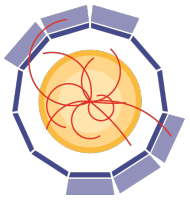

AIDA 2020

Advanced European Infrastructures for Detectors at Accelerators

DDG4

A Simulation Toolkit for High Energy Physics Experiments using Geant4 and the DD4hep Geometry Description

M. Frank

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

CERN, 1211 Geneva 23, Switzerland

Abstract

Simulating the detector response is an essential tool in high energy physics to analyze the sensitivity of an experiment to the underlying physics. Such simulation tools require a detailed though convenient detector description as it is provided by the `DD4hep` toolkit. We will present the generic simulation toolkit `DDG4` using the `DD4hep` detector description toolkit. The toolkit implements a modular and flexible approach to simulation activities using Geant4. User defined simulation applications using `DDG4` can easily be configured, extended using specialized action routines. The design is strongly driven by easy of use; developers of detector descriptions and applications using them should provide minimal information and minimal specific code to achieve the desired result.

Document History		
Document version	**Date**	**Author**
1.0	**19/11/2013**	**Markus Frank CERN/LHCb**

Contents

1 Introduction

This manual should introduce to the DDG4 framework. One goal of DDG4 is to easily configure the simulation applications capable of simulating the physics response of detector configurations as they are used for example in high energy physics experiments. In such simulation programs the user normally has to define the experimental setup in terms of its geometry and in terms of its active elements which sample the detector response.
The goal of DDG4 is to generalize the configuration of a simulation application to a degree, which does not force users to write code to test a detector design. At the same time it should of course be feasible to supply specialized user written modules which are supposed to seamlessly operate together with standard modules supplied by the toolkit. Detector-simulation depends strongly on the use of an underlying simulation toolkit, the most prominent candidate nowadays being Geant4 [8]. DD4hep supports simulation activities with Geant4 providing an automatic translation mechanism between geometry representations. The simulation response in the active elements of the detector is strongly influenced by the technical choices and precise simulations depends on the very specific detection techniques.
Similar to the aim of DD4hep [1], where with time a standard palette of detector components developed by users should become part of the toolkit, DDG4 also hopes to provide a standard palette of components used to support simulation activities for detector layouts where detector designers may base the simulation of a planned experiment on these predefined components for initial design and optimization studies. The longterm vision is to construct simulation applications writing only new components not yet present i.e. the main work will be to select the appropriate components from the palette and connect them to a functional program.
This is not a manual to Geant4 nor the basic infrastructure of DD4hep . It is assumed that this knowledge is present and the typical glossary is known.

2 The Geant4 User Interface

The Geant4 simulation toolkit [8] implements a very complex machinery to simulate the energy deposition of particles traversing materials. To ease its usage for the clients and to shield clients from the complex internals when actually implementing a simulation applications for a given detector design, it provides several user hooks as shown in Figure 1. Each of these hooks serves a well specialized purpose, but unfortunately also leads to very specialized applications. One aim of DDG4 is to formalize these user actions so that the invocation at the appropriate time may be purely data driven.
In detail the following object-hooks allow the client to define user provided actions:

- The **User Physics List** allows the client to customize and define the underlying physics process(es) which define the particle interactions inside the detector defined with the geometry description. These interactions define the detector response in terms of energy depositions.
- The **Run Action** is called once at the start and end of a run. i.e. a series of generated events. These two callbacks allow clients to define run-dependent actions such as statistics summaries etc.
- The **Primary Generator Action** is called for every event. During the callback all particles are created which form the microscopic kinematic action of the particle collision. This input may either origin directly from an event generator program or come from file.
- The **Event Action** is called once at the start and the end of each event. It is typically used for a simple analysis of the processed event. If the simulated data should be written to some persistent medium, the call at the end of the event processing is the appropriate place.
- The **Tracking Action**
- The **Stepping Action**
- The **Stacking Action**

Geant4 provides all callbacks with the necessary information in the form of appropriate arguments.

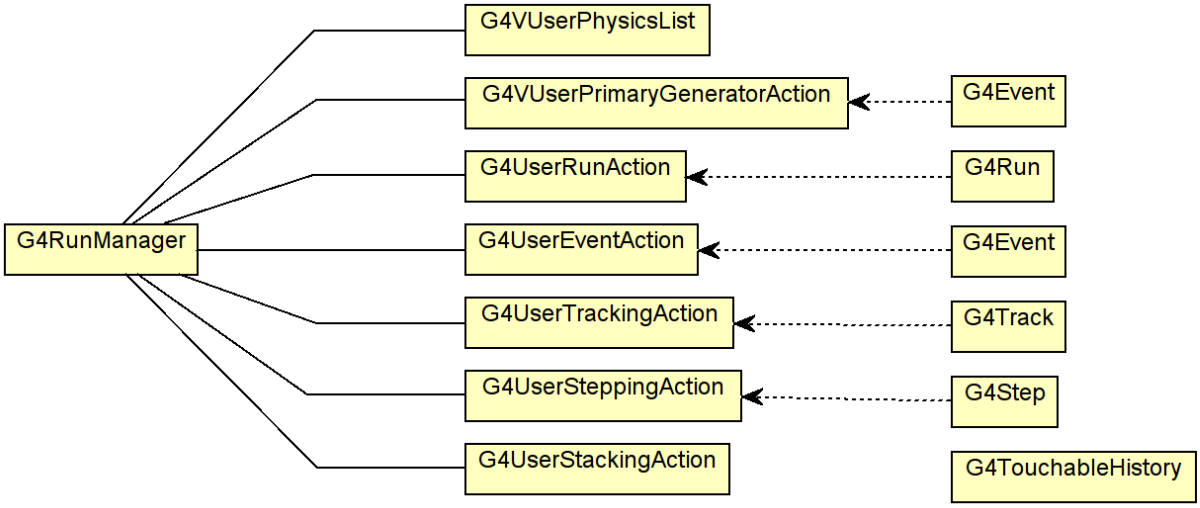

Figure 1: The various user hooks provided by Geant4. Not shown here is the callback system interfacing to the active elements of the detector design.

Besides the callback system, Geant4 provides callbacks whenever a particle traverses a sensitive volume. These callbacks are called - similar to event actions - once at the start and the end of the event, but in addition, if either the energy deposit of a particle in the sensitive volume exceeds some threshold. The callbacks are formalized within the base class `G4VSensitiveDetector`.

3 DDG4 Implementation

A basic design criteria of the a `DDG4` simulation application was to process any user defined hook provided by Geant4 as a series of algorithmic procedures, which could be implemented either using inheritance or by a callback mechanism registering functions fulfilling a given signature. Such sequences are provided for all actions mentioned in the list in Section 2 as well as for the callbacks to sensitive detectors.
The callback mechanism was introduced to allow for weak coupling between the various actions. For example could an action performing monitoring using histograms at the event level initialize or reset its histograms at the start/end of each run. To do so, clearly a callback at the start/end of a run would be necessary.
In the following sections a flexible and extensible interface to hooks of Geant4 is discussed starting with the description of the basic components `Geant4Kernel` and `Geant4Action` followed by the implementation of the relevant specializations. The specializations exposed are sequences of such actions, which also call registered objects. In later section the configuration and the combination of these components forming a functional simulation application is presented.

3.1 The Application Core Object: Geant4Kernel

The kernel object is the central context of a `DDG4` simulation application and gives all clients access to the user hooks (see Figure 2). All Geant4 callback structures are exposed so that clients can easily objects implementing the required interface or register callbacks with the correct signature. Each of these action sequences is connected to an instance of a Geant4 provided callback structure as it is shown in Figure 1.

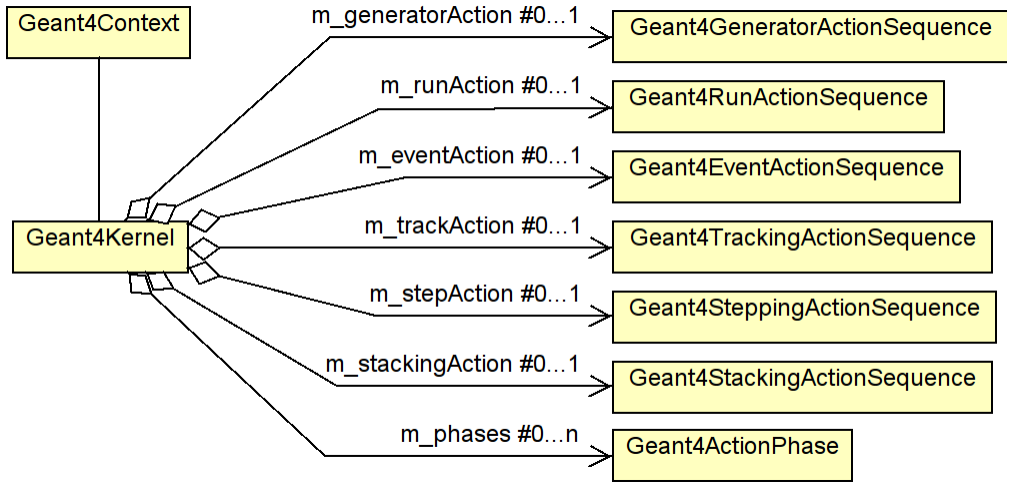

Figure 2: The main application object gives access to all sequencing actions in a DDG4 4 application. Sequence actions are only container of user actions calling one user action after the other. Optionally single callbacks may be registered to a user action.

3.2 Action Sequences

As shown in

3.3 The Base Class of DDG4 Actions: Geant4Action

The class `Geant4Action` is a common component interface providing the basic interface to the framework to

- configure the component using a property mechanism
- provide an appropriate interface to Geant4 interactivity. The interactivity included a generic way to change and access properties from the Geant4 UI prompt as well as executing registered commands.
- As shown in Figure 3, the base class also provides to its sub-class a reference to the `Geant4Kernel` objects through the `Geant4Context`.

The `Geant4Action` is a named entity and can be uniquely identified within a sequence attached to one Geant4 user callback.

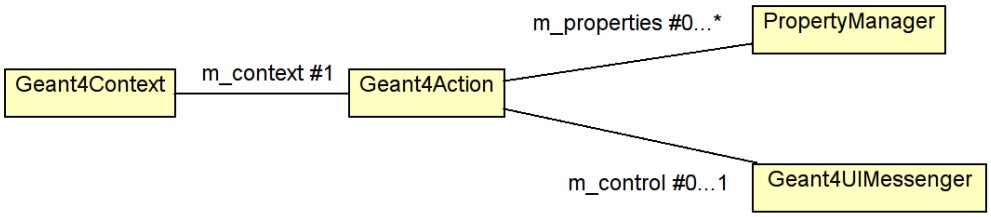

Figure 3: The design of the common base class `Geant4Action`.

DDG4 knows two types of actions: global actions and anonymous actions. Global actions are accessible externally from the `Geant4Kernel` instance. Global actions are also re-usable and hence may be contribute to several action sequences (see the following chapters for details). Global actions are uniquely

identified by their name. Anonymous actions are known only within one sequence and normally are not shared between sequences.

3.3.1 The Properties of Geant4Action Instances

Nearly any subclass of a `Geant4Action` needs a flexible configuration in order to be reused, modified etc. The implementation of the mechanism uses a very flexible value conversion mechanism using `boost::spirit`, which support also conversions between unrelated types provided a dictionary is present. Properties are supposed to be member variables of a given action object. To publish a property it needs to be declared in the constructor as shown here:

```
declareProperty("OutputLevel", m_outputLevel = INFO);
declareProperty("Control",     m_needsControl = false);
```

The internal setup of the `Geant4Action` objects then ensure that all declared properties will be set after the object construction to the values set in the setup file.
Note: Because the values can only be set **after** the object was constructed, the actual values may not be used in the constructor of any base or sub-class.

3.4 Geant4 Action Sequences

All Geant4 user hooks are realized as action sequences. As shown in Figure 2 these sequences are accessible to the user, who may attach specialized actions to the different action sequences. This allows a flexible handing of specialized user actions e.g. to dynamically add monitoring actions filling histograms or to implement alternative hit creation mechanism in a sensitive detector for detailed detector studies. Figure 4 shows the schematic call structure of an example `Geant4TrackingActionSequence`:

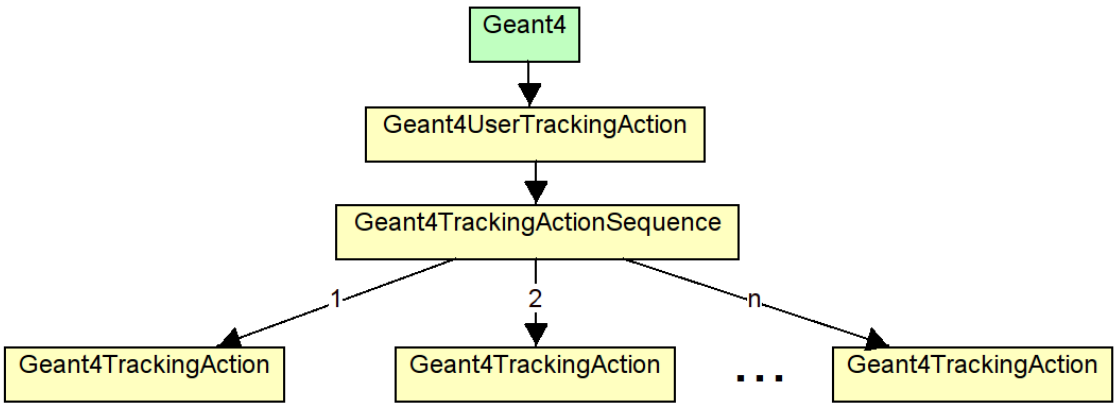

Figure 4: The design of the tracking action sequence. Specialized tracking action objects inherit from the `Geant4TrackingAction` object and must be attached to the sequence.

Geant4 calls the function from the virtual interface (`G4UserTrackingAction`), which is realised by the `Geant4UserTrackingAction` with the single purpose to propagate the call to the action sequence, which then calls all registered clients of type `Geant4TrackingAction`.
The main action sequences have a fixed name. These are

- The **RunAction** attached to the `G4UserRunAction`, implemented by the `Geant4RunActionSequence` class and is called at the start and the end of every run (beamOn). Members of the `Geant4RunActionSequence` are of type `Geant4RunAction` and receive the callbacks by overloading the two routines:

```
/// begin-of-run callback
virtual void begin(const G4Run* run);
/// End-of-run callback
virtual void end(const G4Run* run);
```

or register a callback with the signature `void (T::*)(const G4Run*)` either to receive begin-of-run or end-or-calls using the methods:

```
/// Register begin-of-run callback. Types Q and T must be polymorph!
template <typename Q, typename T> void callAtBegin(Q* p, void (T::*f)(const G4Run*));
/// Register end-of-run callback. Types Q and T must be polymorph!
template <typename Q, typename T> void callAtEnd(Q* p, void (T::*f)(const G4Run*));
```

of the `Geant4RunActionSequence` from the `Geant4Context` object.

- The **EventAction** attached to the `G4UserEventAction`, implemented by the `EventActionSequence` class and is called at the start and the end of every event. Members of the `Geant4EventActionSequence` are of type `Geant4EventAction` and receive the callbacks by overloading the two routines:

```
/// Begin-of-event callback
virtual void begin(const G4Event* event);
/// End-of-event callback
virtual void end(const G4Event* event);
```

or register a callback with the signature `void (T::*)(const G4Event*)` either to receive begin-of-run or end-or-calls using the methods:

```
/// Register begin-of-event callback
template <typename Q, typename T> void callAtBegin(Q* p, void (T::*f)(const G4Event*));
/// Register end-of-event callback
template <typename Q, typename T> void callAtEnd(Q* p, void (T::*f)(const G4Event*));
```

of the `Geant4EventActionSequence` from the `Geant4Context` object.

- The **GeneratorAction** attached to the `G4VUserPrimaryGeneratorAction`, implemented by the `Geant4GeneratorActionSequence` class and is called at the start of every event and provided all initial tracks from the Monte-Carlo generator. Members of the `Geant4GeneratorActionSequence` are of type `Geant4EventAction` and receive the callbacks by overloading the member function:

```
/// Callback to generate primary particles
virtual void operator()(G4Event* event);
```

or register a callback with the signature `void (T::*)(G4Event*)` to receive calls using the method:

```
/// Register primary particle generation callback.
template <typename Q, typename T> void call(Q* p, void (T::*f)(G4Event*));
```

of the `Geant4GeneratorActionSequence` from the `Geant4Context` object.

- The **TrackingAction** attached to the `G4UserTrackingAction`, implemented by the `Geant4- Tracking-ActionSequence` class and is called at the start and the end of tracking one single particle trace through the material of the detector. Members of the `Geant4TrackingActionSequence` are of type `Geant4TrackingAction` and receive the callbacks by overloading the member function:

```
/// Pre-tracking action callback
virtual void begin(const G4Track* trk);
/// Post-tracking action callback
virtual void end(const G4Track* trk);
```

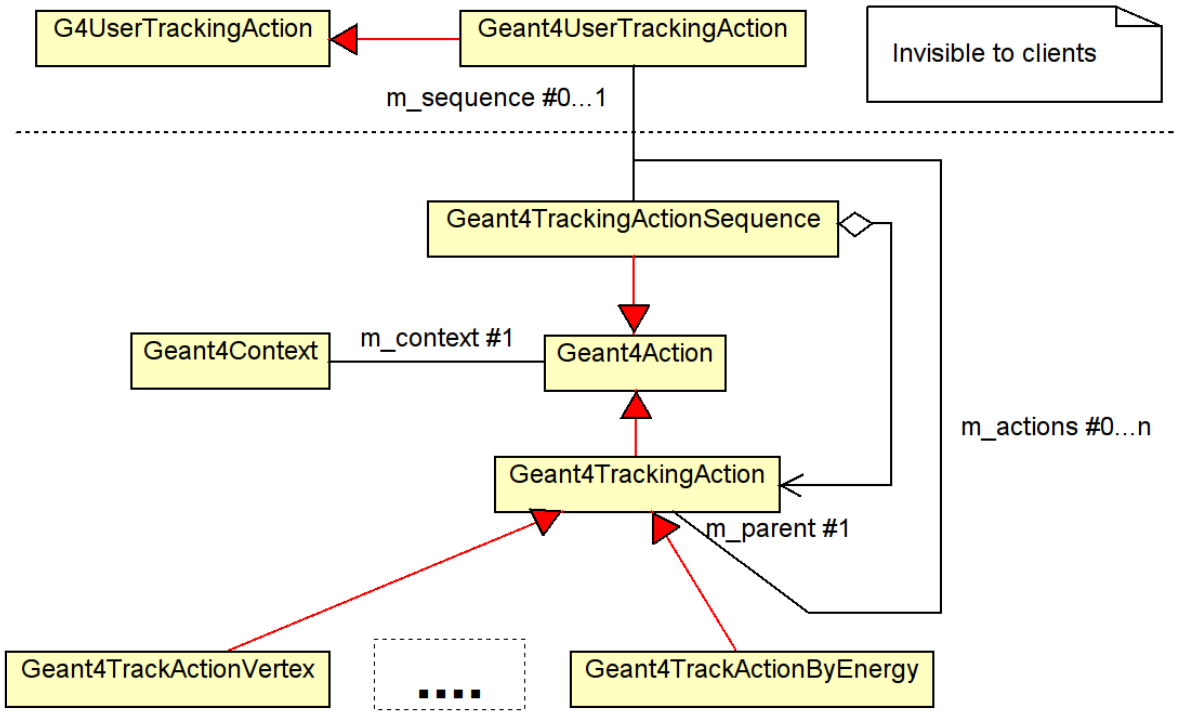

Figure 5: The design of the tracking action sequence. Specialized tracking action objects inherit from the `Geant4TrackingAction` object and must be attached to the sequence.

or register a callback with the signature `void (T::*)(const G4Step*, G4SteppingManager*)` to receive calls using the method:

```
/// Register Pre-track action callback
template <typename Q, typename T> void callAtBegin(Q* p, void (T::*f)(const G4Track*));
/// Register Post-track action callback
template <typename Q, typename T> void callAtEnd(Q* p, void (T::*f)(const G4Track*));
```

Figure 5 show as an example the design (class-diagram) of the `Geant4TrackingAction`.

- The **SteppingAction** attached to the `G4UserSteppingAction`, implemented by the `Geant4- SteppingActionSequence` class and is called for each step when tracking a particle. Members of the `Geant4SteppingActionSequence` are of type `Geant4SteppingAction` and receive the callbacks by overloading the member function:

  ```
  /// User stepping callback
  virtual void operator()(const G4Step* step, G4SteppingManager* mgr);
  ```

  or register a callback with the signature `void (T::*)(const G4Step*, G4SteppingManager*)` to receive calls using the method:

  ```
  /// Register stepping action callback.
  template <typename Q, typename T> void call(Q* p, void (T::*f)(const G4Step*,
                                                                 G4SteppingManager*));
  ```

- The **StackingAction** attached to the `G4UserStackingAction`, implemented by the `Geant4-StackingActionSequence` class. Members of the `Geant4StackingActionSequence` are of type `Geant4StackingAction` and receive the callbacks by overloading the member functions:

```
/// New-stage callback
virtual void newStage();
/// Preparation callback
virtual void prepare();
```

or register a callback with the signature `void (T::*)()` to receive calls using the method:

```
/// Register begin-of-event callback. Types Q and T must be polymorph!
template <typename T> void callAtNewStage(T* p, void (T::*f)());
/// Register end-of-event callback. Types Q and T must be polymorph!
template <typename T> void callAtPrepare(T* p, void (T::*f)());
```

All sequence types support the method `void adopt(T* member_reference)` to add the members. Once adopted, the sequence takes ownership and manages the member. The design of all sequences is very similar.

3.5 Sensitive Detectors

Sensitive detectors are associated by the detector designers to all active materials, which would produce a signal which can be read out. In Geant4 this concept is realized by using a base class `G4VSensitiveDetector`. The mandate of a sensitive detector is the construction of hit objects using information from steps along a particle track. The `G4VSensitiveDetector` receives a callback at the begin and the end of the event processing and at each step inside the active material whenever an energy deposition occurred.
The sensitive actions do not necessarily deal only the collection of energy deposits, but could also be used to simply monitor the performance of the active element e.g. by producing histograms of the absolute value or the spacial distribution of the depositions.
Within `DDG4` the concept of sensitive detectors is implemented as a configurable action sequence of type `Geant4SensDetActionSequence` calling members of the type `Geant4Sensitive` as shown in Figure 6. The actual processing part of such a sensitive action is only called if the and of a set of required filters of type `Geant4Filter` is positive (see also section 3.5.3). No filter is also positive. Possible filters are e.g. particle filters, which ignore the sensitive detector action if the particle is a `geantino` or if the energy deposit is below a given threshold.
Objects of type `Geant4Sensitive` receive the callbacks by overloading the member function:

```
/// Method invoked at the beginning of each event.
virtual void begin(G4HCofThisEvent* hce);
/// Method invoked at the end of each event.
virtual void end(G4HCofThisEvent* hce);
/// Method for generating hit(s) using the information of G4Step object.
virtual bool process(G4Step* step, G4TouchableHistory* history);
/// Method invoked if the event was aborted.
virtual void clear(G4HCofThisEvent* hce);
```

or register a callback with the signature `void (T::*)(G4HCofThisEvent*)` respectively `void (T::*)(G4Step*, G4TouchableHistory*)` to receive callbacks using the methods:

```
/// Register begin-of-event callback
template <typename T> void callAtBegin(T* p, void (T::*f)(G4HCofThisEvent*));
/// Register end-of-event callback
template <typename T> void callAtEnd(T* p, void (T::*f)(G4HCofThisEvent*));
/// Register process-hit callback
template <typename T> void callAtProcess(T* p, void (T::*f)(G4Step*, G4TouchableHistory*));
/// Register clear callback
template <typename T> void callAtClear(T* p, void (T::*f)(G4HCofThisEvent*));
```

Please refer to the Geant4 Applications manual from the Geant4 web page for further details about the concept of sensitive detectors.

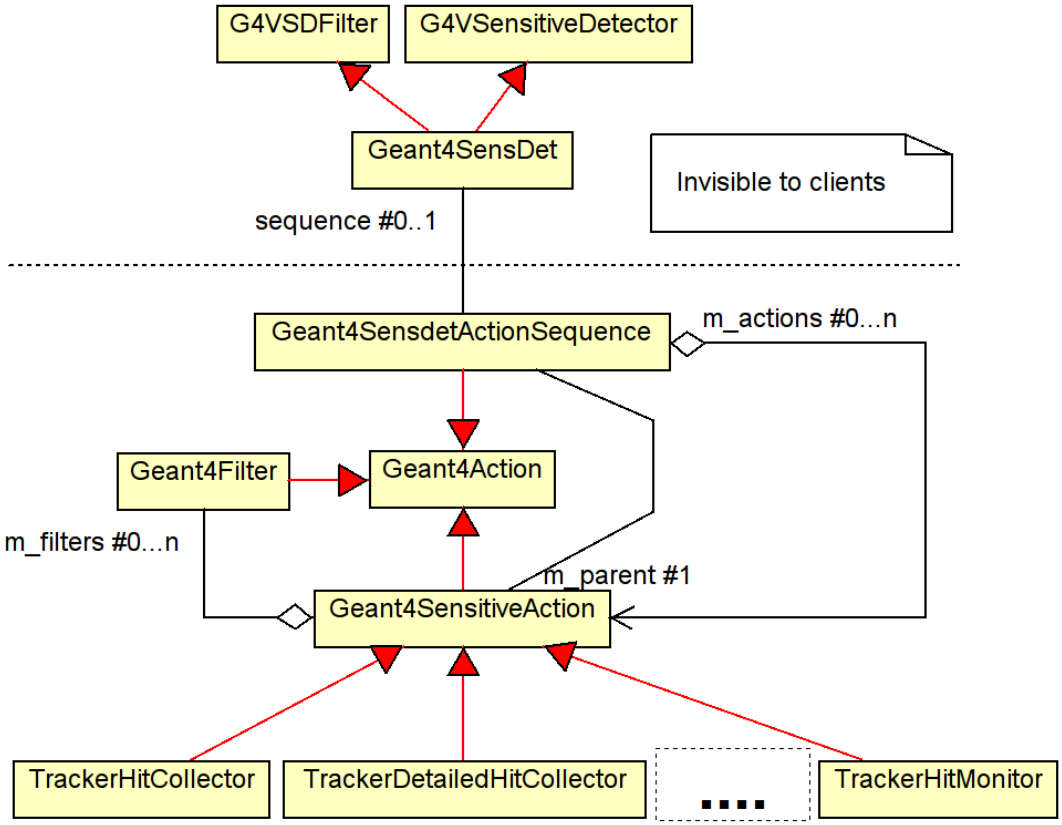

Figure 6: The sensitive detector design. The actual energy deposits are collected in user defined subclasses of the `Geant4Sensitive`. Here, as an example possible actions called `TrackerHitCollector`, `TrackerDetailedHitCollector` and `TrackerHitMonitor` are shown.

3.5.1 Helpers of Sensitive Detectors: The Geant4VolumeManager

Sooner or later, when a hit is created in a sensitive placed volume, the hit must be associated with this volume. For this purpose `DD4hep` provides the concept of the `VolumeManager`, which identifies placed volumes uniquely by a 64-bit identifier, the $CellID$. This mechanism allows to quickly retrieve a given volume given the hit data containing the $CellID$. The $CellID$ is a very compressed representation for any element in the hierarchy of placed volumes to the sensitive volume in question.
During the simulation the reverse mechanism must be applied: Geant4 provides the hierarchy of `G4PhysicalVolumes` to the hit location and the local coordinates of the hit within the sensitive volume. Hence to determine the volume identifier is essential to store hits so that they can be later accessed and processed efficiently. This mechanism is provided by the `Geant4VolumeManager`. Clients typically do not interact with this object, any access necessary is provided by the `Geant4Sensitive` action:

```
/// Method for generating hit(s) using the information of G4Step object.
bool MySensitiveAction:process(G4Step* step,G4TouchableHistory* /*hist*/ ) {
  ...
  Hit* hit = new Hit();
  // *** Retrieve the cellID  ***
  hit->cellID = cellID(step);
  ...
}
```

The call is realized using a member function provided by the `Geant4Sensitive` action:

```
/// Returns the cellID of the sensitive volume corresponding to the step
/** The CellID is the VolumeID + the local coordinates of the sensitive area.
 *  Calculated by combining the VolIDS of the complete geometry path (Geant4TouchableHistory)
 *  from the current sensitive volume to the world volume
 */
long long int cellID(G4Step* step);
```

Note:
The `Geant4VolumeManager` functionality is not for free! It requires that
– match Geant4 volume with TGeo volume

3.5.2 DDG4 Intrinsic Sensitive Detectors

Currently there are two generic sensitive detectors implemented in DDG4:

- The `Geant4TrackerAction`, which may be used to handle tracking devices. This sensitive detector produces one hit for every energy deposition of Geant4 i.e. for every callback to

  ```
  /// Method for generating hit(s) using the information of G4Step object.
  virtual bool process(G4Step* step, G4TouchableHistory* history);
  ```

  See the implementation file DDG4/plugins/Geant4SDAction.cpp for details. The produced hits are of type Geant4Tracker::Hit .
- The `Geant4CalorimeterAction`, which may be used to handle generic calorimeter like devices. This sensitive detector produces at most one hit for every cell in the calorimeter. If several tracks contribute to the energy deposit of this cell, the contributions are added up. See the implementation file DDG4/plugins/Geant4SDAction.cpp for details. The produced hits are of type Geant4Calorimeter::Hit . propagate the MC truth information with respect to each track kept in the particle record.

Both sensitive detectors use the `Geant4VolumeManager` discussed in section 3.5.1 to identify the sensitive elements.
PLEASE NOTE:
The above palette of generic sensitive detectors only contains two very often used implementations. We hope, that this palette over time grows from external contributions of other generic sensitive detectors. We would be happy to extend this palette with other generic implementations. One example would be the handling of the simulation response for optical detectors like RICH-Cerenkov detectors.

3.5.3 Sensitive Detector Filters

The concept of filters allows to build more flexible sensitive detectors by restricting the hit processing of a given instance of a sensitive action.

- Examples would be to demand a given particle type before a sensitive action is invoked: a sensitive action dealing with optical photons (RICH detectors, etc), would e.g. not be interested in energy depositions of other particles. A filter object restricting the particle type to optical photons would be appropriate.
- Another example would be to implement a special action instance, which would be only called if the filter requires a minimum energy deposit.

There are plenty of possible applications, hence we would like to introduce this feature here.
Filters are called by Geant4 before the hit processing in the sensitive detectors start. The global filters may be shared between many sensitive detectors. Alternatively filters may be directly attached to the sensitive detector in question. Attributes are directly passed as properties to the filter action.
Technically do `Geant4Filter` objects inherit from the base class `Geant4Filter` (see Figure 7. Any filter inherits from the common base class `Geant4Filter`, then several specializations may be configured like filters to select/reject particles, to specify the minimal energy deposit to be processed etc. A sensitive detector is called if the filter callback with the signature returns a true result:

```
/// Filter action. Return true if hits should be processed
virtual bool operator()(const G4Step* step) const;
```

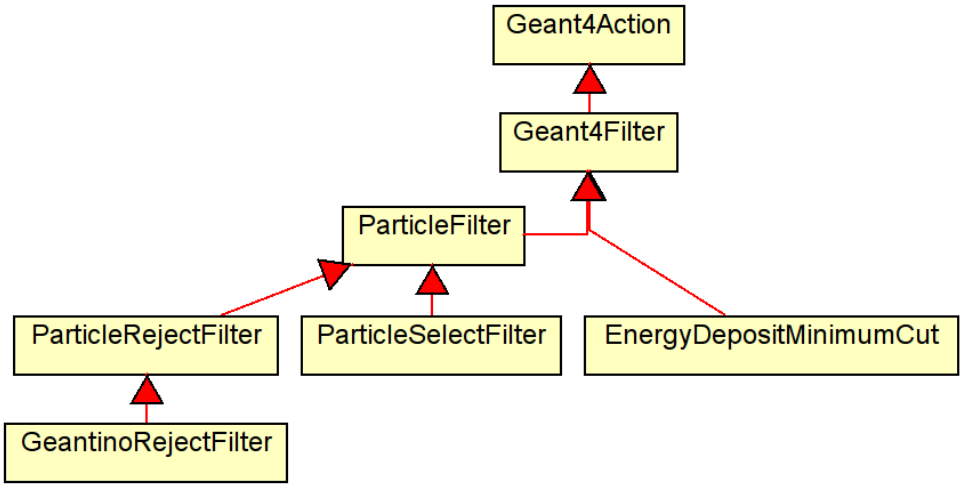

Figure 7: The sensitive detector filters design. The shown class diagram is actually implemented.

3.6 The Geant4 Physics List

Geant4 provides the base class `G4VUserPhysicsList`, which allows users to implement customized physics according to the studies to be made. Any user defined physics list must provide this interface. DDG4 provides such an interface through the ROOT plugin mechanism using the class `G4VModularPhysicsList`. The flexibility of `DDG4` allows for several possibilities to setup the Geant4 physics list. Instead of explicitly coding the physics list, `DDG4` foresees the usage of the plugin mechanism to instantiate the necessary calls to Geant4 in a sequence of actions:

- The **physics list** is realized as a sequence of actions of type `Geant4PhysicsListActionSequence` . Members of the `Geant4PhysicsListActionSequence` are of type `Geant4PhysicsList` and receive the callbacks by overloading the member functions:

  ```
  /// Callback to construct the physics constructors
  virtual void constructProcess(Geant4UserPhysics* interface);
  /// constructParticle callback
  virtual void constructParticles(Geant4UserPhysics* particle);
  /// constructPhysics callback
  virtual void constructPhysics(Geant4UserPhysics* physics);
  ```

  or register a callback with the signature `void (T::*)(Geant4UserPhysics*)` to receive calls using the method:

  ```
  /// Register process construction callback t
  template <typename Q, typename T> void constructProcess(Q* p, void (T::*f)(Geant4UserPhysics*));
  /// Register particle construction callback
  template <typename Q, typename T> void constructParticle(Q* p, void (T::*f)(Geant4UserPhysics*));
  ```

  The argument of type `Geant4UserPhysics` provides a basic interface to the original `G4VModularPhysicsList`, which allows to register physics constructors etc.

- In most of the cases the above approach is an overkill and often even too flexible. Hence, alternatively, the physics list may consist of a single entry of type `Geant4PhysicsList` .

The basic implementation of the `Geant4PhysicsList` supports the usage of various

- particle constructors , such as single particle constructors like `G4Gamma` or `G4Proton`, or whole particle groups like `G4BosonConstructor` or `G4IonConstrutor`,
- physics process constructors , such as e.g. `G4GammaConversion`, `G4PhotoElectricEffect` or `G4ComptonScattering`,
- physics constructors combining particles and the corresponding interactions, such as e.g. `G4OpticalPhysics`, `HadronPhysicsLHEP` or `G4HadronElasticPhysics` and
- predefined Geant4 physics lists , such as `FTFP_BERT`, `CHIPS` or `QGSP_INCLXX`. This option is triggered by the content of the string property "extends" of the `Geant4Kernel::physicsList()` action.

These constructors are internally connected to the above callbacks to register themselves. The constructors are instantiated using the ROOT plugin mechanism.
The description of the above interface is only for completeness. The basic idea is, that the physics list with its particle and physics constructors is configured entirely data driven using the setup mechanism described in the following chapter. However, DDG4 is not limited to the data driven approach. Specialized physics lists may be supplied, but there should be no need. New physics lists could always be composed by actually providing new physics constructors and actually publishing these using the factory methods:

```
1 // Framework include files
2 #include "DDG4/Factories.h"
3
4 #include "My_Very_Own_Physics_Constructor.h"
5 DECLARE_GEANT4_PHYSICS(My_Very_Own_Physics_Constructor)
```

where `My_Very_Own_Physics_Constructor` represents a sub-class of `G4VPhysicsConstructor`.

3.7 The Support of the Geant4 UI: `Geant4UIMessenger`

The support of interactivity in Geant4 is mandatory to debug detector setups in small steps. The Geant4 toolkit did provide for this reason a machinery of UI commands.

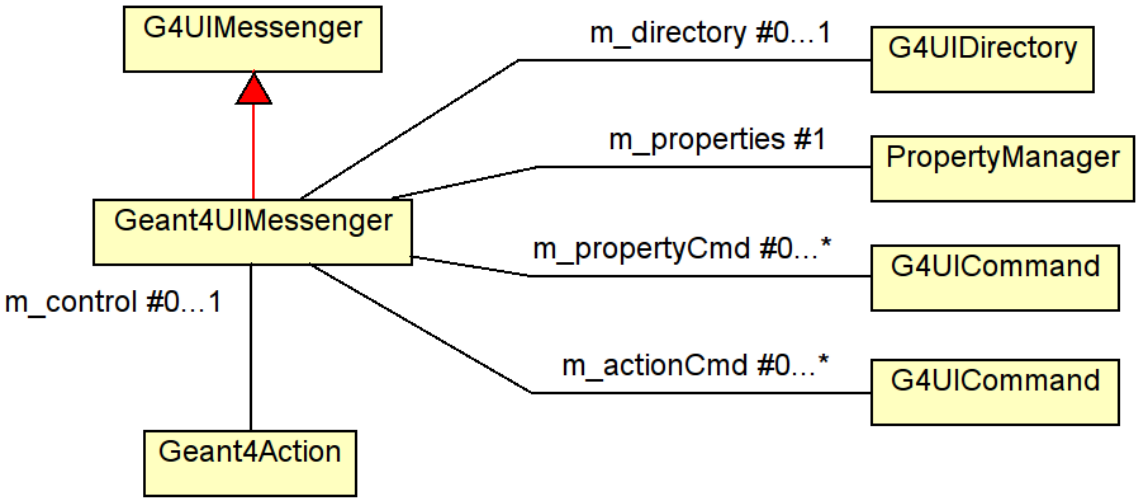

Figure 8: The design of the `Geant4UIMessenger` class responsible for the interaction between the user and the components of `DDG4` and Geant4.

The UI control is enabled, as soon as the property "Control" (boolean) is set to true. Be default all properties of the action are exported. Similar to the callback mechanism described above it is also feasible to register any object callback invoking a method of a `Geant4Action`-subclass.
The following (shortened) screen dump illustrates the usage of the generic interface any Geant4Action offers:

```
Idle> ls
Command directory path : /
 Sub-directories :
   /control/   UI control commands.
   /units/   Available units.
   /process/   Process Table control commands.
   /ddg4/   Control for all named Geant4 actions
   ...
Idle> cd /ddg4
Idle> ls
...
Control for all named Geant4 actions

 Sub-directories :
   /ddg4/RunInit/   Control hierarchy for Geant4 action:RunInit
   /ddg4/RunAction/   Control hierarchy for Geant4 action:RunAction
   /ddg4/EventAction/   Control hierarchy for Geant4 action:EventAction
   /ddg4/GeneratorAction/   Control hierarchy for Geant4 action:GeneratorAction
   /ddg4/LCIO1/   Control hierarchy for Geant4 action:LCIO1
   /ddg4/Smear1/   Control hierarchy for Geant4 action:Smear1
   /ddg4/PrimaryHandler/   Control hierarchy for Geant4 action:PrimaryHandler
   /ddg4/TrackingAction/   Control hierarchy for Geant4 action:TrackingAction
   /ddg4/SteppingAction/   Control hierarchy for Geant4 action:SteppingAction
   /ddg4/ParticleHandler/   Control hierarchy for Geant4 action:ParticleHandler
```

```
   /ddg4/UserParticleHandler/   Control hierarchy for Geant4 action:UserParticleHandler
   ...
Idle> ls Smear1
Command directory path : /ddg4/Smear1/
 ...
 Commands :
   show * Show all properties of Geant4 component:Smear1
   Control * Property item of type bool
   Mask * Property item of type int
   Name * Property item of type std::string
   Offset * Property item of type ROOT::Math::LorentzVector<ROOT::Math::PxPyPzE4D<double> >
   OutputLevel * Property item of type int
   Sigma * Property item of type ROOT::Math::LorentzVector<ROOT::Math::PxPyPzE4D<double> >
   name * Property item of type std::string
Idle> Smear1/show
PropertyManager: Property Control = True
PropertyManager: Property Mask = 1
PropertyManager: Property Name = 'Smear1'
PropertyManager: Property Offset = ( -20 , -10 , -10 , 0 )
PropertyManager: Property OutputLevel = 4
PropertyManager: Property Sigma = ( 12 , 8 , 8 , 0 )
PropertyManager: Property name = 'Smear1'

Idle> Smear1/Offset (200*mm, -3*mm, 15*mm, 10*ns)
Geant4UIMessenger: +++ Smear1> Setting property value Offset = (200*mm, -3*mm, 15*mm, 10*ns)
                               native:( 200 , -3 , 15 , 10 ).
Idle> Smear1/show
...
PropertyManager: Property Offset = ( 200 , -3 , 15 , 10 )
```

4 Setting up DDG4

DDG4 offers several possibilities to configure a simulation application using

- XML files,
- by coding a setup script loaded from the ROOT interpreter with the AClick mechanism.
- by creating a setup script using python and ROOT's reflection mechanism exposed by PyROOT.

The following subsection describe these different mechanism. An attempt was made to match the naming conventions of all approaches where possible.

4.1 Setting up DDG4 using XML

A special plugin was developed to enable the configuration of DDG4 using XML structures. These files are parsed identically to the geometry setup in DD4hep the only difference is the name of the root-element, which for DDG4 is <geant4_setup>. The following code snippet shows the basic structure of a DDG4 setup file:

```
<geant4_setup>
  <physicslist>          ,,,  </physicslist>  <!-- Definition of the physics list          -->
  <actions>              ...  </actions>      <!-- The list of global actions              -->
  <phases>               ...  </phases>       <!-- The definition of the various phases    -->
  <filters>              ...  </filters>      <!-- The list of global filter actions       -->
  <sequences>            ...  </sequences>    <!-- The list of defined sequences           -->
  <sensitive_detectors>  ...  </sensitive_detectors>  <!-- The list of sensitive detectors -->
  <properties>           ...  </properties>   <!-- Free format option sequences            -->
</geant4_setup>
```

To setup a DDG4 4 application any number of xml setup files may be interpreted iteratively. In the following subsections the content of these first level sub-trees will be discussed.

4.1.1 Setup of the Physics List

The main tag to setup a physics list is <physicslist> with the name attribute defining the instance of the Geant4PhysicsList object. An example code snippet is shown below in Figure 9.

```
1 <geant4_setup>
2   <physicslist name="Geant4PhysicsList/MyPhysics.0">
3
4     <extends name="QGSP_BERT"/>                    <!-- Geant4 basic Physics list -->
5
6     <particles>                                    <!-- Particle constructors     -->
7       <construct name="G4Geantino"/>
8       <construct name="G4ChargedGeantino"/>
9       <construct name="G4Electron"/>
10      <construct name="G4Gamma"/>
11      <construct name="G4BosonConstructor"/>
12      <construct name="G4LeptonConstructor"/>
13      <construct name="G4MesonConstructor"/>
14      <construct name="G4BaryonConstructor"/>
15      ...
16    </particles>
17
18    <processes>                                    <!-- Process constructors      -->
19      <particle name="e[+-]" cut="1*mm">
20        <process name="G4eMultipleScattering"  ordAtRestDoIt="-1"       ordAlongSteptDoIt="1"
21                                               ordPostStepDoIt="1"/>
22        <process name="G4eIonisation"          ordAtRestDoIt="-1"       ordAlongSteptDoIt="2"
```

```
23                                                ordPostStepDoIt="2"/>
24      </particle>
25      <particle name="mu[+-]">
26        <process name="G4MuMultipleScattering" ordAtRestDoIt="-1"       ordAlongSteptDoIt="1"
27                                               ordPostStepDoIt="1"/>
28        <process name="G4MuIonisation"         ordAtRestDoIt="-1"       ordAlongSteptDoIt="2"
29                                               ordPostStepDoIt="2"/>
30      </particle>
31      ...
32    </processes>
33
34    <physics>                                      <!-- Physics constructors      -->
35      <construct name="G4EmStandardPhysics"/>
36      <construct name="HadronPhysicsQGSP"/>
37      ...
38    </physics>
39
40  </physicslist>
41 </geant4_setup>
```

Figure 9: XML snippet showing the configuration of a physics list.

- To base all these constructs on an already existing predefined Geant4 physics list use the `<extends>` tag with the attribute containing the name of the physics list as shown in line 4.
- To trigger a call to a **particle constructors** (line 7-14), use the `<particles>` section and define the Geant4 particle constructor to be called by name. To trigger a call to
- **physics process constructors**, as shown in line 19-30, Define for each particle matching the name pattern (regular expression!) and the default cut value for the corresponding processes. The attributes ordXXXX correspond to the arguments of the Geant4 call `G4ProcessManager::AddProcess(process,ordAtRestDoIt, ordAlongSteptDoIt,ordPostStepDoIt);` The processes themself are created using the ROOT plugin mechanism. To trigger a call to
- **physics constructors**, as shown in line 34-35, use the `<physics>` section.

If only a predefined physics list is used, which probably already satisfies very many use cases, all these section collapse to:

```
1 <geant4_setup>
2   <physicslist name="Geant4PhysicsList/MyPhysics.0">
3     <extends name="QGSP_BERT"/>                  <!-- Geant4 basic Physics list -->
4   </physicslist>
5 </geant4_setup>
```

4.1.2 Setup of Global Geant4 Actions

Global actions must be defined in the `<actions>` section as shown in the following snippet:

```
1 <geant4_setup>
2   <actions>
3     <action name="Geant4TestRunAction/RunInit">
4       <properties Property_int="12345"
5           Property_double="-5e15"
6           Property_string="Startrun: Hello_2"/>
7     </action>
```

```
 8    <action name="Geant4TestEventAction/UserEvent_2"
 9            Property_int="1234"
10            Property_double="5e15"
11            Property_string="Hello_2" />
12  </actions>
13 </geant4_setup>
```

The default properties of **every** `Geant4Action` object are:

```
Name        [string]              Action name
OutputLevel [int]                 Flag to customize the level of printout
Control     [boolean]             Flag if the UI messenger should be installed.
```

The `name` attribute of an action child is a qualified name: The first part denotes the type of the plugin (i.e. its class), the second part the name of the instance. Within one collection the instance `name` must be unique. Properties of Geant4Actions are set by placing them as attributes into the `<properties>` section.

4.1.3 Setup of Geant4 Filters

Filters are special actions called by `Geant4Sensitives`. Filters may be global or anonymous i.e. reusable by several sensitive detector sequences as illustrated in Section 4.1.4. The setup is analogous to the setup of global actions:

```
 1  <filters>
 2    <filter name="GeantinoRejectFilter/GeantinoRejector"/>
 3    <filter name="ParticleRejectFilter/OpticalPhotonRejector">
 4      <properties particle="opticalphoton"/>
 5    </filter>
 6    <filter name="ParticleSelectFilter/OpticalPhotonSelector">
 7      <properties particle="opticalphoton"/>
 8    </filter>
 9    <filter name="EnergyDepositMinimumCut">
10      <properties Cut="10*MeV"/>
11    </filter>
12    <!--        ... next global filter ...       -->
13  </filters>
```

Global filters are accessible from the `Geant4Kernel` object.

4.1.4 Geant4 Action Sequences

`Geant4 Action Sequences` by definition are `Geant4Action` objects. Hence, they share the setup mechanism with properties etc. For the setup mechanism two different types of sequences are known to `DDG4` : *Action sequences* and *Sensitive detector sequences*. Bot are declared in the `sequences` section:

```
 1 <geant4_setup>
 2  <sequences>
 3    <sequence name="Geant4EventActionSequence/EventAction"> <!-- Sequence "EventAction" of type
 4                                                                 "Geant4EventActionSequence" -->
 5      <action name="Geant4TestEventAction/UserEvent_1">     <!-- Anonymous action                  -->
 6        <properties Property_int="01234"                    <!-- Properties go inline              -->
 7            Property_double="1e11"
 8            Property_string="'Hello_1'"/>
 9      </action>
10      <action name="UserEvent_2"/>                          <!-- Global action defined in "actions" -->
11                                                            <!-- Only the name is referenced here   -->
12      <action name="Geant4Output2ROOT/RootOutput">          <!-- ROOT I/O action                    -->
```

```
13        <properties Output="simple.root"/>                <!-- Output file property              -->
14      </action>
15      <action name="Geant4Output2LCIO/LCIOOutput">        <!-- LCIO output action                 -->
16        <properties Output="simple.lcio"/>                <!-- Output file property              -->
17      </action>
18    </sequence>
19
20
21    <sequence sd="SiTrackerBarrel" type="Geant4SensDetActionSequence">
22      <filter name="GeantinoRejector"/>
23      <filter name="EnergyDepositMinimumCut"/>
24      <action name="Geant4SimpleTrackerAction/SiTrackerBarrelHandler"/>
25    </sequence>
26    <sequence sd="SiTrackerEndcap" type="Geant4SensDetActionSequence">
27      <filter name="GeantinoRejector"/>
28      <filter name="EnergyDepositMinimumCut"/>
29      <action name="Geant4SimpleTrackerAction/SiTrackerEndcapHandler"/>
30    </sequence>
31    <!--    ... next sequence ...     -->
32  </sequences>
33 </geant4_setup>
```

Here firstly the **EventAction** sequence is defined with its members. Secondly a sensitive detector sequence is defined for the subdetector `SiTrackerBarrel` of type `Geant4SensDetActionSequence`. The sequence uses two filters: `GeantinoRejector` to not generate hits from geantinos and `EnergyDepositMinimumCut` to enforce a minimal energy deposit. These filters are global i.e. they may be applied by many subdetectors. The setup of global filters is described in Section 4.1.3. Finally the action `SiTrackerEndcapHandler` of type `Geant4SimpleTrackerAction` is chained, which collects the deposited energy and creates a collection of hits. The `Geant4SimpleTrackerAction` is a template callback to illustrate the usage of sensitive elements in `DDG4` . The resulting hit collection of these handlers by default have the same name as the object instance name. Analogous below the sensitive detector sequence for the subdetector `SiTrackerEndcap` is shown, which reuses the same filter actions, but will build its own hit collection.
Please note:

- **It was already mentioned, but once again**: Event-, run-, generator-, tracking-, stepping- and stacking actions sequences have predefined names! These names are fixed and part of the **common knowledge**, they cannot be altered. Please refer to Section 3.4 for the names of the global action sequences.
- the sensitive detector sequences are matched by the attribute `sd` to the subdetectors created with the `DD4hep` detector description package. Values must match!
- In the event that several xml files are parsed it is absolutely vital that the `<actions>` section is interpreted **before** the `sequences`.
- For each XML file several `<sequences>` are allowed.

4.1.5 Setup of Geant4 Sensitive Detectors

```
1  <geant4_setup>
2    <sensitive_detectors>
3      <sd name="SiTrackerBarrel"
4          type="Geant4SensDet"
5          ecut="10.0*MeV"
6          verbose="true"
7          hit_aggregation="position">
8      </sd>
9      <!-- ...  next sensitive detector ... -->
10   </sensitive_detectors>
11 </geant4_setup>
```

4.1.6 Miscellaneous Setup of Geant4 Objects

This section is used for the flexible setup of auxiliary objects such as the electromagnetic fields used in Geant4:

```
<geant4_setup>
  <properties>
    <attributes name="geant4_field"
          id="0"
          type="Geant4FieldSetup"
          object="GlobalSolenoid"
          global="true"
          min_chord_step="0.01*mm"
          delta_chord="0.25*mm"
          delta_intersection="1e-05*mm"
          delta_one_step="0.001*mm"
          eps_min="5e-05*mm"
          eps_max="0.001*mm"
          largest_step = "10*m"
          stepper="HelixSimpleRunge"
          equation="Mag_UsualEqRhs">
    </attributes>
    ...
  </properties>
</geant4_setup>
```

Important are the tags `type` and `object`, which are used to firstly define the plugin to be called and secondly define the object from the `DD4hep` description to be configured for the use within Geant4.

4.1.7 Setup of Geant4 Phases

Phases are configured as shown below. However, the use is **discouraged**, since it is not yet clear if there are appropriate use cases!

```
<phases>
  <phase type="RunAction/begin">
    <action name="RunInit"/>
    <action name="Geant4TestRunAction/UserRunInit">
  <properties Property_int="1234"
          Property_double="5e15"
          Property_string="'Hello_2'"/>
    </action>
  </phase>
  <phase type="EventAction/begin">
    <action name="UserEvent_2"/>
  </phase>
  <phase type="EventAction/end">
    <action name="UserEvent_2"/>
  </phase>
  ...
</phases>
```

4.2 Setting up DDG4 using ROOT-CINT

The setup of DDG4 directly from the the ROOT interpreter using the AClick mechanism is very simple, but mainly meant for purists (like me ;-)), since it is nearly equivalent to the explicit setup within a C++ main program. The following code section shows how to do it. For explanation the code segment is discussed below line by line.

```
 1 #include "DDG4/Geant4Config.h"
 2 #include "DDG4/Geant4TestActions.h"
 3 #include "DDG4/Geant4TrackHandler.h"
 4 #include <iostream>
 5
 6 using namespace std;
 7 using namespace DD4hep;
 8 using namespace DD4hep::Simulation;
 9 using namespace DD4hep::Simulation::Test;
10 using namespace DD4hep::Simulation::Setup;
11
12 #if defined(__MAKECINT__)
13 #pragma link C++ class Geant4RunActionSequence;
14 #pragma link C++ class Geant4EventActionSequence;
15 #pragma link C++ class Geant4SteppingActionSequence;
16 #pragma link C++ class Geant4StackingActionSequence;
17 #pragma link C++ class Geant4GeneratorActionSequence;
18 #pragma link C++ class Geant4Action;
19 #pragma link C++ class Geant4Kernel;
20 #endif
21
22 SensitiveSeq::handled_type* setupDetector(Kernel& kernel, const std::string& name)   {
23   SensitiveSeq sd = SensitiveSeq(kernel,name);
24   Sensitive  sens = Sensitive(kernel,"Geant4TestSensitive/"+name+"Handler",name);
25   sd->adopt(sens);
26   sens = Sensitive(kernel,"Geant4TestSensitive/"+name+"Monitor",name);
27   sd->adopt(sens);
28   return sd;
29 }
30
31 void exampleAClick()  {
32   Geant4Kernel& kernel = Geant4Kernel::instance(LCDD::getInstance());
33   kernel.loadGeometry("file:../DD4hep.trunk/DDExamples/CLICSiD/compact/compact.xml");
34   kernel.loadXML("DDG4_field.xml");
35
36   GenAction gun(kernel,"Geant4ParticleGun/Gun");
37   gun["energy"] = 0.5*GeV;                          // Set properties
38   gun["particle"] = "e-";
39   gun["multiplicity"] = 1;
40   kernel.generatorAction().adopt(gun);
41
42   Action run_init(kernel,"Geant4TestRunAction/RunInit");
43   run_init["Property_int"] = 12345;
44   kernel.runAction().callAtBegin  (run_init.get(),&Geant4TestRunAction::begin);
45   kernel.eventAction().callAtBegin(run_init.get(),&Geant4TestRunAction::beginEvent);
46   kernel.eventAction().callAtEnd  (run_init.get(),&Geant4TestRunAction::endEvent);
47
48   Action evt_1(kernel,"Geant4TestEventAction/UserEvent_1");
49   evt_1["Property_int"] = 12345;                    // Set properties
50   evt_1["Property_string"] = "Events";
51   kernel.eventAction().adopt(evt_1);
```

```
52
53  EventAction evt_2(kernel,"Geant4TestEventAction/UserEvent_2");
54  kernel.eventAction().adopt(evt_2);
55
56  kernel.runAction().callAtBegin(evt_2.get(),&Geant4TestEventAction::begin);
57  kernel.runAction().callAtEnd  (evt_2.get(),&Geant4TestEventAction::end);
58
59  setupDetector(kernel,"SiVertexBarrel");
60  setupDetector(kernel,"SiVertexEndcap");
61  // .... more subdetectors here .....
62  setupDetector(kernel,"LumiCal");
63  setupDetector(kernel,"BeamCal");
64
65  kernel.configure();
66  kernel.initialize();
67  kernel.run();
68  std::cout << "Successfully executed application .... " << std::endl;
69  kernel.terminate();
70 }
```

Line	
1	The header file `Geant4Config.h` contains a set of wrapper classes to easy the creation of objects using the plugin mechanism and setting properties to `Geant4Action` objects. These helpers and the corresponding functionality are not included in the wrapped classes themselves to not clutter the code with stuff only used for the setup. All contained objects are in the namespace `DD4hep::Simulation::Setup`
. 6-10	Save yourself specifying all the namespaces objects are in....
13-19	CINT processing pragmas. Classes defined here will be available at the ROOT prompt after this AClick is loaded.
22-29	Sampler to fill the sensitive detector sequences for each subdetector with two entries: a handler and a monitor action. Please note, that this here is example code and in real life specialized actions will have to be provided for each subdetector.
31	Let's go for it. here the entry point starts....
32	Create the `Geant4Kernel` object.
33	Load the geometry into `DD4hep` .
34	Redefine the setup of the sensitive detectors.
36-40	Create the generator action of type `Geant4ParticleGun` with name `Gun`, set non-default properties and activate the configured object by attaching it to the `Geant4Kernel`.
42-46	Create a user defined begin-of-run action callback, set the properties and attach it to the begin of run calls. To collect statistics extra member functions are registered to be called at the beginning and the end of each event.
48-51	Create a user defined event action routine, set its properties and attach it to the event action sequence.
53-54	Create a second event action and register it to the event action sequence. This action will be called after the previously created action.
56-57	For this event action we want to receive callbacks at start- and end-of-run to produce additional summary output.
59-63	Call the sampler routine to attach test actions to the subdetectors defined.
65-66	Configure, initialize and run the Geant4 application. Most of the Geant4 actions will only be created here and the action sequences created before will be attached now.
69	Terminate the Geant4 application and exit.

CINT currently cannot handle pointers to member functions [1]. Hence the above AClick only works in compiled mode. To invoke the compilation the following action is necessary from the ROOT prompt:

```
$> root.exe
  ****************************************
  *                                      *
  *        W E L C O M E  to  R O O T    *
  *                                      *
  *   Version   5.34/10    29 August 2013   *
  *                                      *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch         *
  *                                      *
  ****************************************

ROOT 5.34/10 (heads/v5-34-00-patches@v5-34-10-5-g0e8bac8, Sep 04 2013, 11:52:19 on linux)

CINT/ROOT C/C++ Interpreter version 5.18.00, July 2, 2010
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] .X initAClick.C
.... Setting up the CINT include pathes and the link statements.

root [1] .L ../DD4hep.trunk/DDG4/examples/exampleAClick.C+
Info in <TUnixSystem::ACLiC>: creating shared library ....exampleAClick_C.so
.... some Cint warnings concerning member function pointers .....

root [2] exampleAClick()
.... and it starts ...
```

The above scripts are present in the DDG4/example directory located in svn. The initialization script `initAClick.C` may require customization to cope with the installation paths.

4.3 Setting up DDG4 using Python

Given the reflection interface of ROOT, the setup of the simulation interface using DD4hep is of course also possible using the python interpreted language. In the following code example the setup of Geant4 using the `ClicSid` example is shown using python [2].

```
import DDG4
from SystemOfUnits import *

"""

  DD4hep example setup using the python configuration

  @author  M.Frank
  @version 1.0

"""
def run():
  kernel = DDG4.Kernel()
  kernel.loadGeometry("file:../DD4hep.trunk/DDExamples/CLICSiD/compact/compact.xml")
  kernel.loadXML("DDG4_field.xml")

```

[1] This may change in the future once ROOT uses `clang` and `cling` as the interpreting engine.
[2] For comparison, the same example was used to illustrate the setup using XML files.

```
17  lcdd = kernel.lcdd()
18  print '+++   List of sensitive detectors:'
19  for i in lcdd.detectors():
20    o = DDG4.DetElement(i.second)
21    sd = lcdd.sensitiveDetector(o.name())
22    if sd.isValid():
23      print '+++  %-32s type:%s'%(o.name(), sd.type(), )
24
25  # Configure Run actions
26  run1 = DDG4.RunAction(kernel,'Geant4TestRunAction/RunInit')
27  run1.Property_int    = 12345
28  run1.Property_double = -5e15*keV
29  run1.Property_string = 'Startrun: Hello_2'
30  print run1.Property_string, run1.Property_double, run1.Property_int
31  run1.enableUI()
32  kernel.registerGlobalAction(run1)
33  kernel.runAction().add(run1)
34
35  # Configure Event actions
36  evt2 = DDG4.EventAction(kernel,'Geant4TestEventAction/UserEvent_2')
37  evt2.Property_int    = 123454321
38  evt2.Property_double = 5e15*GeV
39  evt2.Property_string = 'Hello_2 from the python setup'
40  evt2.enableUI()
41  kernel.registerGlobalAction(evt2)
42
43  evt1 = DDG4.EventAction(kernel,'Geant4TestEventAction/UserEvent_1')
44  evt1.Property_int=01234
45  evt1.Property_double=1e11
46  evt1.Property_string='Hello_1'
47  evt1.enableUI()
48
49  kernel.eventAction().add(evt1)
50  kernel.eventAction().add(evt2)
51
52  # Configure I/O
53  evt_root = DDG4.EventAction(kernel,'Geant4Output2ROOT/RootOutput')
54  evt_root.Control = True
55  evt_root.Output = "simple.root"
56  evt_root.enableUI()
57
58  evt_lcio = DDG4.EventAction(kernel,'Geant4Output2LCIO/LcioOutput')
59  evt_lcio.Output = "simple_lcio"
60  evt_lcio.enableUI()
61
62  kernel.eventAction().add(evt_root)
63  kernel.eventAction().add(evt_lcio)
64
65  # Setup particle gun
66  gun = DDG4.GeneratorAction(kernel,"Geant4ParticleGun/Gun")
67  gun.energy   = 0.5*GeV
68  gun.particle = 'e-'
69  gun.multiplicity = 1
70  gun.enableUI()
71  kernel.generatorAction().add(gun)
72
73  # Setup global filters for use in sensitive detectors
74  f1 = DDG4.Filter(kernel,'GeantinoRejectFilter/GeantinoRejector')
```

```
75  f2 = DDG4.Filter(kernel,'ParticleRejectFilter/OpticalPhotonRejector')
76  f2.particle = 'opticalphoton'
77  f3 = DDG4.Filter(kernel,'ParticleSelectFilter/OpticalPhotonSelector')
78  f3.particle = 'opticalphoton'
79  f4 = DDG4.Filter(kernel,'EnergyDepositMinimumCut')
80  f4.Cut = 10*MeV
81  f4.enableUI()
82  kernel.registerGlobalFilter(f1)
83  kernel.registerGlobalFilter(f2)
84  kernel.registerGlobalFilter(f3)
85  kernel.registerGlobalFilter(f4)
86
87  # First the tracking detectors
88  seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiVertexBarrel')
89  act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiVertexBarrelHandler','SiVertexBarrel')
90  seq.add(act)
91  seq.add(f1)
92  seq.add(f4)
93  act.add(f1)
94
95  seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiVertexEndcap')
96  act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiVertexEndcapHandler','SiVertexEndcap')
97  seq.add(act)
98  seq.add(f1)
99  seq.add(f4)
100
101 seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiTrackerBarrel')
102 act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiTrackerBarrelHandler','SiTrackerBarrel')
103 seq.add(act)
104 seq.add(f1)
105 seq.add(f4)
106
107 seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiTrackerEndcap')
108 act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiTrackerEndcapHandler','SiTrackerEndcap')
109 seq.add(act)
110
111 seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/SiTrackerForward')
112 act = DDG4.SensitiveAction(kernel,'Geant4SimpleTrackerAction/SiTrackerForwardHandler','SiTrackerForward')
113 seq.add(act)
114
115 # Now the calorimeters
116 seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/EcalBarrel')
117 act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/EcalBarrelHandler','EcalBarrel')
118 seq.add(act)
119
120 seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/EcalEndcap')
121 act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/EcalEndCapHandler','EcalEndcap')
122 seq.add(act)
123
124 seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/HcalBarrel')
125 act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/HcalBarrelHandler','HcalBarrel')
126 act.adoptFilter(kernel.globalFilter('OpticalPhotonRejector'))
127 seq.add(act)
128
129 act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/HcalOpticalBarrelHandler','HcalBarrel')
130 act.adoptFilter(kernel.globalFilter('OpticalPhotonSelector'))
131 seq.add(act)
132
```

```
133   seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/HcalEndcap')
134   act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/HcalEndcapHandler','HcalEndcap')
135   seq.add(act)
136
137   seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/HcalPlug')
138   act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/HcalPlugHandler','HcalPlug')
139   seq.add(act)
140
141   seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/MuonBarrel')
142   act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/MuonBarrelHandler','MuonBarrel')
143   seq.add(act)
144
145   seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/MuonEndcap')
146   act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/MuonEndcapHandler','MuonEndcap')
147   seq.add(act)
148
149   seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/LumiCal')
150   act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/LumiCalHandler','LumiCal')
151   seq.add(act)
152
153   seq = DDG4.SensitiveSequence(kernel,'Geant4SensDetActionSequence/BeamCal')
154   act = DDG4.SensitiveAction(kernel,'Geant4SimpleCalorimeterAction/BeamCalHandler','BeamCal')
155   seq.add(act)
156
157   # Now build the physics list:
158   phys = kernel.physicsList()
159   phys.extends = 'FTFP_BERT'
160   #phys.transportation = True
161   phys.decays  = True
162   phys.enableUI()
163
164   ph = DDG4.PhysicsList(kernel,'Geant4PhysicsList/Myphysics')
165   ph.addParticleConstructor('G4BosonConstructor')
166   ph.addParticleConstructor('G4LeptonConstructor')
167   ph.addParticleProcess('e[+-]','G4eMultipleScattering',-1,1,1)
168   ph.addPhysicsConstructor('G4OpticalPhysics')
169   ph.enableUI()
170   phys.add(ph)
171
172   phys.dump()
173
174   kernel.configure()
175   kernel.initialize()
176   kernel.run()
177   kernel.terminate()
178
179 if __name__ == "__main__":
180   run()
181
```

4.4 A Simple Example

Bla-bal.

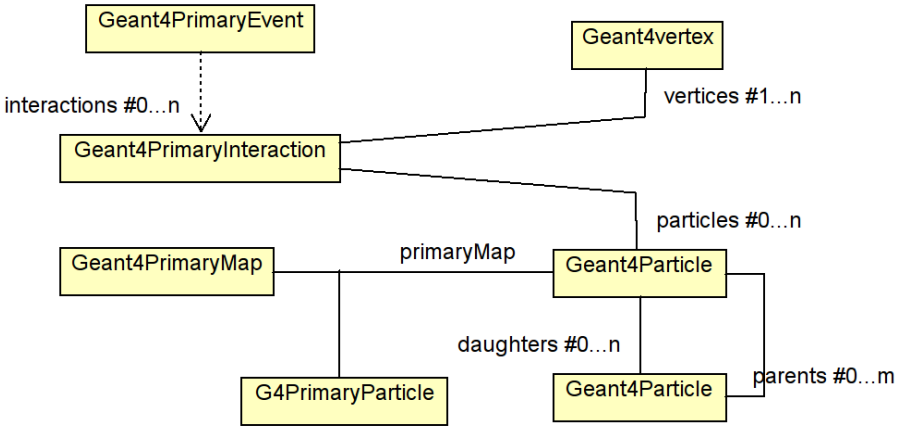

Figure 10: The DDG4 event data model.

5 Higher Level Components

Layered components, which base on the general framework implement higher level functionality such as the handling of Monte-Carlo truth associations between simulated energy deposits and the corresponding particles or the generic handling of input to the simulation.
To generalize such common behavior it is mandatory that the participating components collaborate and understand the data components they commonly access. The data model is shown in Figure 10.
Please note, that this data model is by no means to be made persistent and used for physics user analysis. This model is optimized to support the simulation process and the necessary user actions to handle MC truth, to easily and relatively fast look up and modify parent-daughter relationships etc. This choice is based on the assumption, that the additional overhead to convert particles at the input/output stage is small compared to the actual resource consumption of Geant4 to simulate the proper detector response. On the other hand this choice has numerous advantages:

- Accepting the fact to convert input records allows to adapt DDG4 in a simple and flexible manner to any input format. Currently supported is the input from raw `LCIO` files, `StdHep` records using `LCIO` and `ASCII` files using the `HEPEvt` format.
- Similarly as for the input stage, also the output format can be freely chosen by the clients.
- No assumptions was made concerning the structure to store information from energy deposits. Any information extract produced by the sensitive actions can be adapted provided at the output stage the proper conversion mechanism is present. The sensitive detectors provided by DDG4 are **optional only and by no means mandatory**. User defined classes may be provided at any time. Appropriate tools to extract MC truth information is provided at the output stage.
- The modular approach of the action sequences described in 3.4 allows to easily extend the generation sequence to handle multiple simultaneous interactions, event overlay or spillover response very easily [3]

In section 5.1 the generic mechanism of input data handling is described.
In section 5.3 the MC truth handling is discussed.
In section 6 we describe the output mechanism.

[3]The handling of spillover is only possible if during the digitization step the correct signal shape corresponding to the shift of signal creation is taken into account.

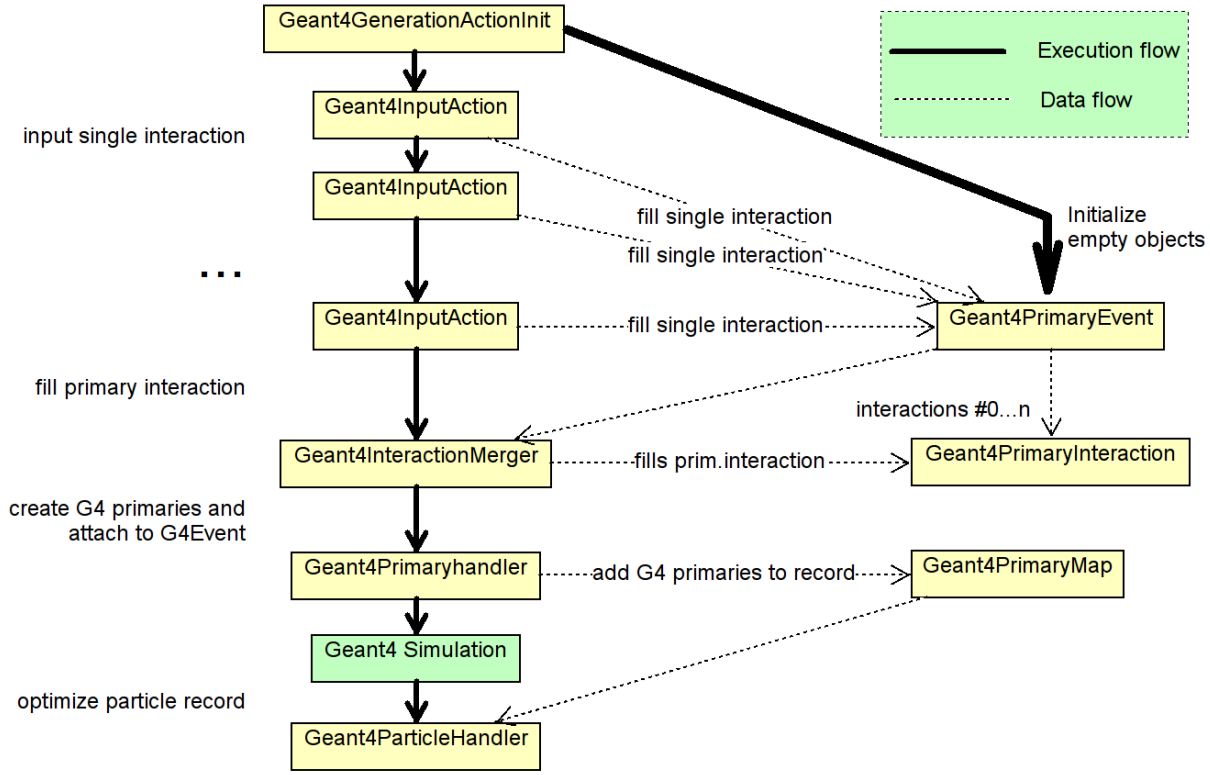

Figure 11: The generic handling of input sources to DDG4.

5.1 Input Data Handling

Input handling has several stages and uses several modules:

- First the data structures `Geant4PrimaryEvent`, `Geant4PrimaryInteraction` and `Geant4PrimaryMap` are initialized by the action `Geant4GenerationActionInit` and attached to the `Geant4Event` structure.
- The initialization is then followed by any number of input modules. Typically each input module add one interaction. Each interaction has a unique identifier, which is propagated later to all particles. Hence all primary particles can later be unambiguously be correlated to one of the initial interactions. Each instance of a `Geant4InputAction` creates and fills a separate instance of a `Geant4PrimaryInteraction`. In section 5.2 the functionality and extensions are discussed in more detail.
- All individual primary interactions are then merged to only single record using the `Geant4-InteractionMerger` component. This components fills the `Geant4PrimaryInteraction` registered to the `Geant4Event`, which serves as input record for the next component,
- the `Geant4PrimaryHandler`. The primary handler creates the proper `G4PrimaryParticle` and `G4PrimaryVertex` objects passed to `Geant4`. After this step all event related user interaction with Geant4 has completed, and the detector simulation may commence.

All modules used are subclasses of the `Geant4GeneratorAction` and must be added to the `Geant4-GeneratorActionSequence` as described in 3.4.
An object of type `Geant4PrimaryEvent` exists exactly once for every event to be simulated. The empty `Geant4PrimaryEvent` is created by the `Geant4GenerationActionInit` component. All higher level components may then access the `Geant4PrimaryEvent` object and subsequently an individual interaction from

the `Geant4Context` using the extension mechanism as shown in the following code:

```
1 /// Event generation action callback
2 void SomeGenerationComponent::operator()(G4Event* event)  {
3   /// Access the primary event object from the context
4   Geant4PrimaryEvent* evt = context()->event().extension<Geant4PrimaryEvent>();
5   /// Access the container of interactions
6   const std::vector<Geant4PrimaryEvent::Interaction*>& inter = evt->interactions();
7   /// Access one single interaction to be manipulated by this component
8   Geant4PrimaryInteraction* evt->get(m_myInteraction_identifier);
9   ....
```

Please note: To keep components simple, each component should only act on one interaction the component has to uniquely identify. The identification may be implemented by e.g. an access mask passed to the component as a property.

5.2 Anatomy of the Input Action

One input action fills one primary interaction. `Geant4InputAction` instances may be followed by decorators, which allow to to smear primary vertex (`Geant4InteractionVertexSmear`) or to boost the primary vertex `Geant4InteractionVertexBoost` and all related particles/vertices.

Please note, that a possible reduction of particles in the output record may break this unambiguous relationship between "hits" and particles. .....

5.3 Monte-Carlo Truth Handling

.....

6 Output Data Handling

.....

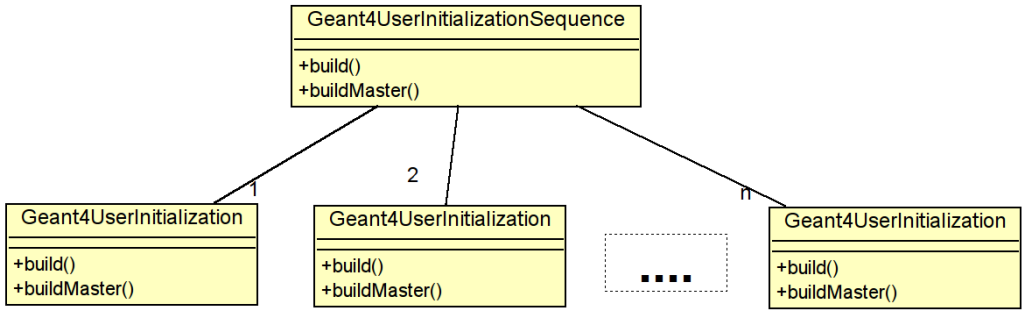

Figure 12: The Geant4 user initialization sequence to setup DDG4 in multi-threaded mode. The callbacks `buildMaster()` is only called in multi-threaded mode.

7 Multi-Threading in DDG4

7.1 Introductory Remarks

Multi-threading as supported by Geant4 is event driven. This means that the simulation of a given event is handled by one single thread. Geant4 provides specific extensions to ease the users the use of its multi-threaded extensions [13] [4]. These extension divide in a formalized manner all actions to be performed to setup a Geant4 multi-threaded program into

- **common** actions to be performed and shared by all threads. This includes the setup of the geometry and the physics list. The other main area are
- **thread-specific** actions to be performed for each thread. These are composed by the user actions called during the processing of each run. These are the run-, event-, generation-, tracking-, stepping and stacking actions.

To understand the interplay between DDG4 and Geant4 let us quickly recapitulate the Geant4 mechanism how to configure multiple threads. The setup of a multi-threaded application in Geant4 is centered around two additional classes, which both deal with single- and multi-threaded issues:

- `G4VUserActionInitialization` class with 2 major callbacks: `Build()` which is executed for each **worker thread** and `BuildForMaster()` which is executed for master thread only.
- `G4VUserDetectorConstruction` class with the callbacks `Construct()`, where the shared geometry is constructed and `ConstructSDandField()` where the sensitive detectors and the electro magnetic fields are provided.

Both these Geant4 provided hooks are modeled in the standard DDG4 way as action queues, which allow a modular and fine grained setup as shown in Figure 12 and Figure 13.
The DDG4 framework ensures that all user callbacks are installed properly to the Geant4 run manager, which calls them appropriately at the correct time.
DDG4 provides three callbacks for each sequence. Each callback receives a temporary context argument, which may be used to shortcut access to basic useful quantities:

```
1    struct Geant4DetectorConstructionContext  {
2      /// Reference to geometry object
3      Geometry::LCDD&     lcdd;
4      /// Reference to the world after construction
5      G4VPhysicalVolume*  world;
```

[4] Please note that the whole of Geant4 and your client code must be compiled with the compile flag $-$DGEANT4$_{\text{B}}$UILD$_{\text{M}}$ULTITHREADED = ON

```
 6       /// The cached geometry information
 7       Geant4GeometryInfo* geometry;
 8       /// G4 User detector initializer
 9       G4VUserDetectorConstruction* detector;
10 };
```

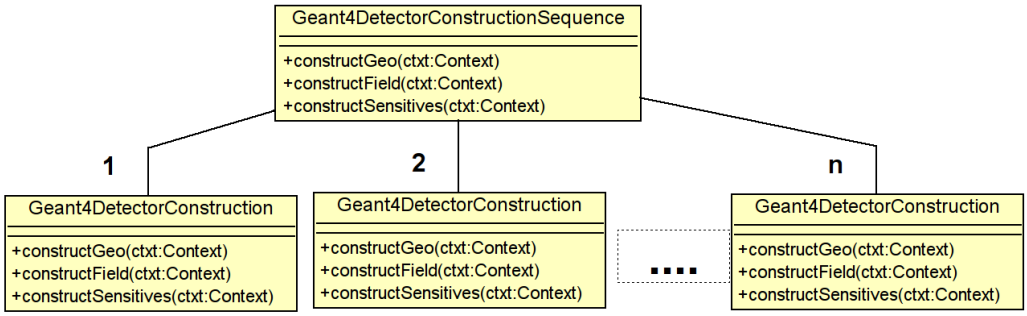

Figure 13: The Geant4 detector initialization sequence to setup DDG4. If supplied, Geant4 calls the components both, in the single-threaded and in the multi-threaded mode.

The callbacks and the expected functionality are:

1. First the detector geometry is constructed. This happens in the callback `constructGeo(...)`. If a standard `DD4hep` geometry is present, this is translation of the geometry could be done by simply calling the plugin `Geant4DetectorGeometryConstruction`. Alternatively a user defined plugin could perform this action.

2. Next the electromagnetic fields for the Geant4 particle tracking is constructed. A generic plugin `Geant4FieldTrackingConstruction` may be attached. The corresponding setup parameters are listed in Section 8. Alternatively a user defined plugin could perform this action.

3. Finally the Geant4 sensitive detectors are instantiated and attached to the sensitive volumes. For generic setups the plugin `Geant4DetectorSensitivesConstruction` may be attached. Alternatively a user defined plugin could perform this action.

7.2 Thread related contexts

`DDG4` provides thread related context, which may be accessed or modified by user code. This context, the `Geant4Context` and it's sub-components, as discussed in Section 5 are available as separate instances for each event and as such also independently for each worker thread. Hence, no user level locking of the event context is necessary in any worker thread.

7.3 Thread-Shared Components

Some actions, though executed in the context of a single thread context may only execute as singletons. An example would be a `GeneratorAction`, which read input events from file. Clearly the reading of data from file must be protected and the reading of one event in a given thread must finish, before the next thread may take over. Another example are data analysis components, which e.g. fill a histogram. Typically the filling mechanism of a histogram is not thread safe and hence must be protected.
To solve such issues all actions, which may involve such shared activities, a shared action is provided, which adopts a singleton instance and executes the relevant callbacks in a protected manner. The shared actions execute the user component in a thread safe envelope.

Clearly no run- or event related state in such shared actions may be carried by the component object across callbacks. The action objects may not be aware of the event related context outside the callback. Default implementations for such shared actions exist for

- the `Geant4RunAction`, where the calls to `Geant4RunAction::begin` and `Geant4RunAction::end` are **globally** locked and the sequential execution of the entire sequence is ensured;
- the `Geant4EventAction`,
- the `Geant4TrackingAction`,
- the `Geant4SteppingAction` and
- the `Geant4StackingAction`.

In the latter cases the framework ensures thread safety, but does not ensure the reentrant execution order of the entire sequence.
General Remark: Simple callbacks registered to the run-, event, etc.-actions cannot be protected. These callbacks may under no circumstances use any event related state information of the called object.

7.4 Backwards- and Single-Thread-Compatibility

As in the single threaded mode of Geant4, also in the multi-threaded mode all user actions are called by an instance of the `G4RunManager` or a sublass thereof, the `G4MTRunManager` [13].
If the recommended actions in sub-section 7.1 are used to configure the Geant4 application, then in a rather transparent way both single-threaded and multi-threaded setups can coexist simply by changing the concrete instance of the `G4RunManager`. There is one single exception: The user initialization function `G4VUserActionInitialization::BuildForMaster()` is **only** executed in multi-threaded mode. For this reason, we deprecate the usage. Try to find solutions, without master specific setup using e.g. shared actions.

7.5 Support for Python Setup in Multi-Threading Mode

The setup of `DDG4` in multi-threaded mode requires separate callbacks for the global configuration (geometry, etc.) and the configuration of the worker threads. In python this setup is performed within python callable objects, which are either functions or member functions of objects. These functions may have arguments. The python specific configuration actions

- The user initialization action `Geant4PythonInitialization` allows to configure python callbacks for the master and the worker thread setup using the calls:

```
1          /// Set the Detector initialization command
2          void setMasterSetup(PyObject* callable, PyObject* args);
3          /// Set the field initialization command
4          void setWorkerSetup(PyObject* callable, PyObject* args);
```

  to be used in python as a call sequence within the master thread:

```
1          init_seq = kernel.userInitialization(True)
2          init_action = UserInitialization(kernel,'Geant4PythonInitialization/PyG4Init')
3          init_action.setWorkerSetup(worker_setup_call, < worker_args > )
4          init_action.setMasterSetup(master_setup_call, < master_args > )
5          init_seq.adopt(init_action)
```

  The callback argument list $< worker\_args >$ and $< master\_args >$ are python tuples containing all arguments expected by the callable objects *worker_setup_call* and *master_setup_call* respecyively. The class `Geant4PythonInitialization` is a subclass of `Geant4UserInitialization` and will call the provided functions according to the protocol explained earlier in this section. If a callback is not set, the corresponding actiion is not executed.

- The detector construction action `Geant4PythonDetectorConstruction` is the corresponding python action to populate the detector construction sequencer. and supports three ccallbacks:

```
1           /// Set the Detector initialization command
2           void setConstructGeo(PyObject* callable, PyObject* args);
3           /// Set the field initialization command
4           void setConstructField(PyObject* callable, PyObject* args);
5           /// Set the sensitive detector initialization command
6           void setConstructSensitives(PyObject* callable, PyObject* args);
```

to be used in python as call sequence within the master thread:

```
1         init_seq = self.master().detectorConstruction(True)
2         init_action = DetectorConstruction(self.master(),name_type)
3         init_action.setConstructGeo(geometry_setup_call, < geometry_args > )
4         init_action.setConstructField(field_setup_call, < field_args > )
5         init_action.setConstructSensitives(sensitives_setup_call, < sensitives_args >)
6         init_seq.adopt(init_action)
```

If any of the three callback is not set, the corresponding actiion is not executed. Hereby are *geometry_setup_call*, *field_setup_call* and *sensitives_setup_call* the callable objects to configure the geometry, the tracking field and the sensitive detectors. $< geometry\_args >$, $< field\_args >$ and $< sensitives\_args >$ are the corresponding callable arguments in the form of a python tuple object.

All python callbacks are supposed to return the integer '1' on success. Any other return code is assumed to be failure.

7.6 DDG4 Multi-Threading Example

```
1 """
2
3   DD4hep simulation example setup DDG4
4   in multi-threaded mode using the python configuration
5
6   @author  M.Frank
7   @version 1.0
8
9 """
10 import os, time, DDG4
11
12 def setupWorker(geant4):
13   kernel = geant4.kernel()
14   print '#PYTHON: +++ Creating Geant4 worker thread ....'
15   print "#PYTHON:  Configure Run actions"
16   run1 = DDG4.RunAction(kernel,'Geant4TestRunAction/RunInit')
17     ...
18   print "#PYTHON:  Configure Event actions"
19   prt = DDG4.EventAction(kernel,'Geant4ParticlePrint/ParticlePrint')
20   kernel.eventAction().adopt(prt)
21     ...
22   print "\n#PYTHON:  Configure I/O\n"
23   evt_root = geant4.setupROOTOutput('RootOutput','CLICSiD_'+time.strftime('%Y-%m-%d_%H-%M'))
24     ...
25   gen = DDG4.GeneratorAction(kernel,"Geant4GeneratorActionInit/GenerationInit")
26   kernel.generatorAction().adopt(gen)
27   print "#PYTHON:  First particle generator: pi+"
28   gen = DDG4.GeneratorAction(kernel,"Geant4IsotropeGenerator/IsotropPi+");
```

```
29     ...
30   print "#PYTHON:  Merge all existing interaction records"
31   gen = DDG4.GeneratorAction(kernel,"Geant4InteractionMerger/InteractionMerger")
32   kernel.generatorAction().adopt(gen)
33   print "#PYTHON:  Finally generate Geant4 primaries"
34   gen = DDG4.GeneratorAction(kernel,"Geant4PrimaryHandler/PrimaryHandler")
35   kernel.generatorAction().adopt(gen)
36   print "#PYTHON:  ....and handle the simulation particles."
37   part = DDG4.GeneratorAction(kernel,"Geant4ParticleHandler/ParticleHandler")
38   kernel.generatorAction().adopt(part)
39
40   user = DDG4.Action(kernel,"Geant4TCUserParticleHandler/UserParticleHandler")
41     ...
42   part.adopt(user)
43   print '#PYTHON: +++ Geant4 worker thread configured successfully....'
44   return 1
45
46 def setupMaster(geant4):
47   kernel = geant4.master()
48   print '#PYTHON: +++ Setting up master thread for ',kernel.NumberOfThreads,' workers.'
49   return 1
50
51 def setupSensitives(geant4):
52   print "#PYTHON:  Setting up all sensitive detectors"
53   geant4.printDetectors()
54   print "#PYTHON:  First the tracking detectors"
55   seq,act = geant4.setupTracker('SiVertexBarrel')
56     ...
57   print "#PYTHON:  Now setup the calorimeters"
58   seq,act = geant4.setupCalorimeter('EcalBarrel')
59     ...
60   return 1
61
62 def run():
63   kernel = DDG4.Kernel()
64   lcdd = kernel.lcdd()
65   install_dir = os.environ['DD4hepINSTALL']
66   DDG4.Core.setPrintFormat("%-32s %6s %s")
67   kernel.loadGeometry("file:"+install_dir+"/DDDetectors/compact/SiD.xml")
68   DDG4.importConstants(lcdd)
69
70   kernel.NumberOfThreads = 3
71   geant4 = DDG4.Geant4(kernel,tracker='Geant4TrackerCombineAction')
72   print "#  Configure UI"
73   geant4.setupCshUI()
74
75   print "#  Geant4 user initialization action"
76   geant4.addUserInitialization(worker=setupWorker, worker_args=(geant4,),
77                                master=setupMaster,master_args=(geant4,))
78   print "#  Configure G4 geometry setup"
79   seq,act = geant4.addDetectorConstruction("Geant4DetectorGeometryConstruction/ConstructGeo")
80
81   print "# Configure G4 sensitive detectors: python setup callback"
82   seq,act = geant4.addDetectorConstruction("Geant4PythonDetectorConstruction/SetupSD",
83                                            sensitives=setupSensitives,sensitives_args=(geant4,))
84   print "# Configure G4 sensitive detectors: atach'em to the sensitive volumes"
85   seq,act = geant4.addDetectorConstruction("Geant4DetectorSensitivesConstruction/ConstructSD")
86
```

```
 87   print "#  Configure G4 magnetic field tracking"
 88   seq,field = geant4.addDetectorConstruction("Geant4FieldTrackingConstruction/MagFieldTrackingSetup")
 89   field.stepper            = "HelixGeant4Runge"
 90   field.equation           = "Mag_UsualEqRhs"
 91   field.eps_min            = 5e-05 * mm
 92   ...
 93   print "#  Setup random generator"
 94   rndm = DDG4.Action(kernel,'Geant4Random/Random')
 95   rndm.Seed = 987654321
 96   rndm.initialize()
 97   print "#  Now build the physics list:"
 98   phys = geant4.setupPhysics('QGSP_BERT')
 99   geant4.run()
100
101 if __name__ == "__main__":
102   run()
```

8 Existing DDG4 components

In the introduction the longterm goal was expressed, that with DDG4 users should be able to pick components from a growing palette and connect the selected components using the setup mechanisms described in Section 4.
Such a palette based approach obviously depends on the availability of documentation for existing components describing the properties of each component and the interaction of each component within the `DDG4` framework.
All components defer from the basic type `Geant4Action`. This means **all** components have the **default** properties described in the table below:

Component Properties:		`default`
OuputLevel	[int]	Output level of the component to customize printouts
Name	[string]	Component name [read-only]
Control	[boolean]	Steering of the Geant4 Messenger creation

Important notice for developers:
Since the documentation of developed components is VERY important, please never forget to supply the corresponding documentation.

At least supply the minimal documentation ash shown below in the appended examples for the "Simple" detector response and I/O components.

8.1 Generic Action Modules

8.1.1 Geant4UIManager

The `Geant4UIManager` handles interactivity aspects between Geant4, its command handlers (i.e. terminal) and the various components the actions interact.
The `Geant4UIManager` is a component attached to the `Geant4Kernel` object. All properties of all `Geant4Action` instances may be exported to `Geant4` messengers and *may* hence be accessible directly from the `Geant4` prompt. To export properties from any action, call the `enableUI()` method of the action. The callback signature is: `void operator()(G4Event* event)`.

Class name	`Geant4UIManager`
File name	`DDG4/src/Geant4UIManager.cpp`
Type	`Geant4Action`
Component Properties:	defaults apply
SessionType (string)	Session type (csh, tcsh, etc.
SetupUI (string)	Name of the UI macro file
SetupVIS (string)	Name of the visualization macro file
HaveVIS (bool)	Flag to instantiate Vis manager (def:false, unless VisSetup set)
HaveUI (bool)	Flag to instantiate UI (default=true)

8.1.2 Geant4Random

Mini interface to the random generator of the application. Necessary, that on every object creates its own instance, but accesses the main instance available through the `Geant4Context`.
This is mandatory to ensure reproducibility of the event generation process. Particular objects may use a dependent generator from an experiment framework like `GAUDI`.
internally the engine factory mechanism of `CLHEP` is used. Please refer there within for valid engine names and the random seeding mechanism, which may vary between different engines.
Any number of independent random objects may be created and used in parallel. This however, is not advised to ensure reproducibility.
The first instance of the random action is automatically set to be the `Geant4` instance. If another instance should be used by `Geant4`, use `setMainInstance(Geant4Random* ptr)` class method to override this behavior. Provision, steered by options, is taken to ensure the `gRandom` of `ROOT` uses the same random number engine.

Class name	`Geant4Random`
File name	`DDG4/src/Geant4Random.cpp`
Type	`Geant4Random`
Component Properties:	defaults apply
File (string)	File name if initialized from file. If set, engine name and seeds are ignored
Engine (string)	Engine type name. All engines defined in the `CLHEP::EngineFactory` class are available. If no type is supplied the engine from the HepRandom generator instance is taken.
Seed (long)	Initial random seed. Default: 123456789. If not ZERO terminated, termination is added.
Replace_gRandom (bool)	Flag to replace the `ROOT gRandom` instance with this random number engine. This ensures `ROOT` and `Geant4` use the same random number engine, hence the same random sequence.

8.2 Geant4UserInitialization Implementations

8.2.1 Geant4PythonInitialization

Please see Section 7.5 for an illustration of the usage. The configuration by construction must be performed using setter-functions rather than properties.

Class name	`Geant4PythonInitialization`
File name	`DDG4/src/python/Geant4PythonInitialization.cpp`
Type	`Geant4Action`
Component Properties:	defaults apply

8.2.2 Geant4PythonDetectorConstruction

Please see Section 7.5 for an illustration of the usage. The configuration by construction must be performed using setter-functions rather than properties.

Class name	`Geant4PythonDetectorConstruction`
File name	`DDG4/src/python/Geant4PythonDetectorConstruction.cpp`
Type	`Geant4Action`
Component Properties:	defaults apply

8.3 Predefined Geant4 Physics List Objects

The physics list may be defined entirely data driven using the factory mechanism using a variety of predefined objects. Though users are free to define private physics lists, typically the predefined physics lists from `Geant4` are used.
The inventory changes over time, new lists appear and obsolete lists are purged, hence we will not list them explicitly here. For the inventory of available physics lists, please refer to the implementation files:

- Inventory of predefined physics lists, which may be inherited:
  DDG4/plugins/Geant4PhysiscsLists.cpp
- Inventory of predefined physics constructors, which may be instantiated:
  DDG4/plugins/Geant4PhysicsConstructors.cpp
- Inventory of predefined process constructors, which may be instantiated:
  DDG4/plugins/Geant4Processes.cpp
- Inventory of predefined particle constructors, which may be instantiated:
  DDG4/plugins/Geant4Particles.cpp

8.4 Geant4 Generation Action Modules

Here we discuss modules, which are intrinsically part of DDG4 and may be attached to the `Geant4GeneratorActionSequ`

8.4.1 Base class: Geant4GeneratorAction

The `Geant4GeneratorAction` is called for every event. During the callback all particles are created which form the microscopic kinematic action of the particle collision. This input may either origin directly from an event generator program or come from file.
The callback signature is: void operator()(G4Event* event) See also: `Geant4EventAction` in the doxygen documentation.

Class name	`Geant4GeneratorAction`
File name	`DDG4/src/Geant4GeneratorAction.cpp`
Type	`Geant4Action, Geant4GeneratorAction`
Component Properties:	defaults apply

8.4.2 Geant4GeneratorActionSequence

The sequence dispatches the callbacks at the beginning of an event to all registered `Geant4GeneratorAction` members and all registered callbacks.
See also: The `Geant4GeneratorActionSequence` is directly steered by the single instance of the `G4VUserPrimaryGeneratorAction`, the Geant4 provided user hook, which is private.
See also: `Geant4UserGeneratorAction` and `Geant4GeneratorActionSequence` in the doxygen documentation.

Class name	`Geant4Geant4GeneratorActionSequence`
File name	`DDG4/src/Geant4GeneratorAction.cpp`
Type	`Geant4Action`
Component Properties:	defaults apply

8.4.3 Geant4GeneratorActionInit

Initialize the Geant4Event objects to host generator and MC truth related information Geant4 actions to collect the MC particle information. This action should register all event extension required for the further processing. We want to avoid that every client has to check if a given object is present or not and than later install the required data structures.
These by default are extensions of type:

- `Geant4PrimaryEvent` with multiple interaction sections, one for each interaction This is the MAIN and ONLY information to feed Geant4
- `Geant4PrimaryInteraction` containing the track/vertex information to create the primary particles for Geant4. This record is build from the `Geant4PrimaryEvent` information.
- `Geant4PrimaryMap` a map of the `Geant4Particles` converted to `G4PrimaryParticles` to ease particle handling later.
- `Geant4ParticleMap` the map of particles created during the event simulation. This map has directly the correct particle offsets, so that the merging of `Geant4PrimaryInteraction` particles and the simulation particles is easy....

Class name	`Geant4Geant4GeneratorActionInit`
File name	`DDG4/src/Geant4GeneratorActionInit.cpp`
Type	`Geant4GeneratorAction`
Component Properties:	defaults apply
Angle (double)	`Lorentz-Angle of boost`
Mask (int.bitmask)	`Interaction identifier`

8.4.4 Geant4InteractionVertexBoost

Boost the primary vertex and all particles outgoing the primary interaction in X-direction.
The interaction to be processed by the component is uniquely identified by the **Mask** property. Two primary interaction may not have the same mask.
Note [special use case]:
If all contributing interactions of the one event **registered in the primary event at the time the action is called** should be handled by one single component instance, set the **Mask** property to **-1**.

Class name	`Geant4InteractionVertexBoost`
File name	`DDG4/src/Geant4InteractionVertexBoost.cpp`
Type	`Geant4GeneratorAction`
Component Properties:	defaults apply
Angle (double)	`Lorentz-Angle of boost`
Mask (int.bitmask)	`Interaction identifier`

8.4.5 Geant4InteractionVertexSmear

Smear the primary vertex and all particles outgoing the primary interaction.
The interaction to be processed by the component is uniquely identified by the **Mask** property. Two primary interaction may not have the same mask.
Note [special use case]:
If all contributing interactions of the one event **registered in the primary event at the time the action is called** should be handled by one single component instance, set the **Mask** property to **-1**.

Class name	`Geant4InteractionVertexSmear`
File name	`DDG4/src/Geant4InteractionVertexSmear.cpp`
Component Properties:	defaults apply
Offset (PxPyPzEVector)	`Smearing offset`
Sigma (PxPyPzEVector)	`Sigma (Errors) on offset`
Mask (int.bitmask)	`Interaction identifier`

8.4.6 Geant4InteractionMerger

Merge all interactions created by each `Geant4InputAction` into one single record. The input records are taken from the item `Geant4PrimaryEvent` and are merged into the `Geant4PrimaryInteraction` object attached to the `Geant4Event` event context.

Class name	`Geant4InteractionMerger`
File name	`DDG4/src/Geant4InteractionMerger.cpp`
Type	`Geant4GeneratorAction`
Component Properties:	defaults apply

8.4.7 Geant4PrimaryHandler

Convert the primary interaction (object `Geant4PrimaryInteraction` object attached to the `Geant4Event` event context) and pass the result to Geant4 for simulation.

Class name	`Geant4PrimaryHandler`
File name	`DDG4/src/Geant4PrimaryHandler.cpp`
Type	`Geant4GeneratorAction`
Component Properties:	defaults apply

8.4.8 Geant4ParticleGun

Implementation of a particle gun using Geant4Particles.
The `Geant4ParticleGun` is a tool to shoot a number of particles with identical properties into a given region of the detector to be simulated.
The particle gun is a input source like any other and participates in the general input stage merging process like any other input e.g. from file. Hence, there may be several particle guns present each generating it's own primary vertex. Use the mask property to ensure each gun generates it's own, well identified primary vertex.
There is one 'user lazyness' support though: If there is only one particle gun in use, the property 'Standalone', which by default is set to true invokes the interaction merging and the Geant4 primary generation directly.
The interaction to be created by the component is uniquely identified by the **Mask** property. Two primary interaction may not have the same mask.

Class name	`Geant4PrimaryHandler`
File name	`DDG4/src/Geant4PrimaryHandler.cpp`
Type	`Geant4GeneratorAction`
Component Properties:	default
particle (string)	Particle type to be shot
energy (double)	Particle energy in MeV
position (XYZVector)	Pole position of the generated particles in mm
direction (XYZVector)	Momentum direction of the generated particles
isotrop (bool)	Isotropic particle directions in space.
Mask (int.bitmask)	Interaction identifier
Standalone (bool)	Setup for standalone execution including interaction merging etc.

8.4.9 Geant4ParticleHandler

Extract the relevant particle information during the simulation step.
This procedure works as follows:

- At the beginning of the event generation the object registers itself as Monte-Carlo truth handler to the event context.
- At the begin of each track action a particle candidate is created and filled with all properties known at this time.
- At each stepping action a flag is set if the step produced secondaries.
- Sensitive detectors call the MC truth handler if a hit was created. This fact is remembered.
- At the end of the tracking action a first decision is taken if the candidate is to be kept for the final record.
- At the end of the event action finally all particles are reduced to the final record. This logic can be overridden by a user handler to be attached.

Any of these actions may be intercepted by a `Geant4UserParticleHandler` attached to the particle handler. See class `Geant4UserParticleHandler` for details.

Class name	`Geant4ParticleHandler`
File name	`DDG4/src/Geant4ParticleHandler.cpp`
Type	`Geant4GeneratorAction`
Component Properties:	defaults apply
KeepAllParticles (bool)	Flag to keep entire particle record without any reduction. This may result in a huge output record.
SaveProcesses (vector(string))	Array of Geant4 process names, which products and parent should NOT be reduced.
MinimalKineticEnergy (double)	Minimal energy below which particles should be ignored unless other criteria (Process, created hits, etc) apply.

8.5 Geant4 Event Action Modules

8.5.1 Base class: Geant4EventAction

The EventAction is called for every event.
This class is the base class for all user actions, which have to hook into the begin- and end-of-event actions. Typical use cases are the collection/computation of event related properties.
Examples of this functionality may include for example:

- Reset variables summing event related information in the begin-event callback.
- Monitoring activities such as filling histograms from hits collected during the end-event action.

See also: `Geant4EventAction` in the doxygen documentation.

Class name	`Geant4EventAction`
File name	`DDG4/src/Geant4EventAction.cpp`
Type	`Geant4EventAction`
Component Properties:	defaults apply

8.5.2 Geant4EventActionSequence

The `Geant4EventActionSequence` is directly steered by the single instance of the `G4UserEventAction`, the Geant4 provided user hook, which is private.
See also: `Geant4UserEventAction` in the doxygen documentation.

Class name	`Geant4EventAction`
File name	`DDG4/src/Geant4EventAction.cpp`
Type	`Geant4EventAction`
Component Properties:	defaults apply

8.5.3 Geant4ParticlePrint

Geant4Action to print MC particle information.

Class name	`Geant4ParticlePrint`
File name	`DDG4/src/Geant4ParticlePrint.cpp`
Type	`Geant4EventAction`
Component Properties:	defaults apply
OutputType (bool)	Flag to steer output type. 1: Print table of particles. 2: Print table of particles. 3: Print table and tree of particles.
PrintHits	Print associated hits to every particle (big output!)

8.6 Sensitive Detectors

8.6.1 Geant4TrackerAction

Simple sensitive detector for tracking detectors. These trackers create one single hit collection. The created hits may be written out with the output modules described in Section 8.7.1 and 8.7.2.
The basic specifications are:

Basics:	
Class name	`Geant4SensitiveAction<Geant4Tracker>`
File name	`DDG4/plugins/Geant4SDActions.cpp`
Hit collection	`Name of the readout object`
Hit class	`Geant4Tracker::Hit`
File name	`DDG4/include/Geant4Data.h`
Component Properties:	defaults apply

8.6.2 Geant4CalorimeterAction

Simple sensitive detector for calorimeters. The sensitive detector creates one single hit collection. The created hits may be written out with the output modules described in Section 8.7.1 and 8.7.2.
The basic specifications are:

Basics:	
Class name	`Geant4SensitiveAction<Geant4Calorimeter>`
File name	`DDG4/plugins/Geant4SDActions.cpp`
Hit collection	`Name of the readout object`
Hit class	`Geant4Calorimeter::Hit`
File name	`DDG4/include/Geant4Data.h`
Component Properties:	defaults apply

8.7 I/O Components

8.7.1 ROOT Output "Simple"

8.7.2 LCIO Output "Simple"

References

[1] DD4Hep web page, http://aidasoft.web.cern.ch/DD4hep.

[2] LHCb Collaboration, "LHCb, the Large Hadron Collider beauty experiment, reoptimised detector design and performance", CERN/LHCC 2003-030

[3] S. Ponce et al., "Detector Description Framework in LHCb", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2003), La Jolla, CA, 2003, proceedings.

[4] The ILD Concept Group, "The International Large Detector: Letter of Intent", ISBN 978-3-935702-42-3, 2009.

[5] H. Aihara, P. Burrows, M. Oreglia (Editors), "SiD Letter of Intent", arXiv:0911.0006, 2009.

[6] R.Brun, A.Gheata, M.Gheata, "The ROOT geometry package", Nuclear Instruments and Methods **A** 502 (2003) 676-680.

[7] R.Brun et al., "Root - An object oriented data analysis framework", Nuclear Instruments and Methods **A** 389 (1997) 81-86.

[8] S. Agostinelli et al., "Geant4 - A Simulation Toolkit", Nuclear Instruments and Methods **A** 506 (2003) 250-303.

[9] T.Johnson et al., "LCGO - geometry description for ILC detectors", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2007), Victoria, BC, Canada, 2012, Proceedings.

[10] N.Graf et al., "lcsim: An integrated detector simulation, reconstruction and analysis environment", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2012), New York, 2012, Proceedings.

[11] R. Chytracek et al., "Geometry Description Markup Language for Physics Simulation and Analysis Applications", IEEE Trans. Nucl. Sci., Vol. 53, Issue: 5, Part 2, 2892-2896, http://gdml.web.cern.ch.

[12] C.Grefe et al., "The DDSegmentation package", Non existing documentation to be written.

[13] Geant4 Multi threading Guides. Please see for details: https://twiki.cern.ch/twiki/bin/view/Geant4/Geant4MTAdvandedTopicsForApplicationDevelopers, https://twiki.cern.ch/twiki/bin/view/Geant4/QuickMigrationGuideForGeant4V10, http://geant4.slac.stanford.edu/tutorial/MC2015G4WS/Multithreading.pdf