GIGA: $\mathcal{GAUDI}$ Interface for *Geant4* Applications or *Geant4* Interface for $\mathcal{GAUDI}$ Applications[*]

I.Belyaev[†]
ITEP (Moscow)

June 25, 2000

Abstract

Proposal for accessing of *Geant4* facilities from $\mathcal{GAUDI}$ framework is discussed. This proposal allows the smooth step-by-step transition from stand-alone *Geant4* applications into applications fully embedded into $\mathcal{GAUDI}$ architecture to be performed. The proposed service for communications of algorithms within $\mathcal{GAUDI}$ framework with *Geant4* structures allows the usage of *Geant4 Tool Kit* as a black-box without detailed knowledge of its internal features.

Contents

[*] The document could be found in `$GIGAROOT/doc/GiGa.ps`
[†] `E-mail: Ivan.Belyaev@itep.ru`

1 *Simulation* as a ”black-box”

Simulation is an essential part of the general software in modern High Energy Physics. It is foreseen to use *Geant4 Tool Kit* as the major simulation package for the LHC era.

1.1 Natural decomposition of *Simulation Environment*

It is worth to consider any *simulation program* as sequence of steps, each of them could be represented as the black-box with some well defined input and output data flow.

For almost each *simulation program* it is natural to identify the following steps:

- *Initialisation:*
  At this step *simulation program* requires to be provided with some *Input Data*: physical properties of all participating components, like particles properties, properties of physical processes, description of geometry and materials.
- *Event Loop*
  - *Event Initialisation*
    At this step *simulation program* requires to be provided with some *input sata* - initial kinematics
  - *Event Processing*
    At this step simulation program usually neither require to be provided with some *input data* nor produce some *output data*, but some *user action* methods are to be supplied to the *simulation program*, e.g. notorious `gustep.F` routine in `GEANT3` package.
  - *Event Finalisation*
    At this step *simulation program* is ready to provide the *user program* with the simulation *output data* - hits, digits and output kinematics - secondary particles.
- *Finalisation*

1.2 Communication categories

From above sketched rough scheme one could deduce that all *communications* of *user program* with *simulation environment* could be naturally subdivided into 2 categories:

- *Method-like Category:*
  Communications from this category are characterised by dealing mainly

with internal structures of *simulation program*. They do not produce or consume the *data* from the outside of the *simulation environment.* Usually they overwrite some internal default dummy methods from *simulation environment.* `gustep.F` could be considered as a typical representative of such category.

- *Stream-like Category:*
  Communications from this category are characterised by either producing the *data*, which to be used outside of the *simulation environment*, like hits, digits, secondary particles, histograms and n-tuples, or they load data flow from the outside of the simulation program, like particle properties, properties of physical processes and detector description. Usually communications from this category could be easily identified as *input streams* or *output streams.* Routine `guout.F` could be considered as a typical representative of *output stream* category and routine `gukine.F` could be considered as a typical representative of *input stream* category.

Also *communications* could be classified into 2 classes:

- *Configuration Communications:*
  Such type of communication is used for configuration of the *simulation environment* and supplying it with *input data* which are constant for a some period, sometimes for the whole job lifetime.
- *Event-by-event Communications:*
  Such type of communication is used on *event-by-event basis*

1.3 *Communications* with *Geant4 Tool Kit*

A schematic view of *communications* with *Geant4 Tool Kit* is presented on figure 1.3. Arrows represent the direction of data flows.

2 GiGa **Service as** *Simulation Service*

2.1 Why *Service* ?

Taking into account the rough schemes from section 1, and definitions of $\mathcal{GAUDI}$ categories, one could easily deduce that the most suitable form for embedding the *simulation environment* into $\mathcal{GAUDI}$ framework is *Service.*

Several simple and independent $\mathcal{GAUDI}$ *Algorithm*s and *Converters* could communicate with with *Simulation Service* suppling it with *input data*: detector description, particle properties, description of physical processes, cut-offs and initial kinematics and retrieving from it the *output data* in the format of hits and secondaries.

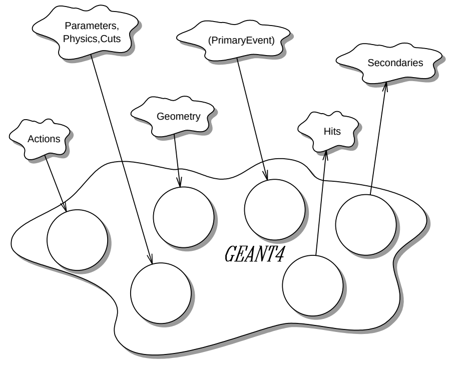

Figure 1: A schematic view of *communications* with *Geant4 Tool Kit*. Data flow directions are indicated by arrows.

The service implements two *abstract interface*s:

- *IGiGaSvc interface:*
  for *event-by-event communication*s
- *IGiGaSetUpSvc interface:*
  for *configuration communication*s

2.2 *IGiGaSvc Interface*

All *event-by-event stream-like communications* with *simulation environment* defined in *IGiGaSvc* abstract interface. This interface is designed to manipulate with *input event data* and *output event data*:

- *Input Event Data* are accepted in the form of
  - `G4PrimaryVertex*`
    representing the *Geant4 primary event record*
  - `MCVertex*`
    representing the whole event record, starting with *primary vertex*
  - `MCParticle*`
    representing the whole event record, starting with *primary particle*
- *Output Event Data*, retrieved from the *Service* could be of the format:
  - `G4Event*`
    provide the access to the whole internal processed event
  - `G4HCofThisEvent*`
    provide the access to the all hit collections of the processed event
  - `CollectionPair*`
    provide the access to the specific hit collection

- `G4TrajectoryContainer*`
  provide the access kinematic of secondaries

Since all communications with *simulation environment* via *IGiGaSvc* interface are *stream-like communications*, the overloaded stream operators are considered as main methods in the interface definition:

```
/////////////////////////////////////////////////////////////////////////////////
///                                                                           ///
virtual IGiGaSvc&   operator <<         ( G4PrimaryVertex  * vertex   ) = 0 ; ///
virtual IGiGaSvc&   operator <<         ( const MCVertex   * vertex   ) = 0 ; ///
virtual IGiGaSvc&   operator <<         ( const MCParticle * particle ) = 0 ; ///
///                                                                           ///
/////////////////////////////////////////////////////////////////////////////////
///                                                                           ///
virtual IGiGaSvc& operator >> ( const G4Event*         & event        ) = 0 ; ///
virtual IGiGaSvc& operator >> ( G4HCofThisEvent*       & collections  ) = 0 ; ///
virtual IGiGaSvc& operator >> ( CollectionPair         & collection   ) = 0 ; ///
virtual IGiGaSvc& operator >> ( G4TrajectoryContainer* & trajectories ) = 0 ; ///
///                                                                           ///
/////////////////////////////////////////////////////////////////////////////////
```

In addition to these straightforward definitions, ordinary old-fashioned function-like methods are defined:

```
/////////////////////////////////////////////////////////////////////////////////
///                                                                           ///
virtual StatusCode  addPrimaryKinematics( G4PrimaryVertex  * vertex   ) = 0 ; ///
virtual StatusCode  addPrimaryKinematics( const MCVertex   * vertex   ) = 0 ; ///
virtual StatusCode  addPrimaryKinematics( const MCParticle * particle ) = 0 ; ///
///                                                                           ///
/////////////////////////////////////////////////////////////////////////////////
///                                                                           ///
virtual StatusCode retrieveEvent          ( const G4Event*          & ) = 0 ; ///
virtual StatusCode retrieveHitCollections ( G4HCofThisEvent*        & ) = 0 ; ///
virtual StatusCode retrieveHitCollection  ( CollectionPair          & ) = 0 ; ///
virtual StatusCode retrieveTrajectories   ( G4TrajectoryContainer*  & ) = 0 ; ///
///                                                                           ///
/////////////////////////////////////////////////////////////////////////////////
```

The functionality of these methods almost the same. The difference belongs to the error handling. In the case of errors the operator-like calls throw `GiGaException`, while function-like calls return the `StatusCode::FAILURE`.

2.3 *IGiGaSetUpSvc Interface*

All *configuration communications* with *Simulation Service* are defined in *IGiGaSetUpSvc interface.*

The interface consists of the following definitions of operator-like calls:

```
/////////////////////////////////////////////////////////////////////////////////
///                                                                           ///
virtual IGiGaSetUpSvc& operator << ( G4VUserDetectorConstruction   * ) = 0 ;  ///
virtual IGiGaSetUpSvc& operator << ( G4VPhysicalVolume             * ) = 0 ;  ///
virtual IGiGaSetUpSvc& operator << ( G4VUserPrimaryGeneratorAction * ) = 0 ;  ///
virtual IGiGaSetUpSvc& operator << ( G4VUserPhysicsList            * ) = 0 ;  ///
virtual IGiGaSetUpSvc& operator << ( G4UserRunAction               * ) = 0 ;  ///
virtual IGiGaSetUpSvc& operator << ( G4UserEventAction             * ) = 0 ;  ///
virtual IGiGaSetUpSvc& operator << ( G4UserStackingAction          * ) = 0 ;  ///
virtual IGiGaSetUpSvc& operator << ( G4UserTrackingAction          * ) = 0 ;  ///
virtual IGiGaSetUpSvc& operator << ( G4UserSteppingAction          * ) = 0 ;  ///
virtual IGiGaSetUpSvc& operator << ( G4VisManager                  * ) = 0 ;  ///
///                                                                           ///
/////////////////////////////////////////////////////////////////////////////////
```

Here the advantage of operator-like methods is not so clear and obvious.

```
////////////////////////////////////////////////////////////////////////////////
///                                                                          ///
virtual StatusCode setConstruction ( G4VUserDetectorConstruction   * ) = 0 ;  ///
virtual StatusCode setDetector     ( G4VPhysicalVolume             * ) = 0 ;  ///
virtual StatusCode setGenerator    ( G4VUserPrimaryGeneratorAction * ) = 0 ;  ///
virtual StatusCode setPhysics      ( G4VUserPhysicsList            * ) = 0 ;  ///
virtual StatusCode setRunAction    ( G4UserRunAction               * ) = 0 ;  ///
virtual StatusCode setEvtAction    ( G4UserEventAction             * ) = 0 ;  ///
virtual StatusCode setStacking     ( G4UserStackingAction          * ) = 0 ;  ///
virtual StatusCode setTracking     ( G4UserTrackingAction          * ) = 0 ;  ///
virtual StatusCode setStepping     ( G4UserSteppingAction          * ) = 0 ;  ///
virtual StatusCode setVisManager   ( G4VisManager                  * ) = 0 ;  ///
///                                                                          ///
////////////////////////////////////////////////////////////////////////////////
```

The functionality of these methods almost the same. The difference belongs to the error handling. In the case of errors the operator-like calls throw `GiGaException`, while function-like calls return the `Status-Code::FAILURE`.

2.4 Initial Kinematics

Primary event record for *Geant4* is created by providing of GIGA Service via *IGiGaSvc* interface with pointers to `G4PrimaryVertex`,`MCVertex` and/or `MCParticle` objects. A very complex event could be prepared by subsequent calls. The call with `NULL` pointer triggers the event generation using the instance of `G4VUserPrimaryGeneratorAction` class, which in this case must be provided to GIGA Service via *IGiGaSetUpSvc interface.*

2.5 Detector Description

GIGA *Service* must be provided with detector description either in the form of pointer to the ”root” object of the type `G4VPhysicslVolume` or by declaration of the instance of the class `G4VUserDetectorConstuction`. Both declarations do through *IGiGaSetUp interface.*

3 Foreseen GIGA Evolution

A detailed review on possible schemes of GIGA evolution are presented in a separate document[1] Here we report only short extracted snapshots of the foreseen GIGA evolution of and related stuff. According to described below phase classification we are now just in the start of Phase II[2].

[1]The document could be found at `$GIGAROOT/doc/GiGaEvolution.ps`

[2]Several steps from Phase II are already performed, but unfortunately they are not tested well enough

3.1 Phase I

At this phase we foreseen the direct usage of GIGA Service and some limited number of *Geant4* classes by *user*'s algorithms.

This case corresponds to a native usage of ”stand-alone” *Geant4* application within *GAUDI* framework. *User*'s algorithms act like wrappers over main program of *Geant4*. Frankly speaking at very beginning the only one profit that ordinary user gets from usage GIGA Service with respect to an ordinary ”stand-alone” *Geant4* program, is the fast and easy access to general and technical *GAUDI* facilities like histogram, code profiling and others. On other side, if one has the working *Geant4* code, it could be embedded into *GAUDI* framework using GIGA facilities in a straightforward way without changing any line of codes[3].

3.2 Phase II

This phase could be considered as *”transition phase”* between Phase I and Phase III and it could be easily subdivided into some steps:

1. At this step we enhance the functionality of GIGA by making possible to extract the event record from *GAUDI event store*
2. At this step we enhance the functionality of GIGA by making possible to get the Detector Description by pointing into the root of already constructed *Geant4* tree
3. At this step we foreseen to implement the automatic translation of *GAUDI* Detector Description into *Geant4* detector description.
4. At this step we foreseen to implement the automatic creation of *Geant4 hits* and *sensitive volume* from their description via XML. A some common brainstorming within close collaboration with sub-detector groups is necessary for performing of this step.
5. At this step we foreseen to automatic translation of *Geant4 hits* into *GAUDI* Monte Carlo objects[4]
6. At this step we foreseen to automatic population of *GAUDI event store* by information from *Geant4 trajectories*[5]

After implementing first two steps GIGA Service could be considered as *”frozen”*, since all other steps to Phase III will not require any change in functionality of GIGA Service itself.

[3]Only the `main()` function is to be removed

[4]Obviously this step could not be done without close collaboration with sub-detector groups

[5]Communications physics and generator groups are required

3.3 Phase III

At this phase no any *user*'s algorithm deals directly with GiGa Service and *Geant4* classes. All knowledge of *Geant4* will be absorbed by set of specific *Converters*. This set of specific *Converters* corresponds to an additional layer in the data flow, making the user free from the knowledge of *Geant4* machinery.

Being at this phase we could start to think about configuration of *Geant4 physics list* and/or *cut-offs* using internal $\mathcal{GAUDI}$ features like *jobOptions Service* and/or *interactive scripting language*. Moreover at this stage we foreseen the embedding of the essential commands from *Geant4* interactive *user interface* into $\mathcal{GAUDI}$ *interactive scripting language*.

4 Summary

Appendix A:
Geant4 user classes

Straightforward communication of user program with *Geant4 environment* could be represented as communications between simulation environments, represented by with `G4RunManager` class and *user program* by set of *user classes* . *User environment* creates these classes, and *simulation environment* uses them. The sketch is presented on figure 4.

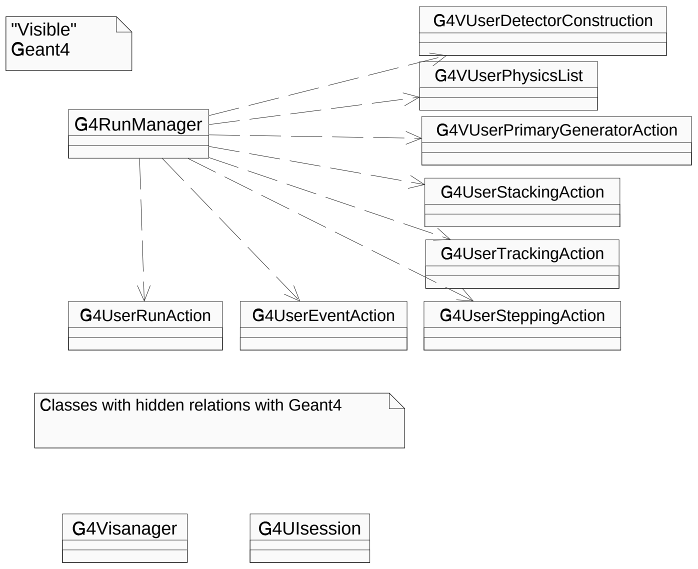

Figure 2: Sketch of classes, essential for configuration of *Geant4 Tool Kit*

`G4RunManager` class requires to be provided with 3 mandatory classes:

- `G4VUserDetectorConstruction`
- `G4VUserPhysicsList`
- `G4VUserPrimaryGeneratorAction`

In addition `G4RunManager` could be provided with:

- `G4UserRunAction`
- `G4UserEventAction`
- `G4UserStackingAction`
- `G4UserSteppingAction`
- $\overset{\frown}{G}$4UserTrackingAction

The exist also 2 classes, which are to be created explicitly within *user environment* and which communicate with `G4RunManager` in a hidden way:

- `G4VisManager` for visualisation
- `G4UIsession` for interactivity

Appendix B: Implementation details of GIGA Service

Class `GiGaSvc` implements both abstract interfaces *IGiGaSvc* and *IGiGaSetUpSvc*. This class is responsible for creation of `GiGaRunManager`, which in turn privately inherits from `G4RunManager` class. Schematic view is presented in figure 4. This implementation prevents the user from instantiation of more then one instance of `GiGaSvc` class.

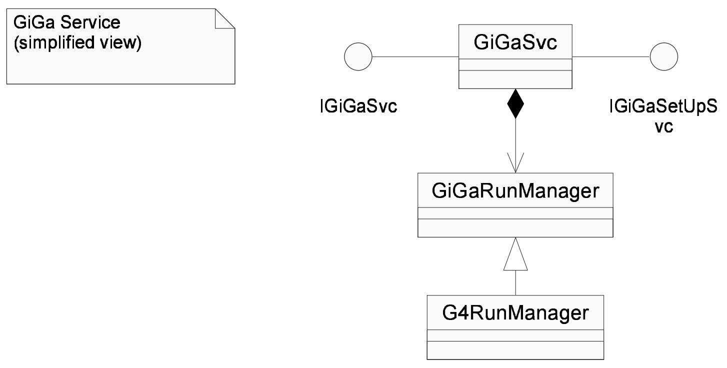

Figure 3: Schematic diagram of GIGA Service

The GIGA Service implementation contains only 2 essential methods - creation of `GiGaRunManager` and `GiGaKineManager` classes. The latter is used for transformation of `MCParticle` and `MCVertex` structures into `G4PrimaryVertex` structure and to be replaced in future by set of specialised *Converter*s. All other methods are just delegation to `GiGaRunManager` class.

`GiGaRunManager` class is the only non-trivial class in the whole chain. It privately inherits from `G4RunManager` class. The direct communication of user with this class are not foreseen. All communications are to be proceed only via GIGA Service.

Interface consists of 2 parts.

- Configuration part corresponds to functionality from *IGiGaSetUpSvc* interface of GIGA Service.

```
///////////////////////////////////////////////////////////////////////////////
///                                                                         ///
virtual StatusCode declare( G4VUserPrimaryGeneratorAction  * ) ;            ///
virtual StatusCode declare( G4VPhysicalVolume              * ) ;            ///
virtual StatusCode declare( G4VUserDetectorConstruction    * ) ;            ///
virtual StatusCode declare( G4VUserPhysicsList             * ) ;            ///
virtual StatusCode declare( G4UserRunAction                * ) ;            ///
virtual StatusCode declare( G4UserEventAction              * ) ;            ///
virtual StatusCode declare( G4UserStackingAction           * ) ;            ///
virtual StatusCode declare( G4UserSteppingAction           * ) ;            ///
virtual StatusCode declare( G4UserTrackingAction           * ) ;            ///
virtual StatusCode declare( G4VisManager                   * ) ;            ///
///                                                                         ///
///////////////////////////////////////////////////////////////////////////////
```

- Event management part corresponds to functionality from *IGiGaSvc* interface of GIGA Service.

```
///////////////////////////////////////////////////////////////////////////////
///                                                                         ///
virtual StatusCode  prepareTheEvent ( G4PrimaryVertex    * vertex = 0 ) ;   ///
virtual StatusCode  processTheEvent (                                 ) ;   ///
virtual StatusCode  retrieveTheEvent( const G4Event      *& event     ) ;   ///
///                                                                         ///
///////////////////////////////////////////////////////////////////////////////
```

All other methods are purely internal and irrelevant for external user, thats why all of them are protected or private.

Internally `GiGaRunManager` acts as *state machine*. Each state could be represented as *”4-bit word”*:

- *”Kernel Is Initialised”*
- *”Run Is Initialised”*
- *”Event Is Prepared”*
- *”Event Is Processed”*

Initial state of `GiGaRunManager` corresponds to the all flags are switched off.

Each communication with configuration part of interface resets this 4-bit state word to zero.

Action of each method from event management part depends on the concrete current state of *GiGaRunManager*.

- `retrieveTheEvent(const G4Event*& event)`
  If flag *”Event Is Processed”* is not activated, this method forces the processing of the event by `processTheEvent` method. Then if flag *”Event Is Processed”* is activated the method retrieves the current `G4Event*` object, otherwise it fails. Method makes sure that the flag *”Event Is Processed”* is activated in the case of success.

- `processTheEvent()`
  If flag *”Event Is Processed”* is activated, the method switched it off and delete the current event. Then if flag *”Event Is Prepared”* is not activated, this method forces the preparation of the event by `pre-pareTheEvent(0)` method. Then if flag *”Event Is Prepared”* is

activated the method start the actual precessing of event, otherwise it fails. Method makes sure that the flag *"Event Is Processed"* is activated in the case of success.

- `prepareTheEvent( G4PrimaryVertex* vertex = 0 )`
  If flag *"Event Is Processed"* is activated, the method switched it off and delete the current event. If flag *"Event Is Prepared"* is activated the method add the `vertex` to the current primary event record[6] If flag *"Event Is Prepared"* is not activated, method starts the real event preparation. Real Event preparation could be possible only if flag *"Run Is Initialised"* is activated, overwise method forced the run initialisation. If flag *"Run Is Initialises"* is activated, a new event is created and `vertex` is added to primary event record. Method makes sure that the flag *"Event Is Prepared"* is activated in the case of success.

Appendix C: GIGA **Examples**

A set of examples of usage of GIGA Service is prepared. This set corresponds to a usage of GIGA Service for the Phase I. All six standard novice examples from *Geant4 Tool Kit* distribution are configured to work within $\mathcal{GAUDI}$ framework.

[6]If `vertex` is `NULL` a new event would be generated using *G4VUserPrimaryGeneratorAction* and added to the current one