTextColour:

```
const G4Colour& colour GetTextColour(text);
```

2.12.6.1.3. Solids

For specific solids, the G4PhysicalVolumeModel that provides the solids also provides, via PreAddSolid, a pointer to its vis attributes. If the vis attribites pointer in the logical volume is zero, it provides a pointer to the default vis attributes in the model, which in turn is (currently) provided by the viewer's vis attributes (see G4VSceneHandler::CreateModelingParameters). So the vis attributes pointer is guaranteed to be pertinent.

If the concrete driver does not implement AddSolid for any particular solid, the base class converts it to primitives (usually a G4Polyhedron) and again, the vis attributes pointer is guaranteed.

2.12.6.1.4. Drawing style

The drawing style is normally determined by the view parameters but for individual drawable objects it may be overridden by the forced drawing style flags in the vis attributes. A utility function G4ViewParameters::DrawingStyle G4VSceneHandler::GetDrawingStyle is provided:

```
G4ViewParameters::DrawingStyle drawing_style = GetDrawingStyle(pVisAtts);
```

2.12.6.1.5. Auxiliary edges

Similarly, the visibility of auxiliary/soft edges is normally determined by the view parameters but may be overridden by the forced auxiliary edge visible flag in the vis attributes. Again, a utility function G4VSceneHandler::GetAuxEdgeVisible is provided:

```
G4bool isAuxEdgeVisible = GetAuxEdgeVisible (pVisAtts);
```

2.12.6.1.6. LineSegmentsPerCircle

Also, the precision of rendering curved edges in the polyhedral representation of volumes is normally determined by the view parameters but may be overridden by a forced attribute. A utility function that respects this, G4VSceneHandler::GetNoOfSides, is provided. For example:

```
G4Polyhedron::SetNumberOfRotationSteps (GetNoOfSides (pVisAttribs));
```

2.12.6.1.7. Marker size

These have nothing to do with vis attributes; they are an extra property of markers, i.e., objects that inherit G4VMarker (circles, squares, text, etc.). However, the algorithm for the actual size is quite complicated and a utility function G4VSceneHandler::GetMarkerSize is provided:

```
MarkerSizeType sizeType;
```

```
G4double size = GetMarkerSize (text, sizeType);
```

sizeType is world or screen, signifying that the size is in world coordinates or screen coordinates respectively.

2.13. Intercoms

2.13.1. Design Philosophy

The intercoms category implements an expandable command interpreter which is the key mechanism in Geant4 for realizing customizable and state-dependent user interactions with all categories without being perturbed by the dependencies among classes. The capturing of commands is handled by a C++ abstract class *G4UIsession*. Various concrete implementations of the command capturer are contained in the [user] interfaces category. Taking into account the rapid evolution of graphical user interface (GUI) technology and consequent dependence on external facilities, plural and extensible GUIs are offered.

Programmers need only know how to register the commands and parameters appropriate to their problem domain; no knowledge of GUI programming is required to allow an application to use them through one of the available GUIs.

The intercoms category also provides the virtual base classes

- G4VVisManager,
- G4VGraphicsScene, and
- G4VGlobalFastSimulationManager.

2.13.2. Class Design

- **G4UISession** -
- **G4UIBatch** -
- **G4UICommand** -
- **G4UIparameter** -
- **G4UImessenger** -
- **G4UIExecutive**: A concrete interface manager. It will register the UI selected by the environment variable set. It will take first by defaul the following order : G4UI_USE_QT, G4UI_USE_XM, G4UI_USE_WIN32, G4UI_USE_TCSH, Terminal

The object-oriented design of the 'user interface' related classes is shown in the class diagram Figure 2.24. The diagram is described in the Booch notation.

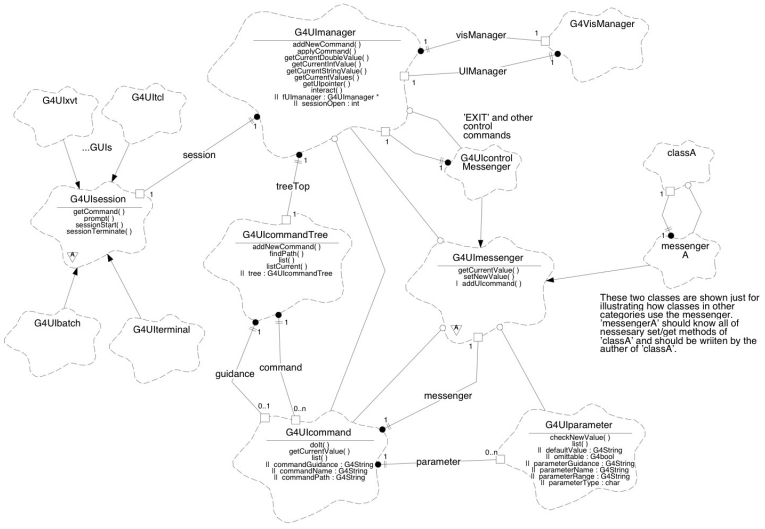

Figure 2.24. Overview of intercom classes

2.14. Parallelism in Geant4: multi-threading capabilities

2.14.1. Event level parallelism

Geant4 event-level parallelism is based on a *master-worker* model in which a set of threads (the *workers*) are spawned and are responsible for the simulation of events, while the steering and control of the simulation is given to an additional entity (the *master*).

Multi-threading functionalities are implemented with new classes or modifications of existing classes in the *run* category:

- The new run-manager class **G4MTRunManager** (that inherits from **G4RunManager**) implements the master model. It uses the mandatory class **G4MTRunManagerKernel**, a multi-threaded equivalent of **G4RunManagerKernel**
- The new run-manager class **G4WorkerRunManager** (that inherits from **G4RunManager**) implements the worker model. It uses the mandatory class **G4WorkerRunManagerKernel** the worker equivalent of **G4RunManagerKernel**
- The new user-initialization class **G4VUserActionInitialization** is responsible for the instantiation of thread-local user actions
- The new user-initialization class **G4UserWorkerInitialization** is responsible for the initialization of worker threads

Additional information on Geant4 multi-threading model can be found in the next section.

In this chapter, after a brief reminder of basic design choices, we will concentrate on aspects that are important for kernel developers, particularly the most critical aspects for multi-threading in Geant4: memory handling, split-classes and thread-local storage. In the following it is assumed that the user is already familiar with the general aspects of multi-threading. The section Additional Material provides more information on this topic.

2.14.2. General Design

Geant4 Version 10.0 introduces parallelism at the event level: events are tracked concurrently by independent threads. The parallelism model is master-worker in which one or more threads are responsible of performing the simulation, while a separate control flow controls and steers the work. A diagram of the general overview of a multi-threaded Geant4 application is shown here:

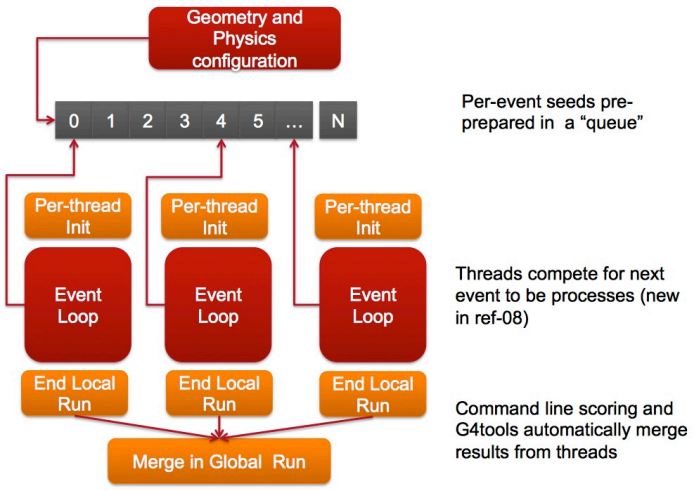

Figure 2.25. Simplified schema of the master-worker model employed in Geant4

The user interacts with the master which is responsible for creating and controlling worker threads. Before the simulation is started per-event seeds are generated by the master. This operation guarantees reproducibility. Once threads are spawned and configured, each worker is responsible for creating a new **G4Run** and for simulating a subset of the events. At the end of the run the results from each run are merged into the global run. Details on how to interact with a multi-threaded simulation are discussed in the *Guide for Application Developers*.

Geant4 parallelization makes use of the POSIX standard. The use of this standard in Geant4 guarantees maximum portability between systems and integration with advanced parallelization frameworks (for example we have verified that this model co-works with TBB and MPI).

To effectively reduce the memory consumption in a multi-threaded application, workers share instances of objects that consume the majority of memory (geometry and physics tables); workers own thread-private instances of the other classes (e.g. SensitiveDetectors, hits, etc). This choice allowed the design of a lock-free code (i.e. no use of mutex during the event loop), which guarantees maximum scalability (cfr: Euro-Par2010, Part II LNCS6272, pp.287-303). Thread safety is obtained via Thread Local Storage.

Similar to the sequential version of Geant4, master and workers are represented by instances of classes inheriting from **G4RunManager**: the **G4MTRunManager** class represents the master model, while **G4WorkerRunManager** instances represent worker models. The user is responsible for instantiating a single **G4MTRunManager** (or derived user-class) instance. This class will instantiatiate and control one or more **G4WorkerRunManager** instances. Users should never instantiate directly an instance of the **G4WorkerRunMAnager** class.

A simplified class-diagram of the relevant classes for multi-threading and their relationship is shown here:

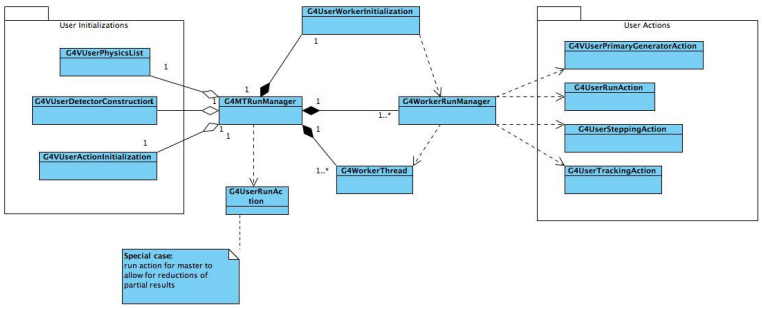

Figure 2.26. Relevant classes and their interaction for multi-threaded applications

As in sequential Geant4 users interact with the Geant4 kernel via user initializations and user actions. User initializations (**G4VUserDetectorConstruction**, **GVUserPhysicsList** and the new **G4VUserActionInitializtion**) instances are shared among all threads (as such they are registered to the **G4MTRunManager** instance); while user actions (**G4VUserPrimaryGeneratorAction**, **G4UserRunAction**, **G4UserSteppingAction** and **G4UserTrackingAction**) are not shared and a separate instance exists for each thread. Since the master does not perform simulation of events user actions do not have functions for **G4MTRunManager** and should not be assigned to it. **G4RunAction** is the exception to this rule since it can be attached to the master **G4MTRunManager** to allow for merging of partial results produced by workers.

2.14.3. Memory handling in Geant4 Version 10.0

2.14.3.1. Introduction

In Geant4 we distinguish two broad types of classes: ones whose instances are separate for each thread (such as a physics process, which has a state), and ones whose instances are shared between threads (e.g. an element **G4Element** which holds constant data).

A few cases classes exist which have mixed behavior - part of their state is constant, and part is per-worker. A simple example of this is a particle definition, such as **G4Electron**, which holds both data (which is constant) and a pointer to the **G4ProcessManager** object for electrons - which must be different for each worker (thread).

We handle these 'split' classes specially, to enable data members and methods which correspond to the per-thread state to give a different result on each worker thread. The implementation of this requires an array for each worker

(thread) and an additional indirection - which imposes a cost each time the method is called. However this overhead is small and has been measured to be about 1%. In this section we will discuss the details of how we achieve thread-safety for different use-cases. The information contained here is of particular relevance for toolkit developers that need to adapt code to multi-threading to increase performances (typically to reduce the memory footprint of an application sharing between threads' memory consuming objects). It is however of general interest to understand some of the more delicate aspects of multi-threading.

2.14.3.2. Thread safety and sharing of objects

To better understand how memory is handled and what are the issues introduced by multi-threading it is easier to proceed with a simplified example.

Let us consider the simplest possible class **G4Class** that consists of a single data member:

```
class G4Class {
    [static] G4double fValue; //static keyword is optional
};
```

Our goal is to transform the code of **G4Class** to make it thread-safe. A class (or better, a method of a class) is thread-safe if more than one thread can simultaneously operate on the class data member or its methods without interfering with each other in an unpredictable way. For example if two threads concurrently write and read the value of the data field *fValue* and this data field is shared among threads, the two threads can interfere with each other if no special code to synchronize the thread is added. This condition is called *data-race* and is particularly dangerous and difficult to debug.

A classical way to solve the data-race problem is to protect the critical section of the code and the concurrent access to a shared memory location using a lock or a mutex (see section Threading model utilities and functions. However this technique can reduce overall performance because only one thread at a time is allowed to be executed. It is important to reduce to a minimum the use of locks and mutexes, especially in the event loop. In Geant4 we have achieved thread-safety via the use of *thread local storage*. This allows for virtually lock-free code at the price of an increased memory footprint and a small CPU penalty. Explanations of thread-local storage are provided by several external resources. For a very simple introduction, but adequate for our discussion, web resources give sufficient detail (e.g. wikipedia).

Before going into the details of how to use the thread-local storage mechanism we need to introduce some terminology.

We define an instance of a variable to be *thread-local* (or *thread-private*) if each thread owns a copy of the variable. A *thread-shared* variable, on the contrary, is an instance of a variable that is shared among the threads (i.e. all threads have access to the same memory location holding the value of the variable). If we need to share the same memory location containing the value of *fValue* between several instances of **G4Class** we call the data field *instance-shared* otherwise (the majority of cases) it is *instance-local*. These defintions are an over-simplification that does not take into account pointers and sharing/ownership of the pointee, however the issues that we will discuss in the following can be extended to the case of pointers and the (shared) pointee.

It is clear that, for the case of *thread-shared* variables, all threads need synchronization to avoid data-race conditions (it is worth recalling that there are no race conditions if the variable is accessed only to be read, for example in the case that the variable is marked as *const*.

One or more instances of **G4Class** can exist at the same time in our application. These instances can be thread-local (e.g. **G4VProcess**) or thread-shared (e.g. **G4LogicalVolume**). In addition the class data field *fValue* can be by itself thread-local or thread-shared. The actions to be taken to transform the code depend on three key aspects:

- Do we need to make the instance(s) of **G4Class**, *thread-local* or *thread-shared*?
- Do we need to make the data field *fValue*, *thread-local* or *thread-shared*?
- In case more than one instance of **G4Class** exits at the same time, do we need *fValue* to be *instance-local* or *instance-shared*?

This gives rise to 8 different possible combinations, summarized in the following figures, each one discussed in detail in the following.

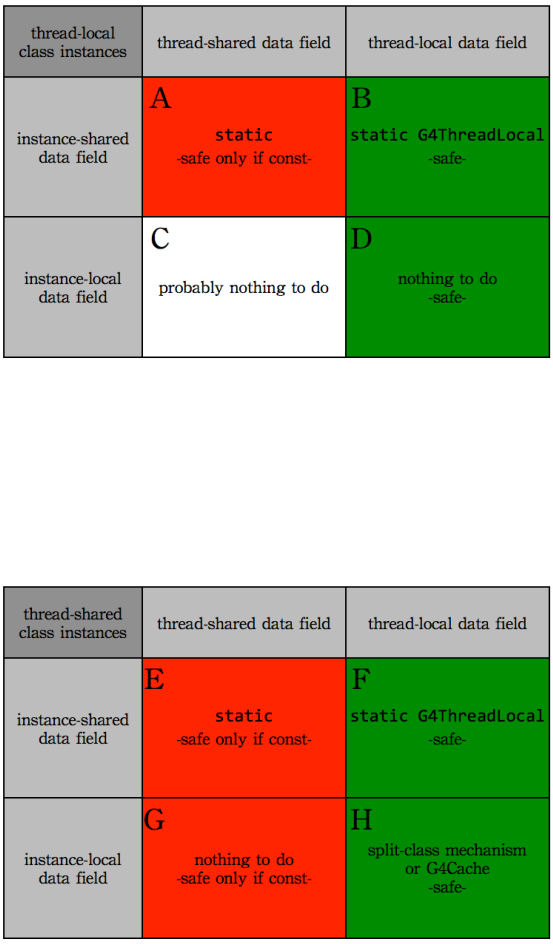

Figure 2.27. The eight possible scenarios for sharing of objects

2.14.3.2.1. Case A: thread-local class instance(s), thread-shared and instance-shared data field

In this case each thread has its own instance(s) of type **G4Class**. We need to share *fValue* both among threads and among instances. As for a sequential application, we can simply add the *static* keyword to the declaration of *fValue*. This technique is common in Geant4 but has the disadvantage that the resulting code is thread-unsafe (unless locks are used). Trying to add *const* or modify its value (with the use of a lock) only outside of the event loop is the simplest and best solution:

```
class G4Class {
    static const G4double fValue;
};
```

2.14.3.2.2. Case B: thread-local class instance(s), thread-local and instance-shared data field.

This scenario is also common in Geant4: we need to share a variable (e.g. a data table) between instances of the same class. However it is impractical or it would lead to incorrect results if we share among threads *fValue* (i.e. the penalty due to the need of locks is high or the data field holds a event-dependent information). To make the code thread-safe we mark the data field thread-local via the keyword **G4ThreadLocal**:

```
#include "G4Types.hh"
class G4Class {
    static G4ThreadLocal G4double fValue;
};
```

It should be noted that only simple data types can be declared **G4ThreadLocal**. More information and the procedures to make an object instance thread-safe via thread-local-storage are explained in this web-page.

2.14.3.2.3. Case C: thread-local class instance(s), thread-shared and instance-local data field

One possible use-case is the need to reduce the application memory footprint, providing a component to the thread-local instances of **G4Class** that is shared among threads (e.g. a large cross-section data table that is different for each instance). Since this scenario strongly depends on the implementation details it is not possible to define a common strategy that guarantees thread-safety. The best option is to try to make this shared component *const*.

2.14.3.2.4. Case D: thread-local class instance(s), thread-local and instance-local data field

This case is the simplest; nothing has to be changed in the original code.

2.14.3.2.5. Case E: thread-shared class instance(s), thread-shared and instance-shared data field

With respect to thread-safety this case is equivalent to Case A, and the same recommendations and comments hold.

2.14.3.2.6. Case F: thread-shared class instance(s), thread-local and instance-shared data field

Concerning thread-safety this case is equivalent to Case B, and the same recommendations and comments hold.

2.14.3.2.7. Case G: thread-shared class instance(s), thread-shared and instance-shared data field

Since the class instances are shared among threads the data fields are automatically thread-shared. No action is needed, however access to the data fields is in general thread unsafe, and the same comments and recommendations for Case A are valid.

2.14.3.2.8. Case H: thread-shared class instance(s), thread-local and instance-local data field

This is the most complex case and it is relatively common in Geant4 Version 10.0. For example **G4ParticleDefinition** instances are shared among the threads, but the **G4ProcessManager** pointer data field needs to be thread- and instance-local. To obtain thread-safe code two possible solutions exist:

- Use the split-class mechanism. This requires some deep understanding of Geant4 multi-threading and coordination with the kernel developers. Split-classes result in thread-safe code with good CPU performance, however they also require modification in other aspects of the kernel category (in particular the run category). The idea behind the split-class mechanism is that each thread-shared instance of **G4Class** initializes the thread-local data fields by *copying* the initial status from the equivalent instance of the master, which is guaranteed to be fully configured. Additional details on split classes are available in a dedicated section. An important side effect of the split-class mechanism is that exactly the same number of instances of G4Class must exist in each thread (e.g. the full set of G4Particles owned by the master is shared by threads. If a new particle is created, this has to be shared by all threads).
- If performance is not a concern a simpler solution is available. This is a simplified version of the split-class mechanism that does not copy the initial status of the thread-local data field from the master thread. A typical example is a *cache* variable that reduces CPU usage, storing in memory the value of a CPU intensive calculation for several events. In such a case the **G4Cache** utility class can be employed (see `G4Cache`).

2.14.3.3. Details on the split classes mechanism

We describe here the split-class mechanism, central to Geant4 multi-threading, by developing a thread-safe split-class starting from our simplified example of G4Class. It will be clear that this technique allows for minimal changes of the public API of the classes and thus is very suitable for making thread-safe code without breaking backward compatibility.

To better describe the changes we introduce a setter and getter methods in the sequential version of our class (e.g. before migration to multi-threading):

```
class G4Class
{
 private:
    G4double fValue;
 public:
    G4Class() {  }
    void SetMyData( G4double aValue ) { fValue = aValue; }
    G4double GetMyData() const { return fValue; }
};
```

Instances of this class will be shared among threads (because they are memory-consuming objects) and we want to transform this class into a split-class.

As a first step we add to the declaration of *fValue* the TLS keyword **G4ThreadLocal** (in a POSIX system, this is a typedef to *__thread*). Unfortunately there are several constraints on what can be specified as TLS. In particular the data member has to be declared static (or be a global variable):

```
#include "tls.hh"
class G4Class
{
 private:
    static G4ThreadLocal G4double fValue;
 public:
    G4Class()  {  }
    void SetMyData( G4double aValue ) { fValue = aValue; }
    G4double GetMyData() const { return fValue; }
};
G4ThreadLocal G4double G4Class::fValue = -1;
```

The problem occurs if we need more than one instance of type G4Class with an instance-local different value of **fValue**. How can this behavior be obtained now that the we have declared the data member as *static*? The method used to solve this problem is called the *split class mechanism*. The idea is to collect all thread-local data fields into a separate new class, instances of which (one per original instance of G4Class) are organized in an array. This array is accessed via an index representing a unique identifier of a given class instance.

We can modify the code as follows:

```
class G4ClassData {
public:
   G4double fValue;
   void intialize() {
           fValue = -1;
   }
};

typedef G4Splitter>G4ClassData< G4ClassManager;
typedef G4ClassManager G4ClassSubInstanceManager;

#define G4MT_fValue ((subInstanceManager.offset[gClassInstanceId]).fValue)
class G4Class {
private:
    G4int gClassInstanceId;
    static G4ClassSubInstanceManager subInstanceManager;
public:
```

```
    G4Class()
   {
      gClassInstanceId = subInstanceManager.CreateSubInstance();
   }
   void SetMyData( G4double aValue ) { G4MT_fValue = aValue; }
   G4double GetMyData() const { return G4MT_fValue; }
};

G4ClassSubInstanceManager G4Class::subInstanceManager;
template >class G4ClassData< G4ThreadLocal G4int G4Splitter>G4ClassData<::workertotalspace = 0;
template >class G4ClassData< G4ThreadLocal G4int G4Splitter>G4ClassData<::offset = 0;
```

As one can see, the use of the value of *fValue* variable is very similar to how we use it in the original sequential mode, all the handling of the TLS is done in the template class **G4Splitter** that can be implemented as:

```
template <class T>
class G4Splitter
{
  private:
    G4int totalobj;
  public:
    static G4ThreadLocal G4int workertotalspace;
    static G4ThreadLocal T* offset;
  public:
    G4Splitter() : totalobj(0) {}
    G4int CreateSubInstance()
    {
       totalobj++;
       if ( totalobj > workertotalspace ) { NewSubInstances(); }
       return (totalobj-1);
    }
    void NewSubInstances()
    {
         if ( workertotalspace >=totalobj ) { return; }
         G4int originaltotalspace = workertotalspace;
         workertotalspace = totalobj + 512;
         offset = (T*) realloc( offset , workertotalspace * sizeof(T) );
         if ( offset == 0 )
         {
            G4Excepetion( "G4Splitter::NewSubInstances","OutOfMemory",FatalException,"Cannot malloc space!");
         }
        for ( G4int i = originaltotalspace; i&lt; workertotalspace ; i++)
        {
            offset[i].intialize();
        }
    }
   void FreeWorker()
   {
     if ( offset == 0 ) { return; }
     delete offset;
   }
};
```

Let's consider a function that can be called concurrently by more than one thread:

```
#include "G4Class.hh"
//Variables at global scope
G4Class a;
G4Class b;

void foo()
{
   a.SetMyData(0.1); //First instance
   b.SetMyData(0.2); //Second instance
   G4cout << a.GetMyData()<< " "<< b.GetMyData() << G4endl;
}
```

We expect that each thread will write on the screen: "0.1 0.2"

When we declare the variable *a*, the static object *subInstanceManager* in memory has the state:

```
totalobj = 0
TLS workertotalspace = 0
TLS offset = NULL
```

The constructor of G4Class calls *CreateSubInstance*, and since at this point *totalobj* equals 1, *G4Splitter::NewSubInstances()* is called. This will create a buffer of 512 pointers of type **G4ClassData**, each of which is initialized (via **G4ClassData::initialize()**) to the value -1. Finally, *G4Splitter::CreateSubInstance()* returns 0 and *a.gClassInstanceId* equals 0. When *a.SetMyData(0.1)* is called, the call is equivalent to:

```
subInstanceManager.offset[0].fValue = 0.1;
```

When now we declare the instance *b* the procedure is repeated, except that, since *totalobj* now equals 1 and *workertotalspace* is 512, there is no need to call *G4Splitter::NewSubInstances()* and we use the next available array position in *offset*. Only if we create more than 512 instances of **G4Class** is the memory array reallocated with more space for the new G4ClassData instances.

Since *offset* and *workertotalspace* are marked *G4ThreadLocal* this mechanism allows each thread to have its own copy of *fValue*. The function *foo()* can be called from different threads and they will use the thread-shared *a* and *b* to access a thread-local *fValue* data field. No data-race condition occurs and there is no need for mutexes and locks.

An additional complication is that if the initialization of the thread-local part is not trivial and we want to copy some values from the corresponding values of the master thread (in our example, how is *fValue* to be initialized to a value that depends on the run condition?). The initial status of the thread-local data field must be initialized, for each worker, in a controlled way. The run cateogory classes must be modified to preapre the TLS space of each thread before any work is performed.

The following diagram shows the chain of calls in **G4ParticleDefinition** when a thread needs to access a process pointer:

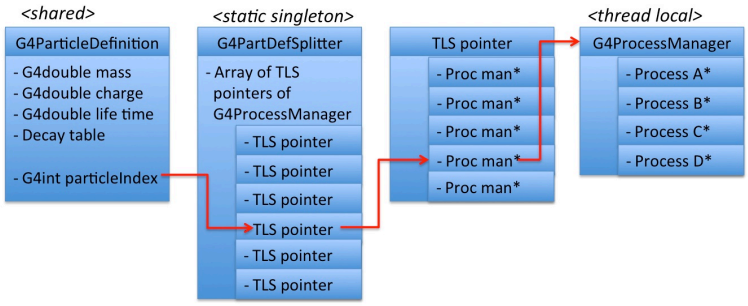

Figure 2.28. Simplified view of the split-class mechanism

2.14.3.3.1. List of split-classes

In Geant4 Version 10.0 the following are split-classes:

- For geometry related split classes the class **G4GeomSplitter** implements the split-class mechanism. These are the geometry-related split-classes:
  i. **G4LogicalVolume**
  ii. **G4PhysicalVolume**
  iii. **G4PVReplica**
  iv. **G4Region**
  v. **G4PloyconeSide**
  vi. **G4PolyhedraSide**
- For Physics related split-classes the classes **G4PDefSplitter** and **G4VUPLSplitter** implement the split-class mechanism. These are the physics-related split-classes:
  i. **G4ParticleDefinition**
  ii. **G4VUserPhysicsList**
  iii. **G4VModularPhysicsList**
  iv. **G4VPhysicsConstructor**

2.14.3.4. Explicit memory handling

In the following, some utility classes and functions to help memory handling are discussed. Before going into detail it should be noted that all of these utilities have a (small) CPU and memory performance penalty; they should be used with caution and only if other simpler methods are not possible. In some cases limitations are present.

2.14.3.4.1. The template class G4Cache

In many cases the full functionality of split-classes is not needed and what we really want are independent thread-local and instance-local data fields in thread-shared instances of G4Class. A concrete example would be a class representing a cross-section that is made shared because of its memory footprint. It requires a data field to act as a *cache* to store the value of a CPU intensive calculation. Since different threads share this instance we need to transform the code in a manner similar to what we do for the split-class mechanism. The helper class **G4Cache** can be used for this purpose (note that the complication of the initial value of the thread-local data field is not present in this case).

G4Cache is a template class that implements a light-weight split-classes mechanism. Being a template it allows for storing any user-defined type. The public API of this class is very simple and it provides two methods

```
T& G4Cache<T>::Get() const;
void G4Cache<T>::Put(const T& val) const;
```

to access a thread-local instance of the cached object. For example:

```
#include "G4Cache.hh"
class G4Class {
    G4Cache<G4double> fValue;
    void foo() {
          // Store a thread-local value
          G4double val = someHeavyCalc();
          fValue.Put( val );
    }
    void bar() {
         //Get a thread-local value:
         G4double local = fValue.Get();
    }
};
```

Since *Get* returns a reference to the cached object is possible to avoid the use of *Put* to update the cache:

```
void G4Class::bar() {
    //Get a reference to the thread-local value:
   G4double  &local = fValue.Get();
   // Use local as in the original sequential code, cache is updated, without the need to use Put
  local++;
}
```

In case the cache holds an instance of an object it is possible to implement lazy initialization, as in the following example:

```
#include "G4Cache.hh"
class G4Class {
    G4Cache<G4Something*> fValue;
    void bar() {
         //Get a thread-local value:
         G4Something* local = fValue.Get();
         if ( local == 0 ) {
                   local = new G4Something( …. );
                   //warning this may cause a memory leak. Use of G4AutoDelete can help, see later
         }
    }
};
```

Since the use of G4Cache implies some CPU penalty, it is good practice to try to minimize its use. For example, do not use a single G4Cache for several data fields; instead use a helper structure as the template parameter for G4Cache:

```
class G4Class {
   struct {
       G4double fValue1;
       G4Something* fValue2;
   } ToBeCached_t;
  G4Cache<ToBeCached_t> fCache;
};
```

Two specialized versions of G4Cache exist that implement the semantics of **std::vector** and **std::map**

- **G4VectorCache<T>** implements a thread-local **std::vector<T>** with methods *Push_back(…)* , *operator[]*, *Begin()*, *End()*, *Clear()*, *Size()* and *Pop_back()*
- **G4MapCache<K,V>** implements a thread-local **std::map<K,V>** with methods *Insert(…)*, *Begin()*, *End()*, *Find(…)*, *Size()*, *Get(…)*, *Erase(…)*, *operator[]* and introduces the method *Has(…)*

A detailed example of the use of these cache classes is discussed in the unit test *source/global/management/test/testG4Cache.cc*.

2.14.3.4.2. G4AutoDelete namespace

During the discussion of G4Cache we have shown the example of storing a pointer to a dynamically created object. A common problem is to correctly delete this object at the end of its life-cycle. Since the G4Class instance is thread-shared, it is not possible to delete the cached object in the destructor of G4Class because it is called by the master and the thread-local instances of the cached object will not be deleted. In some cases, to solve this problem, it is possible to use a helper introduced in the namespace **G4AutoDelete**. A simplified garbage collection mechanism without reference counting is implemented:

```
#include "G4AutoDelete.hh"
void G4Class::bar() {
    //Get a thread-local value:
    G4Something* local = fValue.Get();
    if ( local == 0 ) {
            local = new G4Something( …. );
            G4AutoDelete::Register( local ); //All thread instances will be delete automatically
    }
}
```

This technique will delete all instances of the registered objects **at the end of the program**, after the main function has returned (if they were declared *static*).

This method has several limitations:

i. Registered objects will be deleted only at the end of the program
ii. The order in which objects of different type will be deleted is not specified
iii. Once an object is registered it cannot be deleted anymore explicitly by user
iv. The objects that are registered with this method cannot contain data filed marked G4ThreadLocal and cannot be a split-classes
v. Registered object cannot make use of **G4Allocator** functionalities
vi. These restrictions apply to all data members for which the class owns property

In addition, since the objects will be deleted in a non-specified order after the main program exit, it is recommended to provide a very simple destructor that does not depend on other objects (in particular should not call any kernel functionality).

2.14.3.4.3. Thread Private *singleton*

In Geant4 the singleton pattern is used in several cases. The majority of the managers are implemented via the singleton pattern, the simplest of which is:

```
class G4Singleton {
public:
   G4Singleton* GetInstance() {
```

```
        static G4Singleton anInstance;
        return&anInstance;
   }
};
```

With multi-threading, many managers and singletons are thread-local. For this reason they have been transformed to:

```
class G4Singleton {
private:
   static G4ThreadLocal* instance;
public:
   G4Singleton* GetInstance() {
        if ( instance == 0 ) instance = new G4Singleton;
        return instance;
   }
};
```

This causes a memory leak: it is not possible to delete thread-local instances of the *singletons*. To solve this problem the class **G4ThreadLocalSingleton** has been added to the toolkit. This template class has a single public method *T* G4ThreadLocalSingleton<T>::Instance()* that returns a pointer to a thread-local instance of T. The thread-local instances of T will be deleted, as in the case of G4Cache, at the end of the program.

The example code can be transformed to:

```
#include "G4ThreadLocalSingleton.hh"
class G4Singleton {
   friend class G4ThreadLocalSingleton<G4Singleton>;
public:
   G4Singleton* GetInstance() {
       static G4ThreadLocalSingleton<G4Singleton> theInstance;
       return theInstance.Instance();
   }
};
```

2.14.4. Threading model utilities and functions

Geant4 parallelism is based on POSIX standards and in particular on the *pthreads* library. However all functionalities have been *wrapped* around Geant4 specific names. This allows the inclusion of the WIN32 threading model. In the following, the main functionalities available in the *global/management* category are discussed.

2.14.4.1. Types and functions related to the use of threads

G4Thread defines the type for threads (POSIX **pthread_t**). The types **G4ThreadFunReturnType** and **G4ThreadFunArgType** define respectively the return value and the argument type for a function executed in a thread. Use **G4THREADCREATE** and **G4THREADJOIN** macros to respecively create and join a thread. **G4Pid_t** is the type for the PID of a thread.

Example:

```
#include "G4Threading.hh"

//Define a thread-function using G4 types
G4ThreadFunReturnType myfunc(  G4ThreadFunArgType val) {
  double value = *(double*)val;
  MESSAGE("value is:"<<value);
  return /*(G4ThreadFunReturnType)*/NULL;
}

//Example: spawn 10 threads that execute myfunc
int main(int,char**) {
  MESSAGE( "Starting program ");
  int nthreads = 10;
  G4Thread* tid = new G4Thread[nthreads];
  double *valss = new double[nthreads];
```

```
  for ( int idx = 0 ; idx < nthreads ; ++idx ) {
    valss[idx] = (double)idx;
    G4THREADCREATE(&(tid[idx]) , myfunc,&(valss[idx]) );
  }
  for ( int idx = 0 ; idx < nthreads ; ++idx ) {
    G4THREADJOIN( (tid[idx]) );
  }
  MESSAGE( "Program ended ");
  return 0;
}
```

2.14.4.2. Types and functions related to the use of mutexes and conditions

G4Mutex is the type for mutexes in Geant4 (POSIX **pthread_mutex_t**). The **G4MUTEX_INITIALIZER** and **G4MUTEXINIT** macros are used to initialize a mutex. Use **G4MUTEXLOCK** and **G4MUTEXUNLOCK** functions to lock/unlock a mutex. The **G4AutoLock** class helps the locking/unlocking of a mutex and should be always be used instead of **G4MUTEXLOCK/UNLOCK**.

Example:

```
#include "G4Threading.hh"
#include "G4AutoLock.hh"

//Create a global mutex
G4Mutex mutex = G4MUTEX_INITIALIZER;
//Alternatively, call in the main function G4MUTEXINIT(mutex);

//A shared resource (i.e. manipulated by all threads)
G4int aValue = 1;

G4ThreadFunReturnType myfunc(  G4ThreadFunArgType ) {
  //Explicit lock/unlock
  G4MUTEXLOCK(&mutex );
  ++aValue;
  G4MUTEXUNLOCK(&mutex );
  //The following should be used instead of the previous because it guarantees automatic
  //unlock of mutex.
  //When variable l goes out of scope, G4MUTEXUNLOCK is automatically called
  G4AtuoLock l(&mutex);
  --aValue;
  //Explicit lock/unlock. Note that lock/unlock is only tried if mutex is already locked/unlock
  l.lock();
  l.lock();//No problem here
  ++aValue;
  l.unlock();
  l.lock();
  --aValue;
  return /*(G4ThreadFunReturnType)*/NULL;
}
```

A complete example of the usage of these functionalities is discussed in the unit test *source/global/management/test/ThreadingTest.cc*.

Conditions are also available via the **G4Condition** type, the **G4CONDITION_INITIALIZER** macro and the two functions **G4CONDITIONWAIT** and **G4CONDITIONBORADCAST**. The use of conditions allows the barrier mechanism (e.g. synchronization point for threads) to be implemented. A detailed example on the use of conditions and how to implement correctly a barrier is discussed in **G4MTRunManager** code (at the end of file *source/run/src/G4MTRunManager.cc*). In general there should be no need for kernel classes (with the exception of run category) to use conditions since threads are considered independent and do not need to communicate between them.

2.14.5. Additional material

In this chapter we discussed in detail what are probably the most critical aspects of multi-threading capabilities in Geant4. Additional material can be found in online resources. The main entry point is the Geant4 multi-threading

task-force twiki page. The *Application Developers Guide* contains general information regarding multi-threading that is also relevant for Toolkit Developers.

A beginner's guide to multi-threading targeted to Geant4 developers has been presented during the *18th Collaboration Meeting*: agenda

For additional information consult this page and this page

Several contributions at the 18th Collaboration Meeting discuss multi-threading:

- Plenary Session 3 - Geant4 version 10 (part 1): agenda
- Hadronics issues related to MT: agenda
- Developments for multi-threading: work-spaces: contribution
- Status of the planned developments: coding guidelines, MT migration, g4tools migration, code review: contribution
- G4MT CP on MIC Architecture: contribution

Finally, a few articles and proceedings have been prepared:

- X. Dong et al, Creating and Improving Multi-Threaded Geant4, Journal of Physics: Conference Series 396, no. 5, p. 052029.
- X. Dong et al, Multithreaded Geant4: Semi-automatic Transformation into Scalable Thread-Parallel Software, Euro-Par 2010 - Parallel Pro- cessing (2010), vol. 6272, pp. 287-303.
- S. Ahn et al, Geant4-MT: bringing multi-threaded Geant4 into production, to be published in SNA&MC2013 proceeding

Chapter 3. Extending Toolkit Functionality

3.1. Geometry

3.1.1. What can be extended ?

Geant4 already allows a user to describe any desired solid, and to use it in a detector description, in some cases, however, the user may want or need to extend Geant4's geometry. One reason can be that some methods and types in the geometry are general and the user can utilise specialised knowledge about his or her geometry to gain a speedup. The most evident case where this can happen is when a particular type of solid is a key element for a specific detector geometry and an investment in improving its runtime performance may be worthwhile.

To extend the functionality of the Geometry in this way, a toolkit developer must write a small number of methods for the new solid. We will document below these methods and their specifications. Note that the implementation details for some methods are not a trivial matter: these methods must provide the functionality of finding whether a point is inside a solid, finding the intersection of a line with it, and finding the distance to the solid along any direction. However once the solid class has been created with all its specifications fulfilled, it can be used like any Geant4 solid, as it implements the abstract interface of G4VSolid.

Other additions can also potentially be achieved. For example, an advanced user could add a new way of creating physical volumes. However, because each type of volume has a corresponding navigator helper, this would require to create a new Navigator as well. To do this the user would have to inherit from G4Navigator and modify the new Navigator to handle this type of volumes. This can certainly be done, but will probably be made easier to achieve in the future versions of the Geant4 toolkit.

3.1.2. Adding a new type of Solid

We list below the required methods for integrating a new type of solid in Geant4. Note that Geant4's specifications for a solid pay significant attention to what happens at points that are within a small distance (tolerance, **kCarTolerance** in the code) of the surface. So special care must be taken to handle these cases in considering all different possible scenarios, in order to respect the specifications and allow the solid to be used correctly by the other components of the geometry module.

Creating a derived class of G4VSolid

The solid must inherit from G4VSolid or one of its derived classes and implement its virtual functions.

Mandatory member functions you must define are the following pure virtual of G4VSolid:

```
EInside Inside(const G4ThreeVector& p)
G4double DistanceToIn(const G4ThreeVector& p)
G4double DistanceToIn(const G4ThreeVector& p, const G4ThreeVector& v)
G4ThreeVector SurfaceNormal(const G4ThreeVector& p)
G4double DistanceToOut(const G4ThreeVector& p)
G4double DistanceToOut(const G4ThreeVector& p, const G4ThreeVector& v,
                       const G4bool calcNorm=false,
                             G4bool *validNorm=0, G4ThreeVector *n)
G4bool CalculateExtent(const EAxis pAxis,
                       const G4VoxelLimits& pVoxelLimit,
                       const G4AffineTransform& pTransform,
                             G4double& pMin,
                             G4double& pMax) const
G4GeometryType GetEntityType() const
std::ostream& StreamInfo(std::ostream& os) const
```

They must perform the following functions

```
 EInside Inside(const G4ThreeVector& p)
```

This method must return:

- kOutside if the point at offset p is outside the shape boundaries plus Tolerance/2,
- kSurface if the point is <= Tolerance/2 from a surface, or
- kInside otherwise.

```
G4ThreeVector SurfaceNormal(const G4ThreeVector& p)
```

Return the outwards pointing unit normal of the shape for the surface closest to the point at offset p.

```
G4double DistanceToIn(const G4ThreeVector& p)
```

Calculate distance to nearest surface of shape from an outside point p. The distance can be an underestimate.

```
G4double DistanceToIn(const G4ThreeVector& p, const G4ThreeVector& v)
```

Return the distance along the normalised vector v to the shape, from the point at offset p. If there is no intersection, return kInfinity. The first intersection resulting from `leaving' a surface/volume is discarded. Hence, this is tolerant of points on surface of shape.

```
G4double DistanceToOut(const G4ThreeVector& p)
```

Calculate distance to nearest surface of shape from an inside point. The distance can be an underestimate.

```
G4double DistanceToOut(const G4ThreeVector& p, const G4ThreeVector& v,
                       const G4bool calcNorm=false,
                       G4bool *validNorm=0, G4ThreeVector *n=0);
```

Return distance along the normalised vector v to the shape, from a point at an offset p inside or on the surface of the shape. Intersections with surfaces, when the point is not greater than kCarTolerance/2 from a surface, must be ignored.

If calcNorm is true, then it must also set validNorm to either

- true, if the solid lies entirely behind or on the exiting surface. Then it must set n to the outwards normal vector (the Magnitude of the vector is not defined).
- false, if the solid does not lie entirely behind or on the exiting surface.

If calcNorm is false, then validNorm and n are unused.

```
G4bool CalculateExtent(const EAxis pAxis,
                       const G4VoxelLimits& pVoxelLimit,
                       const G4AffineTransform& pTransform,
                             G4double& pMin,
                             G4double& pMax) const
```

Calculate the minimum and maximum extent of the solid, when under the specified transform, and within the specified limits. If the solid is not intersected by the region, return false, else return true.

```
G4GeometryType GetEntityType() const;
```

Provide identification of the class of an object (required for persistency and STEP interface).

```

```

```
std::ostream& StreamInfo(std::ostream& os) const
```

Should dump the contents of the solid to an output stream.

The method:

```
G4VSolid* Clone() const
```

should be implemented for every solid to provide a way to clone themselves in a new object with same specifications.

The method:

```
G4ThreeVector GetPointOnSurface() const
```

returns a random point located on the surface of the solid. Points returned should not necessarily be uniformly distributed.

The method:

```
G4double GetCubicVolume()
```

should be implemented for every solid in order to cache the computed value (and therefore reuse it for future calls to the method) and to eventually implement a precise computation of the solid's volume. If the method will not be overloaded, the default implementation from the base class will be used (estimation through a Monte Carlo algorithm) and the computed value will not be stored.

The method:

```
G4double GetSurfaceArea()
```

should be implemented for every solid in order to cache the computed value (and therefore reuse it for future calls to the method) and to eventually implement a precise computation of the solid's surface area. If the method will not be overloaded, the default implementation from the base class will be used (estimation through a Monte Carlo algorithm) and the computed value will not be stored.

There are a few member functions to be defined for the purpose of visualisation. The first method is mandatory, and the next four are not.

```
 // Mandatory
 virtual void DescribeYourselfTo (G4VGraphicsScene& scene) const = 0;

 // Not mandatory
 virtual G4VisExtent GetExtent() const;
 virtual G4Polyhedron* CreatePolyhedron () const;
 virtual G4NURBS*      CreateNURBS      () const;
 virtual G4Polyhedron* GetPolyhedron    () const;
```

What these methods should do and how they should be implemented is described here.

```
void DescribeYourselfTo (G4VGraphicsScene& scene) const;
```

This method is required in order to identify the solid to the graphics scene. It is used for the purposes of ``double dispatch''. All implementations should be similar to the one for G4Box:

```
void G4Box::DescribeYourselfTo (G4VGraphicsScene& scene) const
{
  scene.AddSolid (*this);
}
```

The method:

```
 G4VisExtent GetExtent() const;
```

provides extent (bounding box) as a possible hint to the graphics view. You must create it by finding a box that encloses your solid, and returning a VisExtent that is created from this. The G4VisExtent must presumably be given the minus x, plus x, minus y, plus y, minus z and plus z extents of this ``box''. For example a cylinder can say

```
G4VisExtent G4Tubs::GetExtent() const
{
  // Define the sides of the box into which the G4Tubs instance would fit.
  return G4VisExtent (-fRMax, fRMax, -fRMax, fRMax, -fDz, fDz);
}
```

The method:

```
 G4Polyhedron* CreatePolyhedron () const;
```

is required by the visualisation system, in order to create a realistic rendering of your solid. To create a G4Polyhedron for your solid, consult G4Polyhedron.

While the method:

```
 G4Polyhedron* GetPolyhedron () const;
```

is a ``smart'' access function that creates on requests a polyhedron and stores it for future access and should be customised for every solid.

3.1.3. Modifying the Navigator

For the vast majority of use-cases, it is not indeed necessary (and definitely not advised) to extend or modify the existing classes for navigation in the geometry. A possible use-case for which this may apply, is for the description of a new kind of physical volume to be integrated. We believe that our set of choices for creating physical volumes is varied enough for nearly all needs. Future extensions of the Geant4 toolkit will probably make easier exchanging or extending the G4Navigator, by introducing an abstraction level simplifying the customisation. At this time, a simple abstraction level of the navigator is provided by allowing overloading of the relevant functionalities.

Extending the Navigator

The main responsibilities of the Navigator are:

- locate a point in the tree of the geometrical volumes;
- compute the length a particle can travel from a point in a certain direction before encountering a volume boundary.

The Navigator utilises one helper class for each type of physical volume that exists. You will have to reuse the helper classes provided in the base Navigator or create new ones for the new type of physical volume.

To extend G4Navigator you will have then to inherit from it and modify these functions in your ModifiedNavigator to request the answers for your new physical volume type from the new helper class. The ModifiedNavigator should delegate other cases to the Geant4's standard Navigator.

Replacing the Navigator

Replacing the Navigator is another possible operation. It is similar to extending the Navigator, in that any types of physical volume that will be allowed must be handled by it. The same functionality is required as described in the previous section.

However the amount of work is probably potentially larger, if support for all the current types of physical volumes is required.

The Navigator utilises one helper class for each type of physical volume that exists. These could also potentially be replaced, allowing a simpler way to create a new navigation system.

3.2. Electromagnetic Fields

3.2.1. Creating a New Type of Field

Geant4 currently handles magnetic and electric fields and, in future releases, will handle combined electromagnetic fields. Fields due to other forces, not yet included in Geant4, can be provided by describing the new field and the force it exerts on a particle passing through it. For the time being, all fields must be time-independent. This restriction may be lifted in the future.

In order to accommodate a new type of field, two classes must be created: a field type and a class that determines the force. The Geant4 system must then be informed of the new field.

A new Field class

A new type of Field class may be created by inheriting from G4Field

```
     class NewField : public G4Field
     {
        public:
           void  GetFieldValue( const  double Point[3],
                                       double *pField )=0;
     }
```

and deciding how many components your field will have, and what each component represents. For example, three components are required to describe a vector field while only one component is required to describe a scalar field.

If you want your field to be a combination of different fields, you must choose your convention for which field goes first, which second etc. For example, to define an electromagnetic field we follow the convention that components 0,1 and 2 refer to the magnetic field and components 3, 4 and 5 refer to the electric field.

By leaving the GetFieldValue method pure virtual, you force those users who want to describe their field to create a class that implements it for their detector's instance of this field. So documenting what each component means is required, to give them the necessary information.

For example someone can describe DetectorAbc's field by creating a class DetectorAbcField, that derives from your NewField

```
class DetectorAbcField : public NewField
{
  public:
     void  MyFieldGradient::GetFieldValue( const double Point[3],
                                           double *pField );
}
```

They then implement the function GetFieldValue

```
```

```
     void  MyFieldGradient::GetFieldValue( const  double Point[3],
                                           double *pField )
     {
        // We expect pField to point to pField[9];
        // This & the order of the components of pField is your own
        // convention

        // We calculate the value of pField at Point ...
     }
```

A new Equation of Motion for the new Field

Once you have created a new type of field, you must create an Equation of Motion for this Field. This is required in order to obtain the force that a particle feels.

To do this you must inherit from G4Mag_EqRhs and create your own equation of motion that understands your field. In it you must implement the virtual function EvaluateRhsGivenB. Given the value of the field, this function calculates the value of the generalised force. This is the only function that a subclass must define.

```
    virtual void EvaluateRhsGivenB( const  G4double y[],
                             const  G4double B[3],
                                    G4double dydx[] ) const = 0;
```

In particular, the derivative vector dydx is a vector with six components. The first three are the derivative of the position with respect to the curve length. Thus they should set equal to the normalised velocity, which is components 3, 4 and 5 of y.

```
 (dydx[0], dydx[1], dydx[2]) = (y[3], y[4], y[5])
```

The next three components are the derivatives of the velocity vector with respect to the path length. So you should write the "force" components for

```
     dydx[3], dydx[4] and dydx[5]
```

for your field.

Get a G4FieldManager to use your field

In order to inform the Geant4 system that you want it to use your field as the global field, you must do the following steps:

1. Create a Stepper of your choice:

   ```
        yourStepper = new G4ClassicalRK( yourEquationOfMotion );
                  // or if your field is not smooth eg
                  //     new G4ImplicitEuler( yourEquationOfMotion );
   ```

2. Create a chord finder that uses your Field and Stepper. You must also give it a minimum step size, below which it does not make sense to attempt to integrate:

   ```
        yourChordFinder= new G4ChordFinder( yourField,
                                  yourMininumStep, // say 0.01*mm
                                  yourStepper );
   ```

3. Next create a G4FieldManager and give it that chord finder,

   ```
        yourFieldManager= new G4FieldManager();
        yourFieldManager.SetChordFinder(yourChordFinder);
   ```

4. Finally we tell the Geometry that this FieldManager is responsible for creating a field for the detector.

```
    G4TransportationManager::GetTransportationManager()
                          -> SetFieldManager( yourFieldManager );
```

Changes for non-electromagnetic fields

If the field you are interested in simulating is not electromagnetic, another minor modification may be required. The transportation currently chooses whether to propagate a particle in a field or rectilinearly based on whether the particle is charged or not. If your field affects non-charged particles, you must inherit from the G4Transportation and re-implement the part of GetAlongStepPhysicalInteractionLength that decides whether the particles is affected by your force.

In particular the relevant section of code does the following:

```
//  Does the particle have an (EM) field force exerting upon it?
//
if( (particleCharge!=0.0) ){

   fieldExertsForce= this->DoesGlobalFieldExist();
   // Future: will/can also check whether current volume's field is Zero or
   //  set by the user (in the logical volume) to be zero.
}
```

and you want it to ask whether it feels your force. If, for the sake of an example, you wanted to see the effects of gravity on a heavy hypothetical particle, you could say

```


//  Does the particle have my field's force exerted on it?
//
if (particle->GetName() == "VeryHeavyWIMP") {
   fieldExertsForce= this->DoesGlobalFieldExist();  // For gravity
}
```

After doing all these steps, you will be able to see the effects of your force on a particle's motion.

3.3. Particles

3.3.1. Properties of particles

The *G4ParticleDefinition* class contains the properties which characterize individual particles, such as name, mass, charge, spin, and so on. Properties of particles are set during the initialization of each particle type. The default values of these properties are described in each particle class. In the case of heavy nuclei properties may be given by external files. Once initialized, particle properties cannot be changed except for those related to its decay; these are life time, branching ratio of each decay mode and the ``stable'' flag. Geant4 proivides a method to override these properties by using external files.

Properties of nuclei

Individual classes are provided for light nuclei (i.e. deuteron, triton, He3, and He4) with default values of their properties. Other nuclei are dynamically created by requests from processes (and users). *G4IonTable* class handles the creation of such ions. Default properties of nuclei are determined with help of *G4NuclearProperties*.

Users can register a *G4IsotopeTable* to the *G4IonTable*. *G4IsotopeTable* which describes the properties used to create ions. Excitation energy, decay modes, and life times for relatively long-lived nuclei can be obtained by using *G4RIsotopeTable* and data files such as those pointed to by the G4RADIOACTIVEDATA environment variable. *G4IsotopeMagneticMomentTable* provides a table of nuclear magnetic moments using the data file *G4IsotopeMagneticMoment.table*. The environment variable G4IONMAGNETICMOMENT should be set to point to this file.

Changing particle properties

Only in the ``PreInit" phase can properties be modified with the help of the *G4ParticlePropertyTable* class. Particle properties can be overridden with the method

```
G4bool SetParticleProperty(const G4ParticlePropertyData& newProperty)
```

by setting new values in *G4ParticlePropertyData* . In addition, the current particle property values can be extracted to text files by using *G4TextPPReporter* . On the other hand, *G4TextPPRetriever* can change particle properties according to text files.

3.3.2. Adding New Particles

A new particle can be added by creating a new class for it. The new class should be derived from *G4ParticleDefinition*. You can find an example under examples/extended/exoticphysics/monopole. There, the new class for the monopole is defined as follows:

```
class G4Monopole : public G4ParticleDefinition
{
private:
  static G4Monopole*  theMonopole;

  G4Monopole(
       const G4String&     aName,        G4double            mass,
       G4double            width,        G4double            charge,
       G4int               iSpin,        G4int               iParity,
       G4int               iConjugation, G4int               iIsospin,
       G4int               iIsospin3,    G4int               gParity,
       const G4String&     pType,        G4int               lepton,
       G4int               baryon,       G4int               encoding,
       G4bool              stable,       G4double            lifetime,
       G4DecayTable        *decaytable );

public:
   virtual ~G4Monopole();
   static G4Monopole* MonopoleDefinition();
   static G4Monopole* Monopole();
}
```

The static methods above must be defined and implemented so that the new particle instance will be created in the ConstructParticles method of your physics list. New properties may be added if necessary (G4Monopole has a a property for magnetic charge). Values of properties need to be given in the static method as other particle classes.

```
G4Monopole* G4Monopole::MonopoleDefinition(G4double mass, G4int mCharge, G4int eCharge)
{
  if(!theMonopole) {
    theMonopole = new G4Monopole(
               "monopole",         mass,       0.0*MeV,           0,
                     0,               0,             0,
                     0,               0,             0,
               "boson",              0,             0,           0,
                  true,            -1.0,            0);
  }
  return theMonopole;
}
```

3.3.3. Nuclide properties from the Evaluated Nuclear Structure Data File

G4NuclideTable

G4NuclideTable was introduced in Geant4 v10 to provide properties of nuclides. The excitation energy and decay constant of each of 25,497 states are listed in the table and the spin and dipole magnetic moments are available

for some states. The source of these data is ENSDF as of February 2015. In Geant4 v10.02 the hard-coded list of the nuclide states was removed and it now uses only the **ENSDFSTATE.dat**. *G4NuclideTable* fills this array by scanning the file and selecting all nuclides with half-lives greater than 1.0 microsecond.

The user may set the minimum half-life of states that will be selected during the scan of **ENSDFSTATE.dat**. If that value is less than the default of 1 microsecond, more states will be read into *G4NuclideTable*. If it is larger, fewer states will be included. Any nuclide thus read into *G4NuclideTable* will be treated as a track-able particle, which will travel some distance as sampled from its decay time. For each state, *G4NuclideTable* assigns a level number ranging from 0 to the PDG limit of 9. All ground states are assigned the value 0. In a given nucleus, the selected state with the smallest excitation energy is assigned to be level 1, and so on for the subsequent higher selected excitation energy states up to level 8. Note that the ordering of states in a given nucleus will change depending on the minimum half-life value chosen. If more than 8 excited states are found, the excess states are all assigned to level 9. The above action is taken at initialization time and may be invoked in two ways:

- `G4NuclideTable::SetThresholdOfHalfLife (G4double)`
- `/particle/manage/nuclideTable/min_halflife 0.001 ns`

Note that while setting a small minimum half-life will provide a more precise decay chain with more short-lived nuclides, the increased number of states will reduce CPU performance and increase memory footprint.

Adding states

User may add states with user specified values of excitation energy, decay constant, spin and dipole magnetic moment by:

```
AddState (G4int Z, G4int A , G4double excitation_energy, G4double lifetime, G4int ionJ=0,
          G4double ionMu=0.0)
```

This is done at initialization time and all values are read into the *G4NuclideTable* array regardless of the minimum half-life value. These user states are all assigned to level number 9.

In addition to the minimum half-life value, *G4NuclideTable* also allows the user to set the tolerance level of the excitation energy used to identify the state. The default value of the tolerance is 1 eV which the user may modify in one of two ways:

- `G4NuclideTable::SetLevelTolerance(G4double)`
- `/particle/manage/nuclideTable/level_tolerance 1 keV`

This modification is also done during the initialization phase. Note that in adding states a user must specify excited states with a precision that matches the tolerance. Thus, if the tolerance is 1 eV, the level in keV would be given, for example, by 2505.531, otherwise that state will be missed. If it is not desirable to specify levels to such a precision, the tolerance value can be increased. If, however, there are two levels in a nucleus which are closer to one another than the value of the tolerance, one of the levels will be missed. Currently G4RadioactiveDecay and G4PhotoEvporation models share the state information with *G4NuclideTable*. Other models are encouraged to do lifewise.

3.4. Electromagnetic Physics

3.4.1. Introduction

The Geant4 set of electromagnetic (EM) physics processes and models are used in practically all types of simulation applications including high energy and nuclear physics experiments, beam transport, medical physics, cosmic ray interactions and radiation effects in space. In addition to models for low and high energy EM physics for simulation of radiation effects in media, a sub-library of very low energy models was developed within the framework of the Geant4-DNA project, with the goal of simulating radiation effects involving physics and chemistry at the sub-cellular level.

In the early stages of Geant4, low and high energy EM processes were developed independently, with the result that these processes could not be used in the same run. To resolve this problem, the interfaces were unified so that the standard, muons, highenergy, lowenergy, and dna EM physics sub-packages now follow the same design. Migration to this common design resulted in an improvement of overall CPU performance, and made it possible to provide several helper classes which are useful for a variety of user applications (for example `G4EmCalculator`).

3.4.2. General design

The electromagnetic processes of Geant4 follow the basic interfaces:

- `G4VEnergyLossProcess`;
- `G4VEmProcess`;
- `G4VMultipleScattering`.

The class diagram is shown in Figure 3.1.

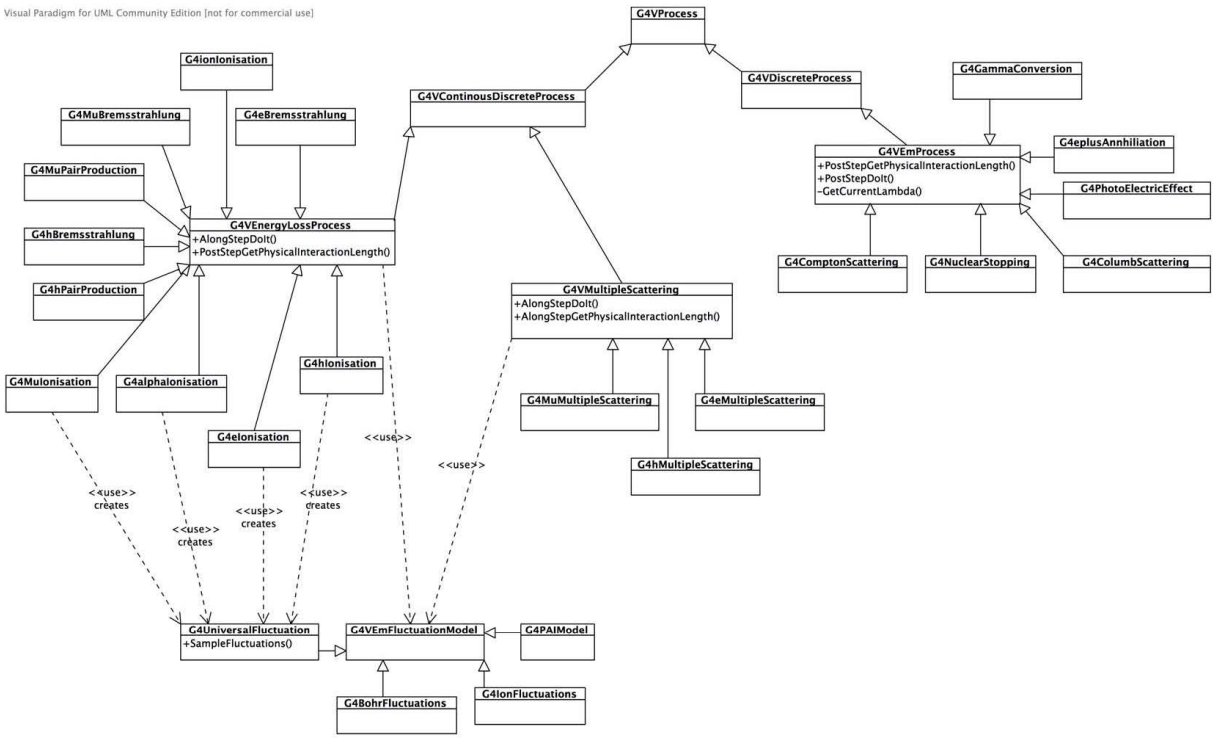

Figure 3.1. Design of EM physics processes.

These common interfaces for all EM sub-packages enabled the full migration of EM classes to multi-threading without significant modification of existing physics model codes. Initialization of the energy loss, stopping power and cross section tables is carried out only once in the master thread at the beginning of simulation. These tables are shared between threads in run time.

3.4.3. Electromagnetic processes

These base classes provide all management work of initialisation of processes, creation and filling of physics tables, and generic run-time actions. Concrete process classes are responsible for the initialisation of parameters

and defining the set of models for the process. t is strongly recommented to use existing processes and not create a new one for each new model. Here is a list of main EM processes:

- `G4PhotoelectricEffect`;
- `G4ComptonScattering`;
- `G4GammaConversion`;
- `G4GammaConversion`;
- `G4RayleighScattering`;
- `G4eIonisation`;
- `G4eBremsstrahlung`;
- `G4hIonisation`;
- `G4MuIonisation`;
- `G4hIonisation`;
- `G4MuBremsstrahlung`;
- `G4eMultipleScattering`;
- `G4MuMultipleScattering`.

More processes are provided in lowenergy, polarisation, and adjoint sub-libraries. In some specific cases, interfaces described above are not applicable and the high level interface `G4VProcess` is used.

Any concrete physics process class may need custom parameters. It is recommended to define following parameters specific to the class in the class constructor:

- process sub-type;
- buildTables flag;
- secondary particle type;
- min/max energy of cross section tables;
- number of bins in tables;
- flag to force zero cross section in the low edge of a table.

Any EM process should implement following methods:

- `IsApplicable(const G4ParticleDefinition& p)`
- `PrintInfo()`

Main initialisation of a process is performed by initialisation methods:

- `InitialiseEnergyLossProcess(const G4ParticleDefinition* part,const G4ParticleDefinition*basePart)`
- `InitialiseProcess(const G4ParticleDefinition*)`

In these methods a default set of EM models and their energy intervals of applicability may be defined. It is strongly recommended that a process class cannot change EM parameters in `G4EmParameters` class or status of the de-excitation module. EM parameters can be modifided in physics lists or via UI commands.

3.4.4. Electromagnetic models

Concrete physics models are implemented via EM model interfaces:

- `G4VEmModel`;
- `G4VMscModel`.

In the majority of use-cases when new EM physics is needed, it is enough to create only a new model class and use it in the existing EM process class. A new model may be added to an existing process using `AddEmModel(G4int, G4VEmModel*, G4Region*)` method. The class diagram is shown in Figure 3.2.

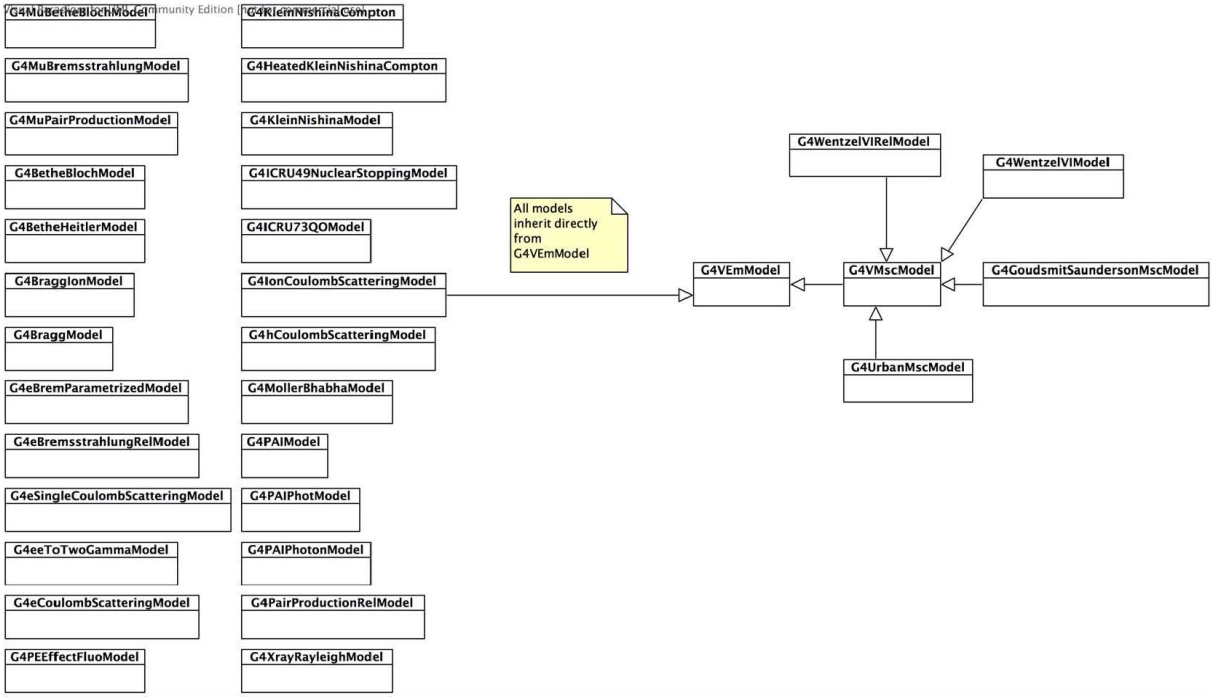

Figure 3.2. Design of EM physics models.

The base class `G4VEmModel` has a lot of virtual methods mostly with the default implementation. Pure virtual methods to be implemented in each model class are following:

- `Initialise(const G4ParticleDefinition*, const G4DataVector&);`
- `SampleSecondaries(std::vector<G4DynamicParticle*>*,const G4MaterialCutsCouple*,const G4DynamicParticle*,G4double tmin=0.0,G4double tmax=DBL_MAX)`.

Any model may have its own data structure or physics table. It is optimal to share data between threads in multi-threaded mode. For that at initialisation it is possible to check if initialisation is performed in master thread using a method `IsMaster()`. In the master thread data of the model should be initilized. A model initialisation may be performed between runs. In that case, potentially materials and cuts may be changed, so re-initialisation of model data should be foreseen. In worker thread it is possible to access shared data implementing virtual method `InitialiseLocal(const G4ParticleDefinition*, G4VEmModel* masterModel)`. It is strongly recommended that a model class cannot change EM parameters in `G4EmParameters` class or status of the de-excitation module. EM parameters can be modifided in physics lists or via UI commands.

3.5. Hadronic Physics

3.5.1. Introduction

Optimal exploitation of hadronic final states played a key role in successes of all recent collider experiment in HEP, and the ability to use hadronic final states will continue to be one of the decisive issues during the analysis phase of the LHC experiments. Monte Carlo programs like Geant4 facilitate the use of hadronic final states, and have been developed for many years.

We give an overview of the Object Oriented frameworks for hadronic generators in Geant4, and illustrate the physics models underlying hadronic shower simulation on examples, including the three basic types of modeling; data-driven, parametrisation-driven, and theory-driven modeling, and their possible realisations in the Object Oriented component system of Geant4. We put particular focus on the level of extendibility that can and has been achieved by our Russian dolls approach to Object Oriented design, and the role and importance of the frameworks in a component system.

3.5.2. Principal Considerations

The purpose of this section is to explain the implementation frameworks used in and provided by Geant4 for hadronic shower simulation as in the 1.1 release of the program. The implementation frameworks follow the Russian dolls approach to implementation framework design. A top-level, very abstracting implementation framework provides the basic interface to the other Geant4 categories, and fulfils the most general use-case for hadronic shower simulation. It is refined for more specific use-cases by implementing a hierarchy of implementation frameworks, each level implementing the common logic of a particular use-case package in a concrete implementation of the interface specification of one framework level above, this way refining the granularity of abstraction and delegation. This defines the Russian dolls architectural pattern. Abstract classes are used as the delegation mechanism [1] .

All framework functional requirements were obtained through use-case analysis. In the following we present for each framework level the compressed use-cases, requirements, designs including the flexibility provided, and illustrate the framework functionality with examples. All design patterns cited are to be read as defined in [Gamma1995].

3.5.3. Level 1 Framework - processes

There are two principal use-cases of the level 1 framework. A user will want to choose the processes used for his particular simulation run, and a physicist will want to write code for processes of his own and use these together with the rest of the system in a seamless manner.

Requirements

1. Provide a standard interface to be used by process implementations.
2. Provide registration mechanisms for processes.

Design and interfaces

Both requirements are implemented in a sub-set of the tracking-physics interface in Geant4}. The class diagram is shown in Figure 3.3.

[1] The same can be achieved with template specialisations with slightly improved CPU performance but at the cost of significantly more complex designs and, with present compilers, significantly reduced portability.

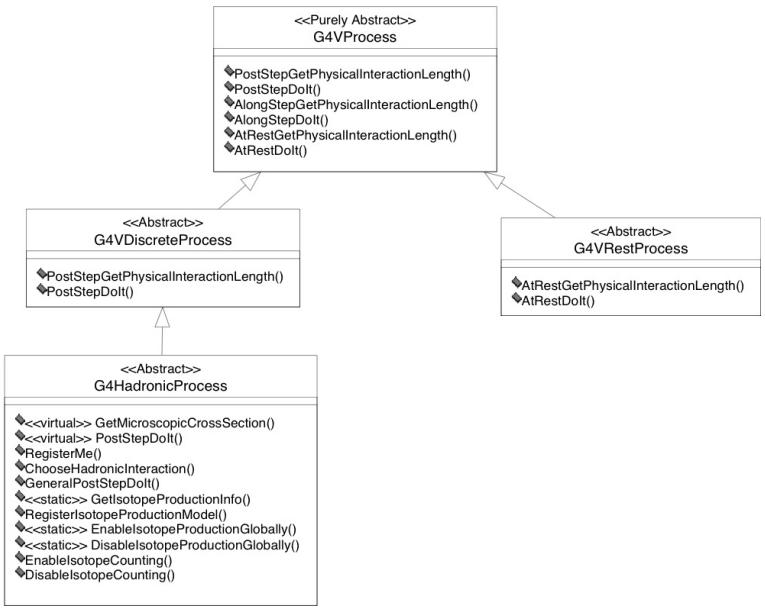

Figure 3.3. Level 1 implementation framework of the hadronic category of GEANT4.

All processes have a common base-class `G4VProcess`, from which a set of specialised classes are derived. Two of them are used as base classes for hadronic processes at rest and in flight (`G4VDiscreteProcess`), and for processes like radioactive decay where the same implementation can represent both these extreme cases (`G4VRestDiscreteProcess`).

Each of these classes declares two types of methods; one for calculating the time to interaction or the physical interaction length, allowing tracking to request the information necessary to decide on the process responsible for final state production, and one to compute the final state. These are pure virtual methods, and have to be implemented in each individual derived class, as enforced by the compiler.

Note on at-rest processes: starting with Geant4 version `9.6` - when the Bertini and Fritiof final-state models have been extended down to zero kinetic energy and used also for simulating the nuclear capture at-rest - the at-rest processes derive from `G4HadronicProcess`, hence from `G4VDiscreteProcess`, instead than from `G4VRestProcess` as in the initial design of at-rest processes. This requires some adaptation a discrete process to handle an at-rest one using top level interface `G4VProcess`. A different solution, under consideration but not yet implemented, would be instead to have `G4HadronicProcess` inheriting from `G4VRestDiscreteProcess`: in this way, `G4HadronicProcess`, and therefore any theory-driven final-state model, could be deployed for any kind of hadronic process, including capture-at-rest processes and radioactive decays.

Framework functionality

The functionality provided is through the use of process base-class pointers in the tracking-physics interface, and the `G4Process-Manager`. All functionality is implemented in abstract, and registration of derived process classes with the `G4Process-Manager` of an individual particle allows for arbitrary combination of both Geant4 provided processes, and user-implemented processes. This registration mechanism is a modification on a Chain of Responsibility. It is outside the scope of the current paper, and its description is available from G4Manual.

3.5.4. Level 2 Framework - Cross Sections and Models

At the next level of abstraction, only processes that occur for particles in flight are considered. For these, it is easily observed that the sources of cross sections and final state production are rarely the same. Also, different sources will come with different restrictions. The principal use-cases of the framework are addressing these commonalities. A user might want to combine different cross sections and final state or isotope production models as provided by Geant4, and a physicist might want to implement his own model for particular situation, and add cross-section data sets that are relevant for his particular analysis to the system in a seamless manner.

Requirements

1. Flexible choice of inclusive scattering cross-sections.
2. Ability to use different data-sets for different parts of the simulation, depending on the conditions at the point of interaction.
3. Ability to add user-defined data-sets in a seamless manner.
4. Flexible, unconstrained choice of final state production models.
5. Ability to use different final state production codes for different parts of the simulation, depending on the conditions at the point of interaction.
6. Ability to add user-defined final state production models in a seamless manner.
7. Flexible choice of isotope production models, to run in parasitic mode to any kind of transport models.
8. Ability to use different isotope production codes for different parts of the simulation, depending on the conditions at the point of interaction.
9. Ability to add user-defined isotope production models in a seamless manner.

Design and interfaces

The above requirements are implemented in three framework components, one for cross-sections, final state production, and isotope production each. The class diagrams are shown in Figure 3.4 for the cross-section aspects, Figure 3.5 for the final state production aspects, and figure Figure 3.6 for the isotope production aspects.

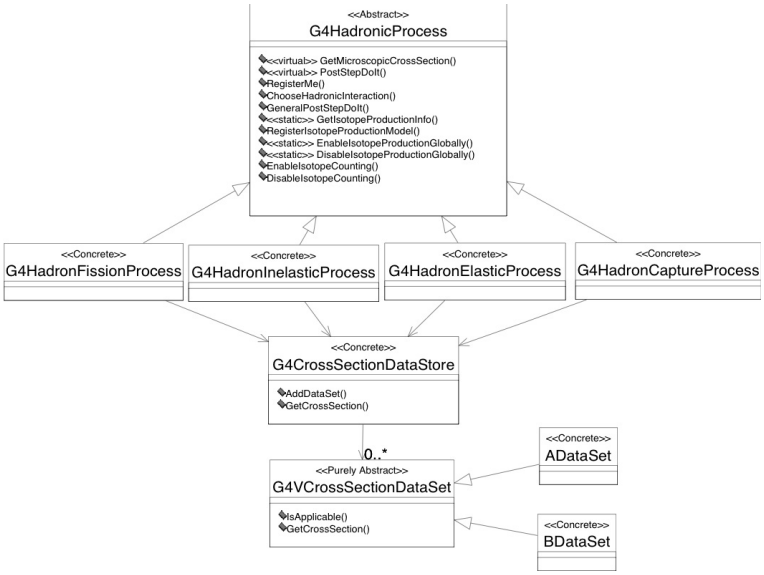

Figure 3.4. Level 2 implementation framework of the hadronic category of Geant4; cross-section aspect.

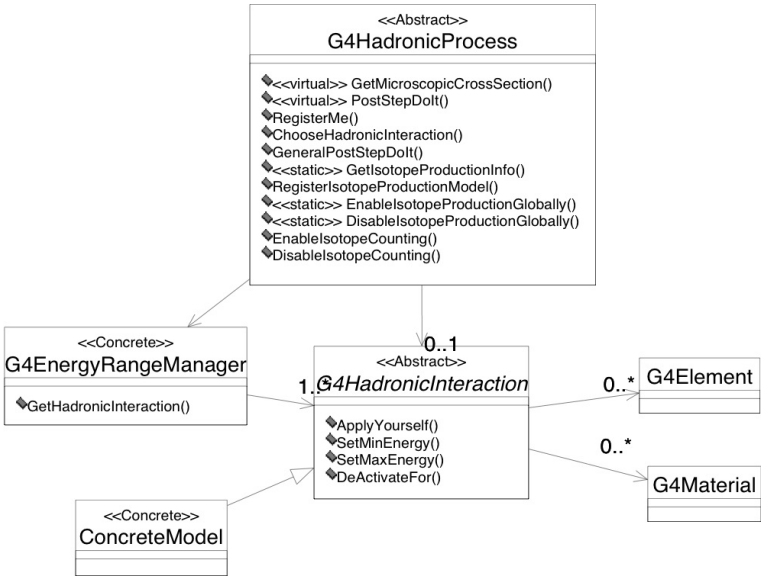

Figure 3.5. Level 2 implementation framework of the hadronic category of Geant4; final state production aspect.

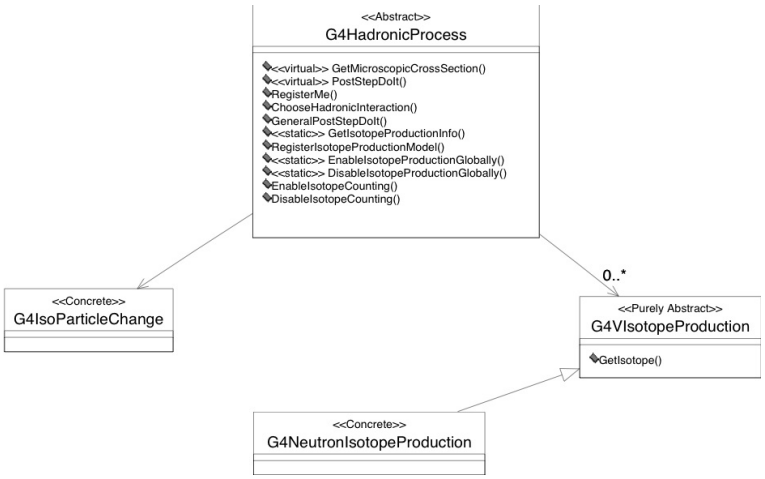

Figure 3.6. Level 2 implementation framework of the hadronic category of Geant4; isotope production aspect

The three parts are integrated in the `G4Hadronic-Process` class, that serves as base-class for all hadronic processes of particles in flight.

Cross-sections

Each hadronic process is derived from `G4Hadronic-Process}`, which holds a list of ``cross section data sets''. The term ``data set'' is representing an object that encapsulates methods and data for calculating total cross sections for a given process in a certain range of validity. The implementations may take any form. It can be a simple equation as well as sophisticated parameterisations, or evaluated data. All cross section data set classes are derived from the abstract class `G4VCrossSection-DataSet}`, which declares methods that allow the process inquire, about the applicability of an individual data-set through `IsApplicable(const G4DynamicParticle*, const G4Element*)`, and to delegate the calculation of the actual cross-section value through `GetCrossSection(const G4DynamicParticle*, const G4Element*)`.

Final state production

The `G4HadronicInteraction` base class is provided for final state generation. It declares a minimal interface of only one pure virtual method: `G4VParticleChange* ApplyYourself(const G4Track &,`

G4Nucleus &)}. G4HadronicProcess provides a registry for final state production models inheriting from G4Hadronic-Interaction. Again, final state production model is meant in very general terms. This can be an implementation of a quark gluon string model [QGSM], a sampling code for ENDF/B data formats [ENDFForm], or a parametrisation describing only neutron elastic scattering off Silicon up to 300~MeV.

Isotope production

For isotope production, a base class (G4VIsotope-Production) is provided. It declares a method G4IsoResult * GetIsotope(const G4Track &, const G4Nucleus &) that calculates and returns the isotope production information. Any concrete isotope production model will inherit from this class, and implement the method. Again, the modeling possibilities are not limited, and the implementation of concrete production models is not restricted in any way. By convention, the GetIsotope method returns NULL, if the model is not applicable for the current projectile target combination.

Framework functionality:

Cross Sections

G4HadronicProcess provides registering possibilities for data sets. A default is provided covering all possible conditions to some approximation. The process stores and retrieves the data sets through a data store that acts like a FILO stack (a Chain of Responsibility with a First In Last Out decision strategy). This allows the user to map out the entire parameter space by overlaying cross section data sets to optimise the overall result. Examples are the cross sections for low energy neutron transport. If these are registered last by the user, they will be used whenever low energy neutrons are encountered. In all other conditions the system falls back on the default, or data sets with earlier registration dates. The fact that the registration is done through abstract base classes with no side-effects allows the user to implement and use his own cross sections. Examples are special reaction cross sections of K^0-nuclear interactions that might be used for #/#' analysis at LHC to control the systematic error.

Final state production

The G4HadronicProcess class provides a registration service for classes deriving from G4Hadronic-Interaction, and delegates final state production to the applicable model. G4Hadronic-Interactionprovides the functionality needed to define and enforce the applicability of a particular model. Models inheriting from G4Hadronic-Interaction can be restricted in applicability in projectile type and energy, and can be activated/deactivated for individual materials and elements. This allows a user to use final state production models in arbitrary combinations, and to write his own models for final state production. The design is a variant of a Chain of Responsibility. An example would be the likely CMS scenario - the combination of low energy neutron transport with a quantum molecular dynamics [QMD], invariant phase space decay [CHIPS], and fast parametrised models for calorimeter materials, with user defined modeling of interactions of spallation nucleons with the most abundant tracker and calorimeter materials.

Isotope production

The G4HadronicProcess by default calculates the isotope production information from the final state given by the transport model. In addition, it provides a registering mechanism for isotope production models that run in parasitic mode to the transport models and inherit from G4VIsotope-Production. The registering mechanism behaves like a FILO stack, again based on Chain of Responsibility. The models will be asked for isotope production information in inverse order of registration. The first model that returns a non-NULL value will be applied. In addition, the G4Hadronic-Process provides the basic infrastructure for accessing and steering of isotope-production information. It allows to enable and disable the calculation of isotope production information globally, or for individual processes, and to retrieve the isotope production information through the G4IsoParticleChange * GetIsotopeProductionInfo()} method at the end of each step. The G4HadronicProcess is a finite state machine that will ensure the GetIsotope-ProductionInfo returns a non-zero value only at the first call after isotope production occurred. An example of the use of this functionality is the study of activation of a Germanium detector in a high precision, low background experiment.

3.5.5. Level 3 Framework - Theoretical Models

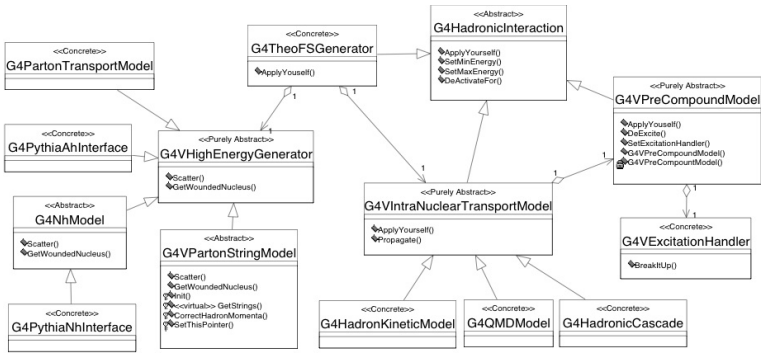

Figure 3.7. Level 3 implementation framework of the hadronic category of Geant4; theoretical model aspect.

Geant4 provides at present one implementation framework for theory driven models. The main use-case is that of a user wishing to use theoretical models in general, and to use various intra-nuclear transport or pre-compound models together with models simulating the initial interactions at very high energies.

Requirements

1. Allow to use or adapt any string-parton or parton transport [VNI],
2. Allow to adapt event generators, ex. [PYTHIA7], state production in shower simulation.
3. Allow for combination of the above with any intra-nuclear transport (INT).
4. Allow stand-alone use of intra-nuclear transport.
5. Allow for combination of the above with any pre-compound model.
6. Allow stand-alone use of any pre-compound model.
7. Allow for use of any evaporation code.
8. Allow for seamless integration of user defined components for any of the above.

Design and interfaces

To provide the above flexibility, the following abstract base classes have been implemented:

- `G4VHighEnergyGenerator`
- `G4VIntranuclearTransportModel`
- `G4VPreCompoundModel`
- `G4VExcitationHandler`

In addition, the class `G4TheoFS-Generator` is provided to orchestrate interactions between these classes. The class diagram is shown in Figure 3.7.

`G4VHighEnergy-Generator` serves as base class for parton transport or parton string models, and for Adapters to event generators. This class declares two methods, `Scatter`, and `GetWoundedNucleus`.

The base class for INT inherits from `G4Hadronic-Interaction`, making any concrete implementation usable as a stand-alone model. In doing so, it re-declares the `ApplyYourself` interface of `G4Hadronic-Interaction`, and adds a second interface, `Propagate`, for further propagation after high energy interactions. `Propagate` takes as arguments a three-dimensional model of a wounded nucleus, and a set of secondaries with energies and positions.

The base class for pre-equilibrium decay models, `G4VPre-CompoundModel`, inherits from `G4Hadronic-Interaction`, again making any concrete implementation usable as stand-alone model. It adds a pure virtual

`DeExcite` method for further evolution of the system when intra-nuclear transport assumptions break down. `DeExcite` takes a `G4Fragment`, the Geant4 representation of an excited nucleus, as argument.

The base class for evaporation phases, `G4VExcitation-Handler`, declares an abstract method, `BreakIt-UP()`, for compound decay.

Framework functionality

The `G4TheoFSGenerator` class inherits from `G4Hadronic-Interaction`, and hence can be registered as a model for final state production with a hadronic process. It allows a concrete implementation of `G4VIntranuclear-TransportModel` and `G4VHighEnergy-Generator` to be registered, and delegates initial interactions, and intra-nuclear transport of the corresponding secondaries to the respective classes. The design is a complex variant of a Strategy. The most spectacular application of this pattern is the use of parton-string models for string excitation, quark molecular dynamics for correlated string decay, and quantum molecular dynamics for transport, a combination which promises to result in a coherent description of quark gluon plasma in high energy nucleus-nucleus interactions.

The class `G4VIntranuclearTransportModel` provides registering mechanisms for concrete implementations of `G4VPreCompound-Model`, and provides concrete intra-nuclear transports with the possibility of delegating pre-compound decay to these models.

`G4VPreCompoundModel` provides a registering mechanism for compound decay through the `G4VExcitation-Handler` interface, and provides concrete implementations with the possibility of delegating the decay of a compound nucleus.

The concrete scenario of `G4TheoFS-Generator` using a dual parton model and a classical cascade, which in turn uses an exciton pre-compound model that delegates to an evaporation phase, would be the following: `G4TheoFS-Generator` receives the conditions of the interaction; a primary particle and a nucleus. It asks the dual parton model to perform the initial scatterings, and return the final state, along with the by then damaged nucleus. The nucleus records all information on the damage sustained. `G4TheoFS-Generator` forwards all information to the classical cascade, that propagates the particles in the already damaged nucleus, keeping track of interactions, further damage to the nucleus, etc.. Once the cascade assumptions break down, the cascade will collect the information of the current state of the hadronic system, like excitation energy and number of excited particles, and interpret it as a pre-compound system. It delegates the decay of this to the exciton model. The exciton model will take the information provided, and calculate transitions and emissions, until the number of excitons in the system equals the mean number of excitons expected in equilibrium for the current excitation energy. Then it hands over to the evaporation phase. The evaporation phase decays the residual nucleus, and the Chain of Command rolls back to `G4TheoFS-Generator`, accumulating the information produced in the various levels of delegation.

3.5.6. Level 4 Frameworks - String Parton Models and Intra-Nuclear Cascade

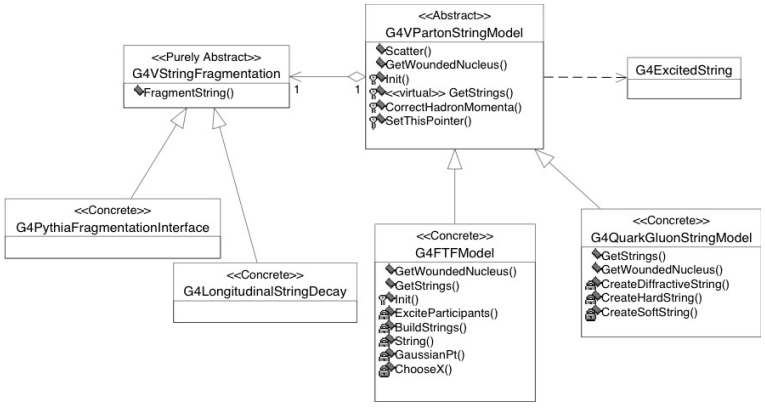

Figure 3.8. Level 4 implementation framework of the hadronic category of Geant4; parton string aspect.

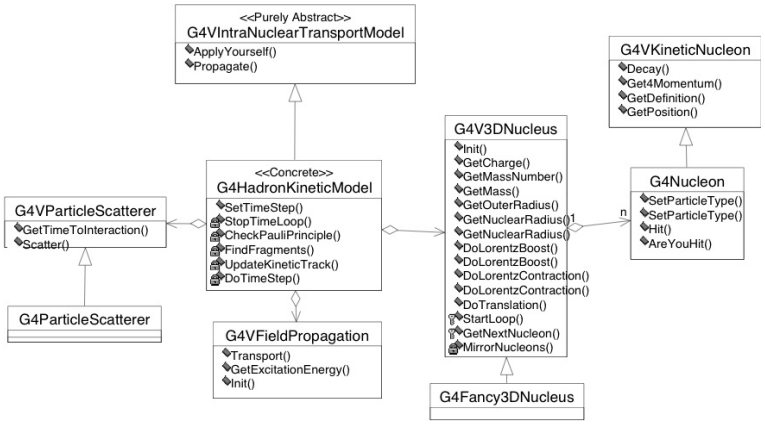

Figure 3.9. Level 4 implementation framework of the hadronic category of Geant4; intra-nuclear transport aspect.

The use-cases of this level are related to commonalities and detailed choices in string-parton models and cascade models. They are use-cases of an expert user wishing to alter details of a model, or a theoretical physicist, wishing to study details of a particular model.

Requirements

1. Allow to select string decay algorithm
2. Allow to select string excitation.
3. Allow the selection of concrete implementations of three-dimensional models of the nucleus
4. Allow the selection of concrete implementations of final state and cross sections in intra-nuclear scattering.

Design and interfaces

To fulfil the requirements on string models, two abstract classes are provided, the `G4VParton-StringModel` and the `G4VString-Fragmentation`. The base class for parton string models, `G4VParton-StringModel`, declares the `GetStrings()` pure virtual method. `G4VString-Fragmentation`, the pure abstract base class for string fragmentation, declares the interface for string fragmentation.

To fulfill the requirements on intra-nuclear transport, two abstract classes are provided, `G4V3DNucleus`, and `G4VScatterer`. At this point in time, the usage of these intra-nuclear transport related classes by concrete codes is not enforced by designs, as the details of the cascade loop are still model dependent, and more experience has to be gathered to achieve standardisation. It is within the responsibility of the implementers of concrete intra-nuclear transport codes to use the abstract interfaces as defined in these classes.

The class diagram is shown in Figure 3.8 for the string parton model aspects, and in Figure 3.9 for the intra-nuclear transport.

Framework functionality

Again variants of Strategy, Bridge and Chain of Responsibility are used. `G4VParton-StringModel` implements the initialisation of a three-dimensional model of a nucleus, and the logic of scattering. It delegates secondary production to string fragmentation through a `G4VString-Fragmentation` pointer. It provides a registering service for the concrete string fragmentation, and delegates the string excitation to derived classes. Selection of string excitation is through selection of derived class. Selection of string fragmentation is through registration.

3.5.7. Level 5 Framework - String De-excitation}

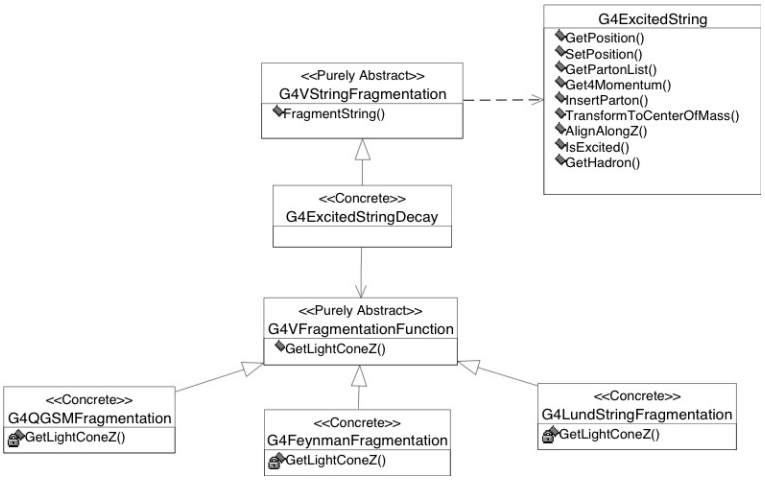

Figure 3.10. Level 5 implementation framework of the hadronic category of Geant4; string fragmentation aspect.

The use-case of this level is that of a user or theoretical physicist wishing to understand the systematic effects involved in combining various fragmentation functions with individual types of string fragmentation. Note that this framework level is meeting the current state of the art, making extensions and changes of interfaces in subsequent releases likely.

Requirements

1. Allow the selection of fragmentation function.

Design and interfaces

A base class for fragmentation functions, `G4VFragmentation-Function}`, is provided. It declares the `Get-LightConeZ()` interface.

Framework functionality

The design is a basic Strategy. The class diagram is shown in Figure 3.10. At this point in time, the usage of the `G4VFragmentation-Function` is not enforced by design, but made available from the `G4VString-Fragmentation` to an implementer of a concrete string decay. `G4VString-Fragmentation` provides a registering mechanism for the concrete fragmentation function. It delegates the calculation of z_f of the hadron to split of the string to the concrete implementation. Standardisation in this area is expected.

3.5.8. Creating Your Own Hadronic Process

For some applications Geant4 might not provide the most appropriate physics implementantion or, in fact, any physics implementation at all. In such cases, it is up to the user to develop the necessary processes, models and cross sections and integrate them into his version of the Geant4 toolkit. The user's process, model or cross section may then be used in a physics list to replace some of those already provided by the toolkit. This modularity requires that user classes be derived from a set of base classes which have been provided to aid integration with the toolkit and to spare the user from consideration of many details not related to the physics at hand.

Processes communicate with Geant4 tracking, telling it where or when an interaction is supposed to occur, and what is supposed to happen at that point. A hadronic process may be implemented directly, or through the use af a framework of classes that modularize physics functionality, make available several utilities and reduce un-neccesary code duplication. In the latter, recommended, approach the user must in general develop as many as three classes: a process, a cross section and a model. Instances of the cross section and model classes must then be assigned to the process. In practice it is usually necessary to develop only a model class or a cross section class, since a number of processes are already provided by Geant4. Before writing any code, users should check that Geant4 has not already povided the necessary models, cross sections or processes.

3.5.8.1. Developing a new hadronic model

A hadronic model is repsonsible for the generation of a set of final state four-vectors, given an initial projectile and target. New models should derive from the *G4HadronicInteraction* base class and at least two methods in this class must be implemented:

```
virtual G4HadFinalState*
ApplyYourself(const G4HadProjectile& aProjectile, G4Nucleus& targetNucleus)
```

which is responsible for generating the final state of the interaction including the specification of all particle types and four-momenta, and

```
virtual G4bool IsApplicable(const G4HadProjectile& aProjectile, G4Nucleus& targetNucleus)
```

which is responsible for checking that the incident particle type and energy, and the Z and A of the target nucleus, can be adequately handled by the model. *G4HadronicInteraction* provides a number of utilities to aid in the implementation of these methods.

When implementing *ApplyYourself()*, the *Get()* methods of the *G4HadProjectile* and *G4Nucleus* classes provide all the initial state information necessary for the generation of the final state. For *G4HadProjectile*, *GetDefinition()* provides the particle type, and *Get4Momentum()* provides the total energy and momentum. For *G4Nucleus*, *GetZ_asInt()*, *GetN_asInt()* and *GetA_asInt()* provide Z, N and A, while *AtomicMass()* provides the mass. Additional utility methods are available for both *G4HadProjectile* and *G4Nucleus* .

Coordinate systems

The inputs to the model assume that the incident particle (*G4HadProjectile*) travels along the z axis and interacts with the target (*G4Nucleus*) which is at rest in the lab frame. Before invoking the *ApplyYourself()* method, the process rotates the direction of the projectile to be along the z axis and then performs the inverse rotation on the final state particles produced by the model.

The model must perform two additional transformations: into the CM frame, and back out of it after the interaction is complete.

Writing the ApplyYourself() method

It is thus the model developer's responsibility to:

- boost the projectile and target into the CM frame using the necessary Lorentz transformations,
- perform all calculations required to generate the final state set of particles,
- boost the final state particles back to the lab frame with the inverse transformation, and
- send the final state particles to the process by filling *G4HadFinalState* and setting its status.

Step 4) is accomplished by using the various *Get()* and *Set()* methods provided by the class *G4HadFinalState*. The developer must also decide whether the original projectile survives the interaction, disappears or is suspended. This is done with the *SetStatusChange()* method. If the projectile survives, the change in its energy and momentum must be set with the provided methods and it must be flagged as "isAlive". If the particle disappears it must be flagged as "stopAndKill".

Geant4 provides a large number of Lorentz transformation tools which may be used to complete steps 1) and 3).

How step 2) is accomplished is entirely up to the developer. This could be as simple as a look-up table which assigns a final state to an initial state, or as complex as a theoretical high energy generator. Typically, the user will have to provide methods of sampling final state multiplicities, energies and angles using random number generators provided by Geant4. For example, the cosine of the polar angle of an isotropic angular distribution could be sampled as follows:

```
 G4double cosTheta = 2.*G4UniformRandom() - 1.;
```

The developer must also see to it that the model conserves energy and momentum. Currently the hadronic framework checks that final states do not exceed reasonably small limits of non-conservation.

Using the hadronic framework

For complex models it is recommended that the user become familiar with the Geant4 hadronic framework. This is covered in detail in the chapter on Extended Functionality in this manual. The framework uses the object-oriented principles of abstraction and re-use to provide a number of services to the developer. The part of the framework used will depend on the type of model. For example, high energy models can take advantage of already-developed string excitation and decay functions and medium energy models can use the intra-nuclear propagation base class and the nuclear de-excitation handler.

Writing the IsApplicable() method

The is a straightforward, but important, method. Most models are quite specific in their range of use and the developer must codify this. It is recommended that this method test for ranges of projectile energy, particle type and target atomic number and weight, and return false when these ranges are exceeded.

3.5.8.2. Developing a new cross section set

New cross section sets should derive from *G4VCrossSectionDataSet* . This class serves as a container of cross section data and provides a number of access methods that must be implemented by the developer. The essential methods are:

```
G4double GetElementCrossSection(const G4DynamicParticle*, G4int Z)
```

which retrieves element-based cross sections,

```
G4double GetIsoCrossSection(const G4DynamicParticle*, G4int Z, G4int A)
```

which retrieves isotope-based cross sections,

```
G4bool IsElementApplicable(const G4DynamicParticle*, G4int Z)
```

which sets the Z range of the element-based data set,

```
G4bool IsIsoApplicable(const G4DynamicParticle*, G4int, G4int A)
```

which sets the Z and A range of the isotope-based data set, and

```
SetMinKinEnergy(G4double)
SetMaxKinEnergy(G4double)
GetMinKinEnergy()
and GetMaxKinEnergy()
```

for defining the applicable energy range of the data set.

3.5.8.3. Developing a new hadronic process

As mentioned above, it is preferable to add new physics in terms of a model, and assign the model to an existing process, rather than develop a new, specific process. Under certain circumstances though, a directly implemented process may be necessary. In that case it must derive from G4HadronicProcess and three methods of that class must be implemented:

```
virtual G4VParticleChange* PostStepDoIt(const G4Track&, const G4Step&) ,
virtual G4bool IsApplicable(const G4ParticleDefinition&) , and
G4double GetMeanFreePath(const G4Track& aTrack, G4double, G4ForceCondition*).
```

PostStepDoIt() is responsible for generating the final state of an interaction given the track and step information. It must update the state of the track, flagging it as "Alive", "StopButAlive", "StopAndKill", "KillTrackAndSecondaries", "Suspend", or "PostponeToNextEvent". It is roughly analagous to the *ApplyYourself()* method in models.

IsApplicable() serves the same purpose in processes as it does in models.

GetMeanFreePath() gets the cross section as a function of particle type, energy, and target material, and converts it to a mean free path, which is in turn passed on to the tracking. This method can be quite simple:

```
G4double particle = aTrack.GetDynamicParticle();
G4double material = aTrack.GetMaterial();
return factor/theCrossSectionDataStore->GetCrossSection(particle, material);
```

provided an appropriate cross section data set is already available in the data store.

The above discussion refers to in-flight processes. At-rest hadronic processes do not currently derive from *G4HadronicProcess*, but from *G4VRestProcess* or *G4VRestDiscreteProcess*. As such they do not employ the full hadronic framework and must be implemented directly without models. The methods to be implemented are similar to those in the in-flight case:

```
virtual G4VParticleChange* AtRestDoIt(const G4Track&, const G4Step&) , and
G4bool IsApplicable(const G4ParticleDefinition&) .
```

3.6. Generic Event Biasing

3.6.1. Introduction

This section presents the generic biasing classes which have been introduced since release 10.0. Theses classes are meant to virtually allow any type of process-level based biasing.

In 10.0 and 10.1, only `PostStep` biasing actions are possible, which allows:

- Physics process biasing of:
  - occurence law, ie biasing acting on the process `PostStepGetPhysicalInteractionLength(...)` level;
  - final state production, ie biasing acting on the process `PostStepDoIt(...)`
- Biasing of non-physics type:
  - Where by that we mean biasing actions which act on particles, but are not modifying a physics process behavior
  - Splitting and killing are the important example of such cases

3.6.2. Design of Generic Biasing

We have decided to split the *actual biasing actions* (change probability occurence of a process, change its final state generation, split, kill, etc...) from the *decisions* about these actions to be taken. The reason for this is that several biasing actions are often needed to build one biasing technique, these actions having to be selected along some specific logic of the technique.

For example, a technique like the "force collision" of MCNP involves a splitting, a force interaction of one copy in the volume and a force free flight (under zero weight) of the other copy in the volume.

The classes which provides the interfaces for these biasing actions and decisions are respectively:

- `G4VBiasingOperation`
- `G4VBiasingOperator`

A third class, `G4BiasingProcessInterface` provides the interface between these classes and the stepping.

3.6.2.1. The `G4VBiasingOperation` Interface Class

`G4VBiasingOperation` defines two types of methods:

- methods for physics-based biasing:
  - `virtual const G4VBiasingInteractionLaw* ProvideOccurenceBiasingInteractionLaw( const G4BiasingProcessInterface* /* callingProcess */, G4ForceCondition& /* proposeForceCondition */ ) = 0;`
    - which is discussed in more details below (Section 3.6.3)
  - `virtual G4VParticleChange* ApplyFinalStateBiasing( const G4BiasingProcessInterface* /* callingProcess */, const G4Track* /* track */, const G4Step* /* step */, G4bool& /* forceBiasedFinalState */) = 0;`

    - which is meant for final state biasing.
    - the biasing operation gets called (by the `G4BiasingProcessInterface callingProcess`) and has to return a biased particle change. This one should take care of providing the proper weight values for the biasing applied.
- methods for non-physics biasing:
  - `virtual G4double DistanceToApplyOperation( const G4Track* /* track */, G4double /* previousStepSize */, G4ForceCondition* /* condition */) = 0;`
    - which returns the distance at which the operation must be applied, and returns also the force condition for this.
    - the returned values (distance, force condition) are returned to the stepping manager by the the `G4BiasingProcessInterface callingProcess` to make the biasing opetation to compete for limiting the step.
  - `virtual G4VParticleChange* GenerateBiasingFinalState( const G4Track* /* track */, const G4Step* /* step */) = 0;`
    - called if the operation has limited the step.
    - must return the final state generated by the operation.

3.6.2.2. The `G4VBiasingOperator` Interface Class

The `G4VBiasingOperator` class is meant to define the interface to pilot biasing operations. It selects biasing operations or sequences of biasing operations to build up the logic of a specific biasing technique. It is messaged by each `G4BiasingProcessInterface callingProcess` instance, which pass their "identity" through their own pointer values. The operator has hence all the necessary information for taking decisions.

In addition, since 10.1, the `G4BiasingProcessInterface callingProcess` instances do "anticipated" calls to their underneath wrapped physics process (if any) : the first of the `G4BiasingProcessInterface callingProcess` instance, trigger calls to all `PostStepGetPhysicalInteractionLength(...)` methods of biased processes. In this way, all process cross-sections have been updated before the first call to the biasing operator. This one has hence all ready-to-use physics information the first it is messaged in the step by the first `G4BiasingProcessInterface callingProcess` instance.

The `G4VBiasingOperator` class defines the following interface:

- `virtual G4VBiasingOperation* ProposeNonPhysicsBiasingOperation( const G4Track* track, const G4BiasingProcessInterface* callingProcess ) = 0;`
  - This method is called by each `G4BiasingProcessInterface` instance which does not hold/wrap a physics process.
  - The `G4VBiasingOperation` returned will have its `DistanceToApplyOperation(...)` and `GenerateBiasingFinalState(...)` methods called at proper times (post step GPIL and post step DoIt times, respectively) by the `G4BiasingProcessInterface callingProcess`.
- `virtual G4VBiasingOperation* ProposeOccurenceBiasingOperation( const G4Track* track, const G4BiasingProcessInterface* callingProcess ) = 0;`
  - The `G4VBiasingOperation` returned will have its `ProvideOccurenceBiasingInteraction-Law(...)` called to get the biased interaction law to be used.
  - This method is called at the `PostStepGetPhysicalInteractionLength(...)` level.
- `virtual G4VBiasingOperation* ProposeFinalStateBiasingOperation( const G4Track* track, const G4BiasingProcessInterface* callingProcess ) = 0;`
  - The `G4VBiasingOperation` returned will have its `ApplyFinalStateBiasing(...)` called to generated the physics-based biased final state.
  - This method is called at the `PostStepDoIt(...)` level.

3.6.3. Physics Process Occurence Biasing

The occurence biasing is the biasing which affects the probability for a physics process to occur. **For now, we discuss only the case of neutral particles, ie, having no continuous energy loss along a step.** The case of charged particles is expected to be treated in later releases.

The process occurence is driven by the exponential law, which parameter in Geant4 is the process mean free path λ, which is also the inverse of the cross-section σ. The (analog or natural) probability density function (*pdf*) of

interactions is given by $p_{a(\#)}=\sigma^a \cdot \exp(-\sigma^a \#)$, where # is the distance at which the interaction occurs, and where we have relabelled σ as σ^a. In Geant4, volumes are made of a single material, meaning that σ^a does not depend on # : at some position, starting point of a step, the track "sees" the same cross-section at all positions # in the volume.

The occurence biasing consists in substituting $p_a(\#)$ by some arbitrary biased interaction law $p_b(\#)$.

3.6.3.1. Formalism for occurence biasing

More details about the formalism will be provided in the physics reference manual biasing related part (to come).

An arbitrary interaction law $p_b(\#)$ can be recasted in term of an "effective" cross-section $\sigma^b(\#)$, which depends on # in general, as

- $p_b(\#) = \sigma^b(\#) \cdot \exp(-\int \sigma^b(s)s)$, where the integration runs over [0,#],
- where $\sigma^b(\#)$ is given by $\sigma^b(\#) = p_b(\#) / P_{NI,b}(\#)$
- where $P_{NI,b}(\#)$ is the probability for non-interaction along segment [0,#] and is given by $P_{NI,b}(\#) = 1-\#p_b(s)ds = \exp(-\int \sigma^b(s)ds)$ where the integration runs over [0,#].

which is the formalism that corresponds to non-constant cross-sections.

When applying this formalism to the analog *pdf* $p_a(\#)$, we simply have $\sigma^b(\#) = \sigma^a$ and $P_{NI,b}(\#) = \exp(-\sigma^a \#)$, as this must.

In contrast with the analog case, the "effective" cross-section may depend on #, even in a volume made of single material. An example of this is the forced interaction: if it is decided that the interaction of a process, with analog cross-section σ^a, will be forced and will happen somewhere on the segment [0,L], then we have

- Forced interaction over segment [0,L] case:
- $p_b(\#) = \sigma^a \exp(-\sigma^a \#)/(1-\exp(-\sigma^a L))$,
- $P_{NI,b}(\#) = 1 - (1-\exp(-\sigma^a \#))/(1-\exp(-\sigma^a L))$,
- $\sigma^b(\#) = \sigma^a / (1-\exp(-\sigma^a (L-\#))$.

In the occurence biasing, two weights have to be taken into account:

- When the track travels from 0 to # without interaction, it has a different probability to do so in the biased and analog schemes, meaning that a weight for non-interaction has to be applied, this weight being:
  - $w_{NI} = P_{NI,a}(\#) / P_{NI,b}(\#)$.

  If several processes are biased, each with a dedicated law, then, as the total probability for non-interaction is the product of individual probabilities, the total non-interaction weight is simply the product of the individual weights. This non-interaction weight has hence to be applied for all biased processes.
- If the track then makes an interaction in the next segment d# the analog process would have a probability $\sigma^a d\#$ to do so, while this probability is $\sigma^b(\#)d\#$ for the biased process. A weight for interaction has hence to applied and is
  - $w_I = \sigma^a / \sigma^b(\#)$,

  where the analog and biased cross-sections are for *the* process which is taking place.

3.6.3.2. Implementation of Occurence Biasing

Previous section shows that weights for non-interaction probability and effective cross-section are needed to compute the related non-interaction and interaction weights. The class `G4VBiasingInteractionLaw` is the interface for implementing "interaction laws", it defines the pure virtual methods

- `virtual G4double ComputeNonInteractionProbabilityAt(G4double length) const = 0;`
- `virtual G4double ComputeEffectiveCrossSectionAt(G4double length) const = 0;`

that are used in these weights calculations. It defines also the pure virtual method:

- `virtual G4double SampleInteractionLength() = 0;`

In the case of the occurence biasing, the dedicated `virtual const G4VBiasingOperation::ProvideOccurenceBiasingInteractionLaw(const G4BiasingProcessInterface* callingProcess, G4ForceCondition& proposeForce-`

`Condition ) = 0;` is meant to return the biased law. The `G4BiasingProcessInterface` will not change the state of the law. It will collect the sampled interaction length (that the biasing operation must have asked to law to sample) and will used the non-interaction probability method in its `AlongStepDoIt(...)` to compute the weight for non-interaction, these weight being multiplied among biased processes, and it will use the effective cross-section of process "i", if process "i" wins the interaction length race, in its `PostStepDoIt(...)` to compute the weight for interaction.

To compute these weights, the `G4BiasingProcessInterface` holds a private interaction law, to which it sets the analog process cross-section that it collects at the beginning of the step.

As occurence biasing and final state biasing are independent operations, the weight correction for interaction due to the occurence biasing is applied on top of the final state generated by the process (this final state being biased or not).

3.7. Visualisation

This Chapter is intended to be read after Chapter Section 2.12 on Visualisation object oriented design in Part II. Many of the concepts used here are defined there, and it strongly recommended that a writer of a new visualisation driver or trajectory drawer reads Chapter Section 2.12 first. The class structure described there is summarised in Figure 3.11.

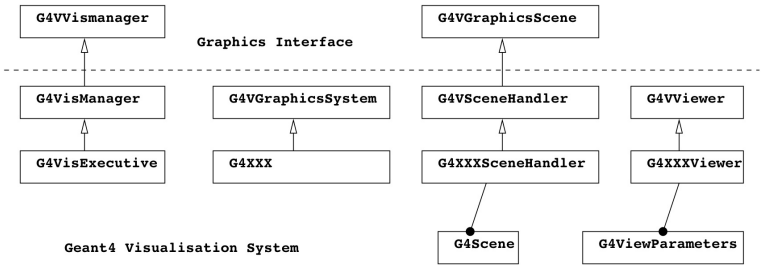

Figure 3.11. Geant Visualisation System Class Diagram

3.7.1. Creating a new graphics driver

To create a new graphics driver for Geant4, it is necessary to implement a new set of three classes derived from the three base classes, `G4VGraphicsSystem`, `G4VSceneHandler` and `G4VViewer`.

3.7.1.1. A useful place to start

A skeleton set of classes is included in the code distribution in the visualisation category under subdirectory `visualisation/XXX` (but they are not default-registered graphics systems [2]

There are several sets of classes, described in more detail below. A recommended approach is to copy the files that best match your graphics system to a new subdirectory with a name that suits your graphics system .

Then

1. Change the name of the files (change the code -- `XXX` or `XXXFile`, etc., as chosen -- to something that suits your graphics system).
2. Change `XXX` similarly in all files.
3. Change `XXX` similarly in `name := G4XXX` in `GNUmakefile`.
4. Add your new subdirectory to `SUBDIRS` and `SUBLIBS` in `visualisation/GNUmakefile`.
5. Look at the code and use it to build your visualisation driver. You might also find it useful to look at `ASCIITree` (and `VTree`) as an example of a minimal graphics driver . Look at `FukuiRenderer` as an example of a driver which implements `AddSolid` methods for some solids. Look at `OpenGL` as an example of a driver which implements a graphical database (display lists) and the machinery to decide when to rebuild. (OpenGL is complicated by the proliferation of combinations of the use or not of display lists for

[2] To do this,simply instantiate and register, for example: `visManager->RegisterGraphicsSystem(new G4XXX)` before `visManager->Initialise()`.

three window systems, X-windows, X with motif (interactive), Microsoft Windows (Win32), a total of six combinations, and much use is made of inheritance to avoid code duplication.)

6. If it requires external libraries, introduce two new environment variables `G4VIS_BUILD_XXX_DRIVER` and `G4VIS_USE_XXX` (where `XXX` is your choice as above) and make the modifications to:
   - `source/visualization/management/include/G4VisExecutive.icc`
   - `config/G4VIS_BUILD.gmk`
   - `config/G4VIS_USE.gmk`

3.7.1.1.1. Graphics driver templates in the XXX sub-category

You may use the following templates to help you get started writing a graphics driver . (The word ``template" is used in the ordinary sense of the word; they are not C++ templates.)

- `G4XXX, G4XXXSceneHandler, G4XXXViewer` Templates for the simplest possible graphics driver . These would be suitable for an ``immediate" driver, i.e., one which renders each object immediately to a screen. Of course, if the view needs re-drawing, as, for example, after a change of viewpoint, the viewer requests a re-issue of drawn objects.
- `G4XXXFile, G4XXXFileSceneHandler, G4XXXFileViewer` Templates for a file-writing graphics driver. The particular features are: delayed opening of the file on receipt of the first item; rewinding file on ClearView (to simulate the clearing of views and prevent the duplication of material in the file); closing of the file on ShowView, which may also trigger the launch of a browser. There are various degrees of sophistication in, for example, the allocation of filenames -- see `FukuiRenderer` or `HepRepFile`.

  These templates also show the use of a specific `AddSolid` function whereby the specific parameters, for example, the dimensions of a `G4Box`, can be accessed.
- `G4XXXStored, G4XXXStoredSceneHandler, G4XXXStoredViewer` Templates for a graphics driver with a store/database. The advantage of a store is that the view can be refreshed, for example, from a different viewpoint, without a need to recompute. It is up to the viewer to decide when a re-computation is necessary. They also show how to distinguish between permanent and transient objects -- see also Section Section 3.7.1.6.
- `G4XXXSG, G4XXXSGSceneHandler, G4XXXSGViewer` Templates for a sophisticated graphics driver with a scene graph. The scene graph, following Open Inventor parlance, is a tree of objects that dictates the order in which the objects are rendered. It obviously lends itself to the rendering of the Geant4 geometry hierarchy. For example, the Open Inventor driver draws only the top level volumes unless made invisible by picking. Thus the user can unwrap a branch of the geometry level by level. This has performance benefits and gives the user significant and useful control over the view. These classes show how to make a scene graph of **drawn** volumes, i.e., the set of volumes that have not been culled. (Normally, volumes marked invisible are culled, i.e., not drawn. Also, the user may wish to limit the number of drawn volumes for performance reasons.) The drivers also have to process non-geometry items and distinguish between transient and permanent objects as above.

3.7.1.2. Important Command Actions

To help understand how the Geant4 Visualization System works, here are a few important function invocation sequences that follow user commands. For an explanation of the commands themselves, see command guidance or the Control section of the Application Developers Guide. For a fuller explanation of the functions, see appropriate base class head files or Software Reference Manual.

- `/vis/viewer/clear`

```
viewer->ClearView();   // Clears buffer or rewinds file.
viewer->FinishView();  // Swaps buffer (double buffer systems).
```

- `/vis/viewer/flush`

```
/vis/viewer/refresh
/vis/viewer/update
```

- `/vis/viewer/rebuild`

```
   viewer->SetNeedKernelVisit(true);
```

- `/vis/viewer/refresh`

  If the view is ``auto-refresh'', this command is also invoked after `/vis/viewer/create`, `/vis/viewer/rebuild` or a change of view parameters (`/vis/viewer/set/`..., etc.).

```
   viewer->SetView();    // Sets camera position, etc.
   viewer->ClearView();  // Clears buffer or rewinds file.
   viewer->DrawView();   // Draws to screen or writes to
                         // file/socket.
```

- `/vis/viewer/update`

```
   viewer->ShowView();   // Activates interactive windows or
                         // closes file and/or triggers
                         // post-processing.
```

- `/vis/scene/notifyHandlers`

  For each viewer of the current scene, the equivalent of

```
   /vis/viewer/refresh
```

  If ``flush'' is specified on the command line, the equivalent of

```
   /vis/viewer/update
```

  `/vis/scene/notifyHandlers` is also invoked after a change of scene (`/vis/scene/add/`..., etc.).

3.7.1.3. What happens in `DrawView`?

This depends on the viewer. Those with their own graphical database, for example, OpenGL's display lists or Open Inventor's scene graph, do not need to re-traverse the scene unless there has been a significant change of view parameters. For example, a mere change of viewpoint requires only a change of model-view matrix whilst a change of rendering mode from wireframe to surface might require a rebuild of the graphical database. A rebuild of the run-duration (persistent) objects in the scene is called a ``kernel visit''; the viewer prints ``Traversing scene data...''.

Note that end-of-event (transient) objects are only rebuilt at the end of an event or run, under control of the visualisation manager. Smart scene handlers keep them in separate display lists so that they can be rebuilt separately from the run-duration objects - see Section 3.7.1.6.

- **Integrated viewers with no graphical database** For example, `G4OpenGLImmediateXViewer::DrawView()`.

```
   NeedKernelVisit();  // Always need to visit G4 kernel.
   ProcessView();
   FinishView();
```

- **Integrated viewers with graphical database** For example, `G4OpenGLStoredXViewer::DrawView()`.

```
   KernelVisitDecision();  // Private function containing...
     if significant change of view parameters...
       NeedKernelVisit();
   ProcessView();
   FinishView();
```

- **File-writing viewers** For example, `G4DAWNFILEViewer::DrawView()`.

```
   NeedKernelVisit();
```

```
    ProcessView();
```

Note that viewers needing to invoke `FinishView` must do it in `DrawView`.

3.7.1.4. What happens in `ProcessView`?

`ProcessView` is inherited from `G4VViewer`:

```
void G4VViewer::ProcessView() {
  // If ClearStore has been requested, e.g., if the scene has changed,
  // of if the concrete viewer has decided that it necessary to visit
  // the kernel, perhaps because the view parameters have changed
  // drastically (this should be done in the concrete viewer's
  // DrawView)...
  if (fNeedKernelVisit) {
    fSceneHandler.ProcessScene(*this);
    fNeedKernelVisit = false;
  }
}
```

3.7.1.5. What happens in `ProcessScene`?

ProcessScene is inherited from `G4VSceneHandler}`. It causes a traversal of the run-duration models in the scene. For drivers with graphical databases, it causes a rebuild (`ClearStore`). Then for the run-duration models:

```
    fReadyForTransients = false;
    BeginModeling();
    for each run-duration model...
      pModel -> DescribeYourselfTo(*this);
    EndModeling();
    fReadyForTransients = true;
```

(A second pass is made on request -- see `G4VSceneHandler::ProcessScene`.) The use of `fReadyFor-Transients` is described in Section 3.7.1.6.

What happens then depends on the type of model:

- `G4AxesModel` `G4AxesModel::DescribeYourselfTo` simply calls sceneHandler.AddPrimitive methods directly.

  ```
    sceneHandler.BeginPrimitives();
    sceneHandler.AddPrimitive(x_axis);  // etc.
    sceneHandler.EndPrimitives();
  ```

  Most other models are like this, except for the following...
- `G4PhysicalVolumeModel` The geometry is descended recursively, culling policy is enacted, and for each accepted (and possibly, clipped) solid:

  ```
    sceneHandler.PreAddSolid(theAT, *pVisAttribs);
    pSol->DescribeYourselfTo(sceneHandler);
    // For example, if pSol points to a G4Box...
    |-->G4Box::DescribeYourselfTo(G4VGraphicsScene& scene){
         scene.AddSolid(*this);
        }
    sceneHandler.PostAddSolid();
  ```

  The scene handler may implement the virtual function { AddSolid(const G4Box&)}, or inherit:

  ```
    void G4VSceneHandler::AddSolid(const G4Box& box) {
      RequestPrimitives(box);
    }
  ```

  `RequestPrimitives` converts the solid into primitives (`G4Polyhedron`) and invokes `AddPrimitive`:

```
BeginPrimitives(*fpObjectTransformation);
pPolyhedron = solid.GetPolyhedron();
AddPrimitive(*pPolyhedron);
EndPrimitives();
```

The resulting default sequence for a `G4PhysicalVolumeModel` is shown in Figure 3.12.

```
DrawView();
|-->ProcessView();
    |-->ProcessScene();
        |-->BeginModeling();
        |-->pModel -> DescribeYourselfTo(*this);
        |   |-->sceneHandler.PreAddSolid(theAT, *pVisAttribs);
        |   |-->pSol->DescribeYourselfTo(sceneHandler);
        |   |   |-->sceneHandler.AddSolid(*this);
        |   |       |-->RequestPrimitives(solid);
        |   |           |-->BeginPrimitives (*fpObjectTransformation);
        |   |           |-->pPolyhedron = solid.GetPolyhedron();
        |   |           |-->AddPrimitive(*pPolyhedron);
        |   |           |-->EndPrimitives();
        |   |-->sceneHandler.PostAddSolid();
        |-->EndModeling();
```

Figure 3.12. The default sequence for a `G4PhysicalVolumeModel}`

Note the sequence of calls at the core:

```
sceneHandler.PreAddSolid(theAT, *pVisAttribs);
pSol->DescribeYourselfTo(sceneHandler);
|-->sceneHandler.AddSolid(*this);
   |-->RequestPrimitives(solid);
      |-->BeginPrimitives (*fpObjectTransformation);
      |-->pPolyhedron = solid.GetPolyhedron();
      |-->AddPrimitive(*pPolyhedron);
      |-->EndPrimitives();
sceneHandler.PostAddSolid();
```

is reduced to

```
sceneHandler.PreAddSolid(theAT, *pVisAttribs);
pSol->DescribeYourselfTo(sceneHandler);
|-->sceneHandler.AddSolid(*this);
sceneHandler.PostAddSolid();
```

if the scene handler implements its own `AddSolid`. Moreover, the sequence

```
BeginPrimitives (*fpObjectTransformation);
AddPrimitive(*pPolyhedron);
EndPrimitives();
```

can be invoked without a prior `PreAddSolid`, etc. The flag `fProcessingSolid` will be false for the last case. The possibility of any or all of these three scenarios occurring, for both permanent and transient objects, affects the implementation of a scene handler if there is any attempt to build a graphical database. This is reflected in the templates `XXXStored` and `XXXSG` described in Section 3.7.1.1.1. Transients are discussed in Section 3.7.1.6.

- `G4TrajectoriesModel` At end of event, the trajectory container is unpacked and, for each trajectory, `sceneHandler.AddCompound` called. The scene handler may implement this virtual function or inherit:

```
}
```

Similarly, the user may implement `DrawTrajectory` or inherit:

```
void G4VTrajectory::DrawTrajectory(G4int i_mode) const {
  G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
  if (0 != pVVisManager) {
    pVVisManager->DispatchToModel(*this, i_mode);
  }
}
```

Thence, the `Draw` method of the current trajectory model is invoked (see Section 3.7.2 for details on trajectory models), which in turn, invokes `Draw` methods of the visualisation manager. The resulting default sequence for a `G4TrajectoriesModel` is shown in Figure 3.13.

```
DrawView();
|-->ProcessView();
    |-->ProcessScene();
        |-->BeginModeling();
        |-->pModel -> DescribeYourselfTo(*this);
        |   |-->AddCompound(trajectory);
        |       |-->trajectory.DrawTrajectory(...);
        |           |-->DispatchToModel(...);
        |               |-->model->Draw(...);
        |                   |-->G4VisManager::Draw(...);
        |                       |-->BeginPrimitives(objectTransform);
        |                       |-->AddPrimitive(...);
        |                       |-->EndPrimitives();
        |-->EndModeling();
```

Figure 3.13. The default sequence for a `G4PhysicalVolumeModel}`

3.7.1.6. Dealing with transient objects

Any visualisable object not defined in the run-duration part of a scene is treated as ``transient''. This includes trajectories, hits or anything drawn by the user through the `G4VVisManager` user-level interface (unless as part of a run-duration model implementation). A flag, `fReadyForTransients}`, is maintained by the scene handler. In fact, its normal state is `true`, and only temporarily, during handling of the run-duration part of the scene, is it set to `false` -- see description of ProcessScene, Section 3.7.1.5.

If the driver supports a graphical database, it is smart to distinguish transient and permanent objects. In this case, every `Add` method of the scene handler must be transient-aware. In some cases, it is enough to open a graphical data base component in `BeginPrimitives`, fill it in `AddPrimitive` and close it appropriately in `EndPrimitives`. In others, initialisation is done in `BeginModeling` and consolidation in `EndModeling` -- see `G4OpenGLStoredSceneHandler`. If any `AddSolid` method is implemented, then the graphical data base component should be opened in `PreAddSolid`, protecting against double opening, for example,

```
void G4XXXStoredSceneHandler::BeginPrimitives
(const G4Transform3D& objectTransformation) {
  G4VSceneHandler::BeginPrimitives(objectTransformation);
  // If thread of control has already passed through PreAddSolid,
  // avoid opening a graphical data base component again.
  if (!fProcessingSolid) {
```

for other solids.

The reason for this distinction is that at end of run the user typically wants to display trajectories on a view of the detector, then, at the end of the next event [3] , erase the old and see new trajectories. The visualisation manager messages the scene handler with `ClearTransientStore` just before drawing the trajectories to achieve this.

[3] There is an option to accumulate trajectories across events and runs -- see commands `/vis/scene/endOfEventAction` and `/vis/scene/endOfRunAction`.

If the driver does not have a graphical database or does not distinguish between transient and persistent objects, it must emulate `ClearTransientStore`. Typically, it must erase everything, including the detector, and re-draw the detector and other run-duration objects, ready for the transients to be added. File-writing drivers must rewind the output file. Typically:

```
void G4HepRepFileSceneHandler::ClearTransientStore() {
  G4VSceneHandler::ClearTransientStore();
  // This is typically called after an update and before drawing hits
  // of the next event.  To simulate the clearing of "transients"
  // (hits, etc.) the detector is redrawn...
  if (fpViewer) {
    fpViewer -> SetView();
    fpViewer -> ClearView();
    fpViewer -> DrawView();
  }
}
```

`ClearView` rewinds the output file and `DrawView` re-draws the detector, etc. (For smart drivers, `DrawView` is smart enough to know not to redraw the detector, etc., unless the view parameters have changed significantly -- see Section 3.7.1.3)

3.7.1.7. More about scene models

Scene models conform to the `G4VModel` abstract interface. Available models are listed and described there in varying detail. Section 3.7.1.5 describes their use in some common command actions.

In the design of a new model, care should be taken to handle the possibility that the `G4ModelingParameters` pointer is zero. Currently the only use of the modeling parameters is to communicate the culling policy. Most models, therefore, have no need for modeling parameters.

3.7.2. Enhanced Trajectory Drawing

3.7.2.1. Creating a new trajectory model

New trajectory models must inherit from G4VTrajectoryModel and implement these pure virtual functions:

```
   virtual void Draw(const G4VTrajectory&, G4int i_mode = 0,
                     const G4bool& visible = true) const = 0;
   virtual void Print(std::ostream& ostr) const = 0;
```

To use the new model directly in compiled code, simply register it with the G4VisManager, eg:

```
 G4VisManager* visManager = new G4VisExecutive;
 visManager->Initialise();

 // Create custom model
 MyCustomTrajectoryModel* myModel =
            new MyCustomTrajectoryModel("custom");

 // Configure it if necessary then register with G4VisManager
 ...
 visManager->RegisterModel(myModel);
```

3.7.2.2. Adding interactive functionality

Additional classes need to be written if the new model is to be created and configured interactively:

- **Messenger classes**

  Messengers to configure the model should inherit from G4VModelCommand. The concrete trajectory model type should be used for the template parameter, eg:

```
class G4MyCustomModelCommand
      : public G4VModelCommand<G4TrajectoryDrawByParticleID> {
...
};
```

A number of general use templated commands are available in G4ModelCommandsT.hh.

- **Factory class**

  A factory class responsible for the model and associated messenger creation must also be written. The factory should inherit from G4VModelFactory. The abstract model type should be used for the template parameter, eg:

```
class G4TrajectoryDrawByChargeFactory
   : public G4VModelFactory<G4VTrajectoryModel> {
 ...
};
```

  The model and associated messengers should be constructed in the Create method. Optionally, a context object can also be created, with its own associated messengers. For example:

```
ModelAndMessengers
G4TrajectoryDrawByParticleIDFactory::
    Create(const G4String& placement, const G4String& name)
{
  // Create default context and model
  G4VisTrajContext* context = new G4VisTrajContext("default");
  G4TrajectoryDrawByParticleID* model =
              new G4TrajectoryDrawByParticleID(name, context);

  // Create messengers for default context configuration
  AddContextMsgrs(context, messengers, placement+"/"+name);

  // Create messengers for drawer
  messengers.push_back(new
    G4ModelCmdSetStringColour<G4TrajectoryDrawByParticleID>
                                          (model, placement));
  messengers.push_back(new
    G4ModelCmdSetDefaultColour<G4TrajectoryDrawByParticleID>
                                          (model, placement));
  messengers.push_back(new
    G4ModelCmdVerbose<G4TrajectoryDrawByParticleID>
                                          (model, placement));

  return ModelAndMessengers(model, messengers);
}
```

The new factory must then be registered with the visualisation manager. This should be done by overriding the G4VisManager::RegisterModelFactory method in a subclass. See, for example, the G4VisManager implementation:

```
G4VisExecutive::RegisterModelFactories()
{
   ...
   RegisterModelFactory(new G4TrajectoryDrawByParticleIDFactory());
}
```

3.7.3. Trajectory Filtering

3.7.3.1. Creating a new trajectory filter model

New trajectory filters must inherit at least from G4VFilter. The models supplied with the Geant4 distribution inherit from G4SmartFilter, which implements some specialisations on top of G4VFilter. The models implement these pure virtual functions:

```
// Evaluate method implemented in subclass
virtual G4bool Evaluate(const T&) = 0;

// Print subclass configuration
virtual void Print(std::ostream& ostr) const = 0;
```

To use the new filter model directly in compiled code, simply register it with the G4VisManager, eg:

```
G4VisManager* visManager = new G4VisExecutive;
visManager->Initialise();

// Create custom model
MyCustomTrajectoryFilterModel* myModel =
      new MyCustomTrajectoryFilterModel("custom");

// Configure it if necessary then register with G4VisManager
...
visManager->RegisterModel(myModel);
```

3.7.3.2. Adding interactive functionality

Additional classes need to be written if the new model is to be created and configured interactively. The mechanism is exactly the same as that used to create enchanced trajectory drawing models and associated messengers. See the filter factories in G4TrajectoryFilterFactories for example implementations.

3.7.4. Other Resources

The following sections contain various information for extending other class functionalities of Geant4 visualisation:

- User's Guide for Application Developers, Chapter 8 - Visualization
- User's Guide for Toolkit Developers, Object-oriented Analysis and Design of Geant4 Classes, Section 2.12.

Bibliography

[Gamma1995] E. Gamma. *Design Patterns* . Addison-Wesley Professional Computing Series . 1995 .

[QGSM] Kaidalov A. B., Ter-Martirosyan. *Phys. Lett.*. B117 (1982) 247.

[ENDFForm] *Data Formats and Procedures for the Evaluated Nuclear Data File* . National Nuclear Data Center, Brookhaven National Laboratory, Upton, NY, USA. .

[QMD] “For example: VUU and (R)QMD model of high-energy heavy ion collisions ”. H. Stocker et al.. *Nucl. Phys.*. A538, 53c-64c (1992).

[CHIPS] P.V. Degtyarenko, M.V. Kossov, H.P. Wellisch. *Eur. Phys J.*. A 8, 217-222 (2000).

[VNI] Klaus Geiger. *Comput. Phys. Commun.*. 104, 70-160 (1997). *Brookhaven*. BNL-63762.

[PYTHIA7] “Pythia version 7-0.0 -- a proof-of-concept version”. M. Bertini, L. Lonnblad, T. Sjorstrand. . LU-TP 00-23, hep-ph/0006152. May 2000.