# GAUDI - A Software Architecture and Framework for building HEP Data Processing Applications

*G. Barrand*[1], *I. Belyaev*[2], *P. Binko*[3], *M. Cattaneo*[3], *R. Chytracek*[3], *G. Corti*[3], *M. Frank*[3], *G. Gracia*[3], *J. Harvey*[3], *E. van Herwijnen*[3], *P. Maley*[3], *P. Mato*[3], *S. Probst*[3], *F. Ranjard*[3]

[1] Laboratoire de l'Accélérateur Linéaire (LAL), Orsay, France
[2] Institute for Theoretical and Experimental Physics (ITEP), Moskva, Russia
[3] European Laboratory for Particle Physics (CERN), Genève, Switzerland

**Abstract**

We present a software architecture and framework that can be used to facilitate the development of data processing applications for High Energy Physics experiments. The development strategy follows an architecture-centric approach as a way of creating a resilient software framework that can withstand changes in requirements and technology over the long lifetimes of experiments. The software architecture, called GAUDI, supports event data processing applications that run in different processing environments, from the high level triggers in the on-line system to the final physics analysis. We present our major architectural design choices and outline the arguments that led to these choices. Several iterations of a software framework based on this architecture have been released and the framework is now being used by the physicists of the collaboration to facilitate the development of data processing algorithms. Object oriented technologies have been used throughout.

Keywords: LHCb, GAUDI, architecture, components, abstract interfaces, framework

## 1 Introduction

Experiments at the LHC collider, situated at CERN, will produce petabytes of raw data each year and these data must be reconstructed and analyzed to produce final physics results. The experiments are expected to run for many years and therefore changes in software requirements and in the technologies used to build software must be envisaged. Special thought must therefore be given to develop flexible and adaptable software that can withstand these changes and which can be easily maintained over the long timescales involved.

With these goals in mind we have started to construct a new object-oriented framework, GAUDI [1], with the intention of using it to develop all event data processing applications running in the various processing environments. Examples include the high level triggers that acquire their data directly from the on-line system, the event simulation software that runs off-line in a batch environment and the event visualization software which is used interactively. The basis of the framework is the architecture which defines the basic components and how they are interfaced. The framework is real software that implements the architecture and ensures that its design features are respected. Use of the framework in all applications will help to ensure the integrity of the overall software design and will result in maximum reuse of the core software components.

Although GAUDI has been developed in the context of the LHCb experiment [2], it has been designed to be easily customizable, so that it can be adapted to the different tasks and integrated with components from other frameworks. It could easily be adapted for use in other experiments.

## 2 Development Strategy

Before the start of the new project, LHCb physicists were using applications developed using FORTRAN libraries that had been in widespread use for the development of software for the previous generation of experiments. However the decision was taken to construct the new software using object oriented design techniques and languages (C++). The change from one set of software programs to the other, and the change of paradigm in which they are built, represents a difficult step that has to be managed very carefully. It is clear that physicists need to continuously study the design of the detector and the physics performance that can be achieved using it. The provision of a new framework was therefore considered essential to simplify the migration from the old to the new software. Without this framework there is a danger that development continues indefinitely in FORTRAN, thereby generating more and more legacy code, or new frameworks are chosen in an independent fashion, leading to fragmentation and duplication of the computing effort.

Requirements were collected in the form of use-cases following informal discussions with physicists who had the task of writing the simulation, reconstruction and analysis algorithms. In some cases there was only a vague knowledge of the precise needs to begin with, but this knowledge improved greatly with time. This serves to underline the importance of continuously involving the users in the specification of the design, to ensure producing a framework which closely matches their needs.

To address these concerns, we have decided to approach the final software framework via incremental releases, adding to the functionality at each release according to the feedback and priorities given by the users. This can only be done using an architecture-driven approach, i.e. to identify a series of components with definite functionality and well defined interfaces which interact with each other to supply the whole functionality of the framework. Since all the components are essentially decoupled from each other, they can be implemented one at a time and in a minimal manner, i.e. supplying sufficient functionality to do their job, but without the many refinements that can be added at a later date. In addition a component can easily be replaced by another component which implements the appropriate interface and provides the appropriate functionality, making the use of ”third-party” software possible.

## 3 Design choices

In the following we describe the main design criteria adopted in the development of the GAUDI software architecture.

### 3.1 Separation between *data* and *algorithms*

Given the use of OO techniques, our decision to distinguish data objects from algorithm objects may appear confusing at first. For example, *hits* and *tracks* will be basically data objects; the algorithms that manipulate these data objects will be encapsulated in different objects such as *TrackFinder*. The methods in the data objects will be limited to manipulations of internal data members. An algorithm will, in general, process data objects of some type and produce new data objects of a different type. For example, the *TrackFinder* algorithm will produce *track* data objects from *hit* data objects.

We have identified three basic categories of data objects:

- *Event data* are data obtained from particle collisions, and their subsequent refinements (raw data, reconstructed data, analysis data, etc.).
- *Detector data* describe and qualify the detecting apparatus, and are used to interpret the event data (structure, geometry, calibration, alignment, environmental parameters).

- *Statistical data* result from processing a set of events (histograms, n-tuples).

### 3.2 Separation of *persistent* and *transient data*

This important design choice has been adopted for all categories of data. We believe that physics algorithms should not use directly the data objects in the persistency store but instead use transient objects. Moreover the two types of object should not know about each other. There are several reasons for this choice:

- Most of the physics related code will be independent of the technology used for object persistency. In fact, we plan to change from the current technology (ZEBRA) to an ODBMS technology, preserving as much as possible the investment in newly developed code.
- The optimization criteria for persistent and transient storage are very different. In the persistent world the goals are to optimize I/O performance, data size, and to avoid data duplication to avoid inconsistencies. On the other hand, in the transient world the goals are to optimize execution performance and ease of use; data duplication can be afforded if it helps to improve performance and ease of use.
- Existing external components can be plugged into the architecture by interfacing them to the data. If they are interfaced to the transient data model, the investment can be reused in many different types of applications, whether or not persistency is required. In particular, the transient data can be used to connect two independent components.

### 3.3 Data store centered architectural style

In this design, the flow of data between algorithms proceeds via the transient data store. This minimizes the coupling between independent algorithms thus allowing the development of physics algorithms in a fairly independent way. Some algorithms will be ”producing” new data objects in the data store whereas others will be ”consuming” them. In order for this to work, the newly produced data objects need to be ”registered” into the data store so that other algorithms may identify them by some ”logical” addressing scheme.

### 3.4 Localization of encapsulated *user code*

The framework needs to be customized when used by different data processing applications in various environments. Usually this customization will be in terms of new specific code and new data structures. Therefore we must create a number of ”place holders” where the physics and subdetector specific code will eventually be added. We envisage two main places: ***Algorithms*** and ***Converters***.

### 3.5 Well defined generic component *interfaces*

Each component of the architecture will implement a number of interfaces (pure abstract classes in C++) for interacting with the other components, as shown in Figure 1). An interface consists of a set of functions specialized for some type of interaction, and should be as generic as possible, i.e. independent of the actual implementation of the components and of the concrete data types that will be added by users when customizing the framework. In order to ease the integration of components we need to select or define an interface model (or component model). This model should have support for interface versioning, dynamic interface discovery and generic component factories. With these features it should be possible to implement run-time loading of components (dynamic libraries) to allow us to implement true plug-and-play data processing applications.

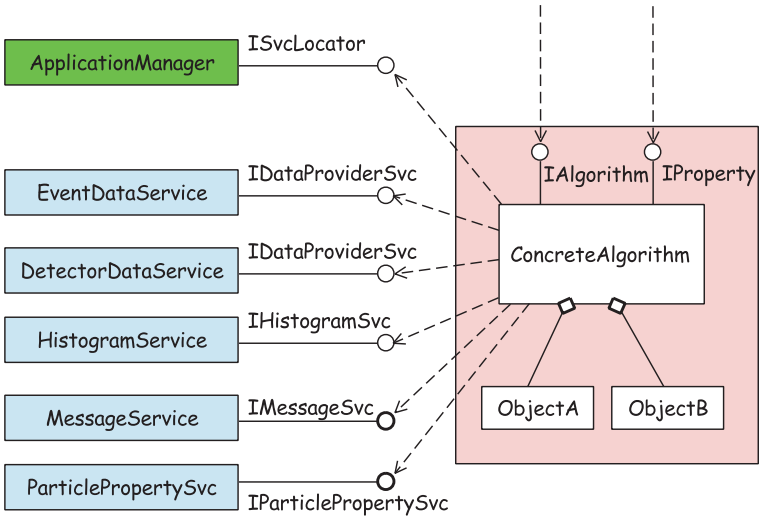


**Figure 1:** Example of interaction via interfaces. The Algorithm class *provides* the *IAlgorithm* and *IProperty* interfaces, and *uses* the various Service interfaces.

### 3.6 Re-use of standard components

We have a single development team covering the traditional domains of off-line and on-line computing. We plan to use the same framework throughout the system, so we should be able to identify and re-use components which are the same or very similar.

## 4 The GAUDI Architecture

The main components of the GAUDI software architecture can be seen in the object diagram shown in (Figure 2). Object diagrams are very illustrative for explaining how a system is decomposed. They represent a hypothetical snapshot of the state of the system, showing the objects (in our case component instances) and their relationships in terms of ownership and usage. They do not illustrate the structure, i.e. class hierarchy, of the software.

### 4.1 Algorithms and Application Manager

The essence of the event data processing applications are the physics algorithms, which are encapsulated into a set of components that we call *algorithms*. Algorithms implement a standard set of generic interfaces and can be called without knowing what they really do. In fact, a complex algorithm can be implemented by using a set of simpler ones. At the top of the hierarchy of algorithms sits the application manager, which knows which algorithms to instantiate and when to call them.

Algorithms are scheduled to be executed explicitly. A complex algorithm schedules directly the execution of the sub-algorithms in the proper order to produce the desired results. If an algorithm requires some data that happens to be produced by another algorithm, then the system has to ensure by explicit coding that the algorithms are executed in the correct sequence. Figure 3 illustrates how a particular dataflow can be achieved by use of the transient data store and appropriate algorithm sequencing.

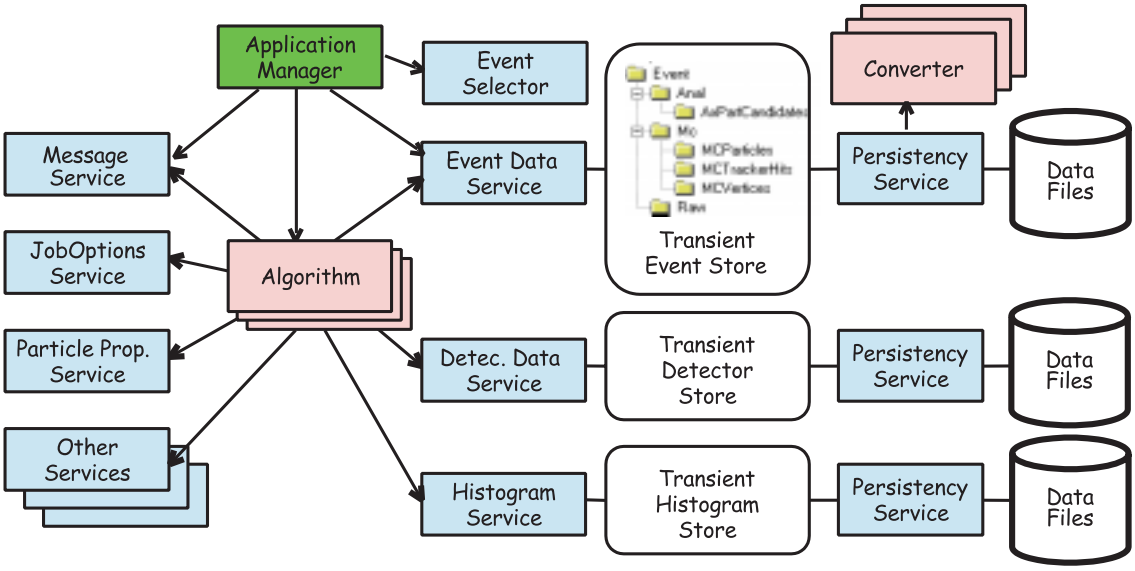


**Figure 2:** Object Diagram of the GAUDI Architecture

### 4.2 Transient data stores

The data objects needed by the algorithms are organized in several transient data stores, depending on the nature of the data itself and its lifetime. The Transient Event Store contains event data that are valid only for the time it takes to process one event. The Transient Detector Store contains data that describe various aspects of the behavior of the detector (e.g. alignment) and generally have a lifetime that corresponds to the processing of many events. The Transient Histogram Store contains statistical data, which typically have a lifetime corresponding to the data processed in a complete job. Although the stores behave slightly differently, particularly with respect to the data lifetime (e.g. the event data store is cleared for each event), their implementations have many things in common and are based on a common component.

A transient store helps to minimize coupling between algorithm objects and data objects. This approach was inspired by the work done in the BaBar experiment [3]. An algorithm can deposit some piece of data into the transient store, and these data can be picked up later by other algorithms for further processing without knowing how they were created. This conforms to the ”blackboard” architectural style, in which the transient store fulfils the role of the blackboard.

The transient data store also serves as an intermediate buffer for any type of data conversion to another representation of the data, in particular the conversion into persistent objects or graphical objects. Thus data can have one transient representation and zero or more persistent or graphical representations.

The organisation of the data within the transient data stores is ”tree-like”, similar to a Unix file system. This allows data items that are logically related, such as Monte Carlo ”truth” information, to be structured and grouped at run-time. Each node in the tree may either contain data members, or other nodes containing further groups of data members (Figure 4). As in a directory structure, each node is the *owner* of everything below it and will delete all these items when it gets deleted. In general, object-oriented data models do not map onto a tree structure. Thus, mesh-like object associations have been implemented using symbolic links (again inspired from the Unix file system) in which the node does not acquire ownership of the referenced item.

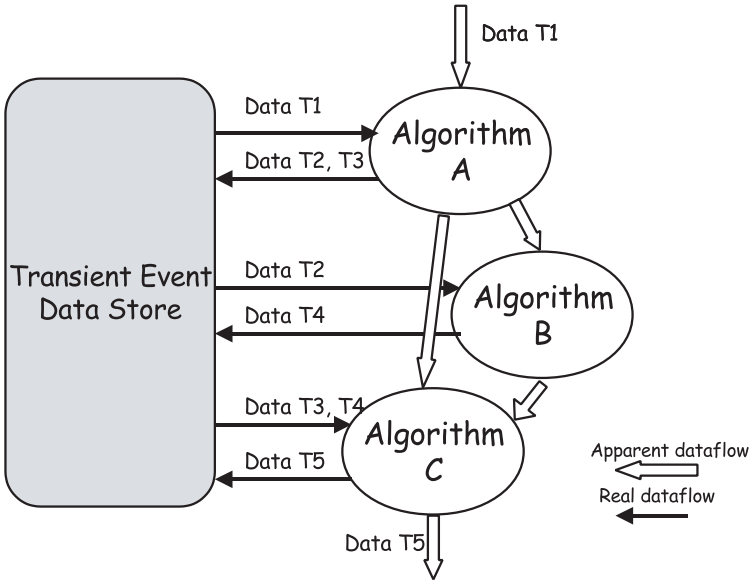


**Figure 3:** An example of implementing a dataflow by properly scheduling a number of algorithms

### 4.3 Services

These are a category of components that should offer all the services directly or indirectly needed by the algorithms. This approach releases the algorithm builder from having to develop the routine software tasks that are typically needed in a physics data processing application. Some examples of services can be seen in the object diagram (see Figure 2). They include:

- The services for managing the different transient stores (event data service, detector data service,...) should offer simplified data access to the algorithms.
- The different persistency services provide the functionality needed to populate the transient data stores from persistent data and vice versa. These services require the help of specific **converters** which know how to convert a specific data object from its persistent representation into its transient one, or the other way around.
- The job options service, message service and particle properties service.

Other services, such as visualisation and event selectors, are also part of the architecture. It is planned to implement many of the services using third party components.

## 5 Object Persistency

Several important considerations have led us to conclude that our software architecture should support in a transparent way the use of different persistency solutions for managing the various types of data that must be treated in our data processing applications. Firstly the volumes for the different data categories vary by many orders of magnitude. The event data from the different processing stages (raw data, reconstructed data and summary data) account for roughly 0.5 PB/year, while the detector data and event catalogues demand a few TB/year. Configuration and bookkeeping data will require only a few MB per year. Different access patterns are typical for these different data stores e.g. write-once/read-many for event data, read and write many for other data, sequential access, random access, etc. In addition, the software framework should support storage and access to legacy data. The simulated data currently used for the detector design studies are stored using ZEBRA [4] and the data produced in some of the test beams are made persistent

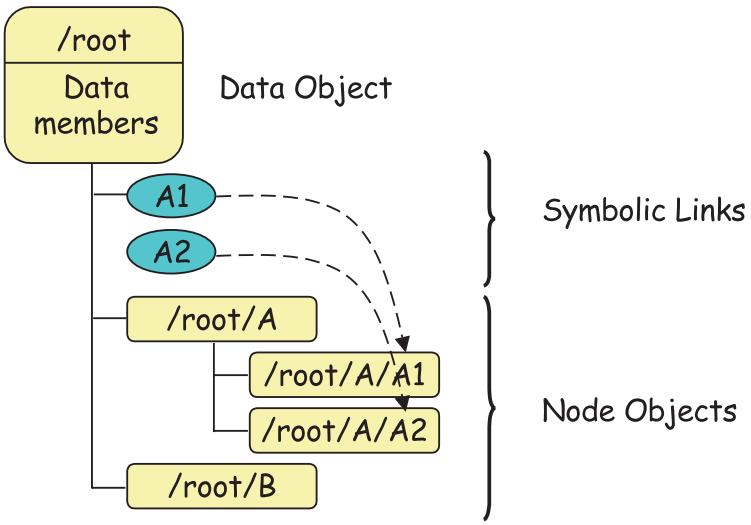


**Figure 4:** A node in the transient store.

using the ROOT framework [5]. For these reasons we believe that a single persistency technology may not be optimal in all cases. The GAUDI software architecture has been designed such that the best-adapted technology can be used for each category of data and allow a smooth integration of the existing data. This approach will also allow us to evolve smoothly with time to more appropriate solutions as they appear in the future.

### 5.1 Data access and data conversion

There are several options for maintaining transient and persistent data representations. One is to describe the user-data types within the persistent storage framework (meta-data) and have utilities able to automatically create both representations using this meta-data. This approach is elegant and relatively easy for basic data types, but is complicated when converting objects with many relationships. Another possibility is to code the conversion specifically for each data type, and this is the approach currently chosen in GAUDI. A *Converter*, with a common interface, is called whenever an object needs to be created in another representation. Each *Converter* is able to convert objects of one type (given by a class identifier) between the transient and one other representation (given by the representation type). The *Converter* can perform complicated operations, such as the combination of many small transient objects into a single object in order to minimise overhead in storage space and I/O. When converted to the transient representation, the persistent representation is expanded to the individual objects. This flexibility is only possible if the code is specifically written.

Every request for an object from the data service invokes the sequence shown in Figure 5:

- The data service searches the data store for the requested object. If the object exists, a reference is returned and the sequence ends.
- Otherwise the request is forwarded to the persistency service. The persistency service dispatches the request to the appropriate conversion service capable of handling the specified storage technology. The selected conversion service uses a set of data converters and forwards the request to the proper converter.
- The converter accesses the persistent data store, creates the requested transient object and returns it to the conversion service.
- The conversion service registers the object with the data store and the data service subse-

quently returns a reference to the client.

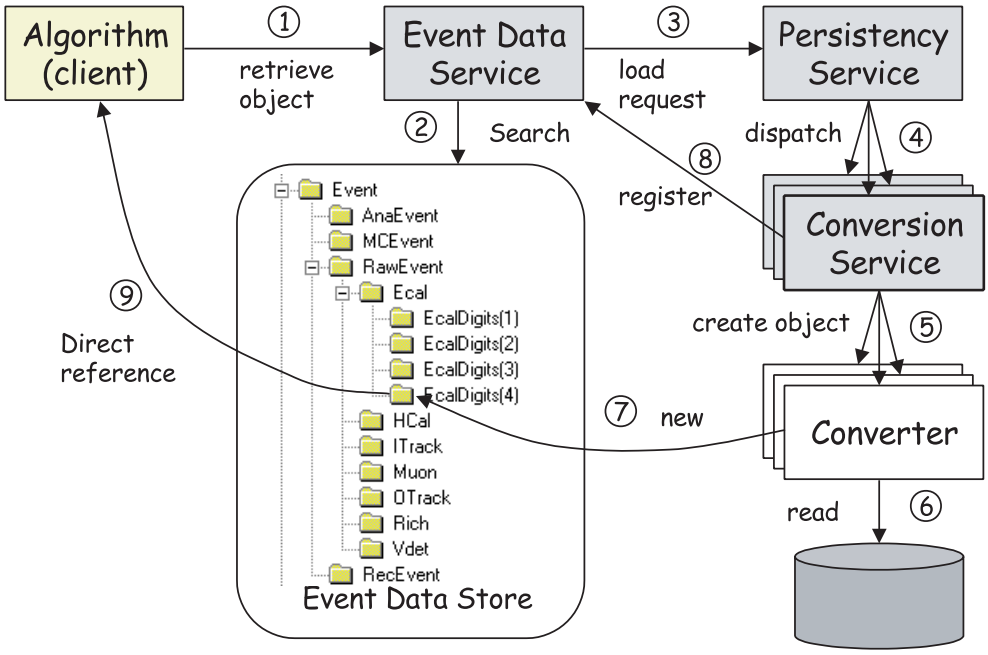


**Figure 5:** Action sequence for loading an object from the persistent medium.

When making objects persistent, the calling sequence is as follows:

- An agent traverses the data tree using the data service collecting references to objects that are to be made persistent.
- The collected references are passed to the persistency service, which dispatches each object referenced to the appropriate data converter.
- Each converter allocates persistent storage on the requested medium and fills the allocated area according to the transient object.
- As a last step all persistent references are filled and the updated persistent objects stored.

Note that this mechanism can also be applied to conversions to other data representations, such as those used in visualization.

### 5.2 The generic persistent model

A simple generic mechanism has been developed for converting data between their transient and persistent representations and for resolving, on demand, associations through different persistent solutions. This mechanism is used for ROOT, Objectivity/DB [6] and ZEBRA (legacy) data.

Our schema assumes that most database technologies are based on files or logical files. Internally these files are partitioned into containers (database containers in Objectivity/DB, ”Root trees” for ROOT I/O, tables for an RDMS)and objects populating these containers. An extended object identifier (XID) has been introduced (see Figure 6) containing all relevant information.

If the XID represents an object in an ODBMS, the storage-dependent part is identical to the OID. Otherwise all information required to locate the database file, the container/table and the corresponding record must be determined from this part.

The combination of object type, storage type, database name, container name, object name/path in the transient store and the object identifier within the container (record ID) form a universal address, which allows data items to be addressed in any technology. To minimize persistent storage,

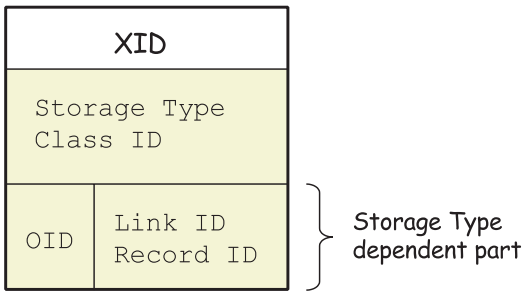


**Figure 6:** Layout of the extended object identifier (XID)

this address is split, with database name, container name and object name path stored in a separate lookup table, identified by a link ID.

We have introduced a set of *smart pointers* that delay the actual loading of the associated objects and encapsulates the XID from the end-user to ease navigation within the data model.

## 6 Detector Description

Figure 7 illustrates the way in which *Algorithms* access detector data through a transient representation (the Transient Detector Store) and how this store is populated from the detector description database. Conversion to other views (representations) is carried out by *Conversion Services*. Examples of other representations are graphical views and the geometry description used by the GEANT4 simulation toolkit [7].

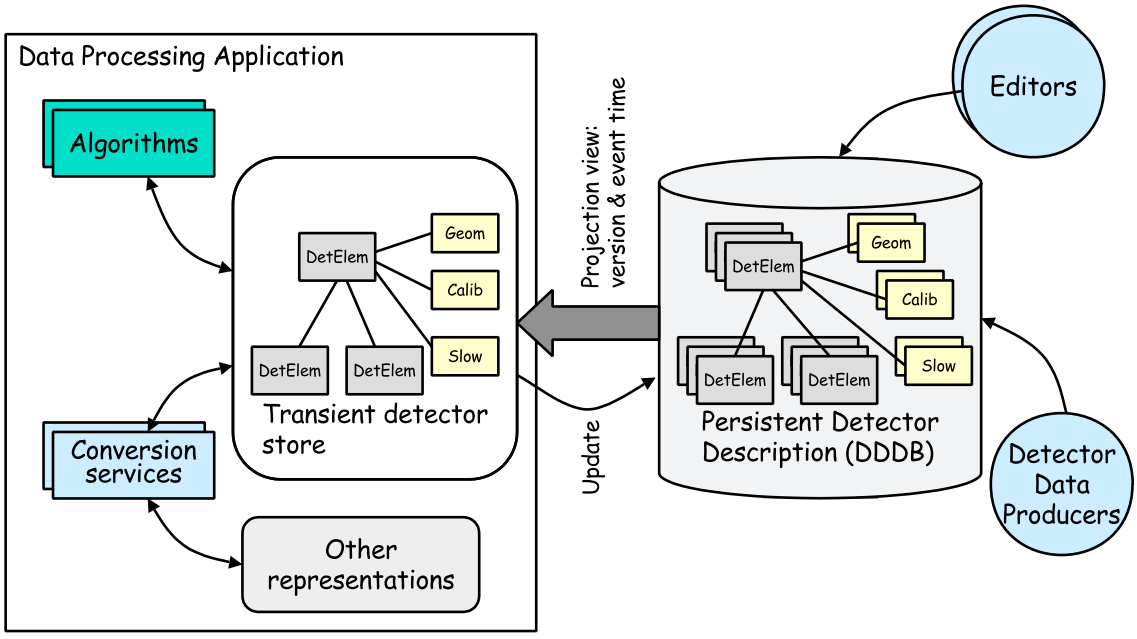


**Figure 7:** Detector description overview

The detector database includes a physical and a logical description of the structure of the detector. The physical description covers the dimensions, shape and material of the different types of elements from which the detector is constructed as well as information specific to each

element which is actually manufactured and assembled into a detector. Both active and passive elements are included. The description of active elements allows for the specification of deficiencies (dead channels), mapping to electronic channels, alignment corrections, calibration of electronic responses, etc.
The logical description provides two main functions.

- To provide simplified access to particular parts of the physical detector description via a hierarchical structure in which a given detector setup is composed of various sub-detectors, each made up of a number stations, modules or layers, etc. There is a simple way for a client to use this description to navigate to the information of interest.
- To provide a means of identifying detector elements. This allows different sets of information correlated to specific detector elements to be correctly associated with each other.

The definition of the detector elements and the data associated to their physical description may vary over time due to real changes to the detector. It is foreseen that such changes are recorded as a different version of the detector element. Additionally, it is also foreseen to capture for a coherent description of the detector a specific version of each element in the detector and to associate a name to that set. This is similar to the way CVS [8] allows a set of files to be tagged so that knowledge of the independent version numbers for each file in the set is not required.

### 6.1 Transient Detector Store structure

The transient representation of the detector description is implemented as a standard GAUDI data store. The data objects are organised in a hierarchical tree structure. The transient store is populated on the demand of *Algorithms* with data objects of a given version and whose validity include the time of the event currently being processed. It is the responsibility of the *Detector Data Service* to make sure the elements in the transient store are up to date.

At present, there are three top level catalogs in the transient store: detector setups (logical structure), geometry elements, and materials. Other catalogs will be added when needed.

Links between objects in different catalogs, e.g. the association of a solid to a material, a detector element to its geometry information, etc. are resolved at runtime when needing to traverse them. For example, a material object corresponding to a detector element will only be loaded when knowledge of the material properties of that detector element is needed.

#### 6.1.1 Detector Setup

A detector setup is described using a tree of detector elements. More than one setup can be loaded concurrently in the transient store. The complete identification of a detector element is by a name in the form ”/dd/Structure/LHCb/Muon/Stations/Station1”. The generic *DetElement* class may be specialized in order to provide answers to concrete questions from the *Algorithms*.

#### 6.1.2 Geometry

The detector geometry is described by a hierarchy of *logical* and *physical* volumes, following very closely the approach used in the Geant4 toolkit [7]. This ensures an easy conversion to Geant4 objects for the simulation application. A logical volume is an unplaced volume described in terms of a solid with a shape or a boolean operation of solids, a material and a number of placed subvolumes (physical volumes). The simplified class diagram showing the relationships between logical and physical volumes, solids and materials is shown in Figure 8.

Transient geometry objects provide basic geometrical functionality to *Algorithms*. Examples are local to global coordinate transformations, finding out if a given 3D point is inside a detector volume, returning the complete hierarchy of volumes containing a given point, etc.

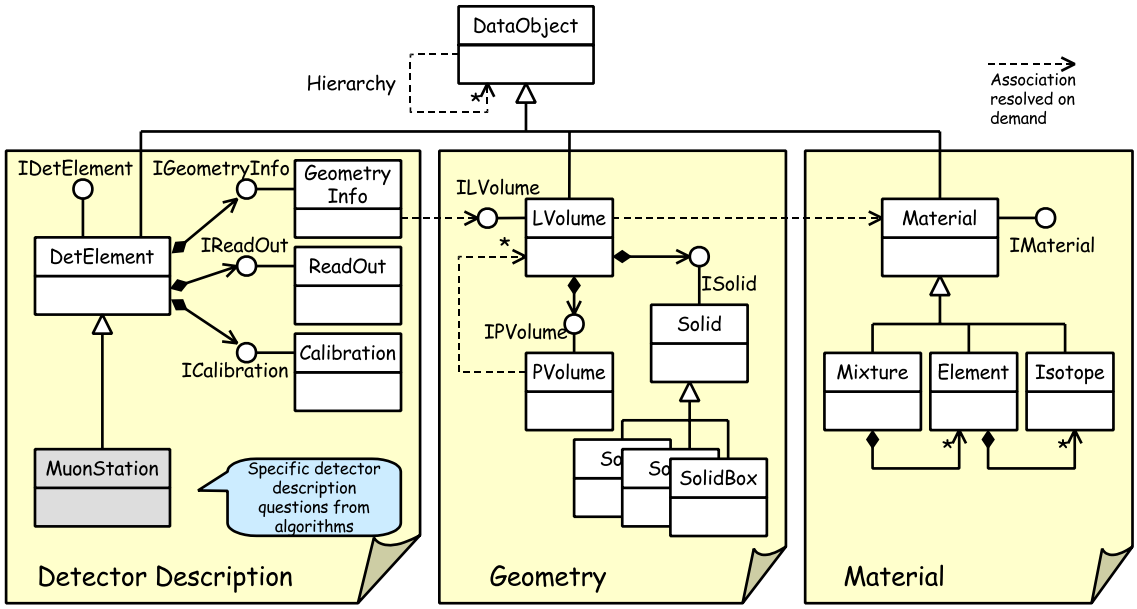


**Figure 8:** Simplified class diagram for the Detector Description

### 6.1.3 Materials

Materials are associated with logical volumes. They are defined as isotopes, elements or mixtures. Elements can optionally be composed of isotopes. Composition is always done by specifying the fraction of the mass. Mixtures can be composed of elements or other mixtures. For a mixture the user can specify composition either by number of atoms or by fraction of mass.

## 6.2 Persistent representation

The current implementation of the persistent detector description model is based on plain text files whose structure is described by XML (eXtendible Markup Language) [9]. XML is an application independent format for describing data. It has recently acquired wide support, being an industry standard issued by the W3 Organization. XML files are self describing and can therefore be directly understood. Using XML a custom markup language can easily be defined for describing a specific application domain.

The structure of an XML document can be easily represented as a tree of XML elements, their attributes and content. This tree structure fits very well into the hierarchical navigation schema used by object-oriented systems. The basic problem is the proper translation from the source OO-language (in our case transient C++ data structures) into the corresponding XML form (persistent data). The translation rules used in the current implementation are shown in Table I.

**Table I:** Mapping C++ to XML

| **C++** | **XML** |
|---|---|
| Class | XML Element |
| Class data members | Element attributes |
| Composition | Element embedded in Content Model |
| Reference | Element referenced with a hyperlink |

#### 6.2.1 XML converters

Following the GAUDI approach of distinguishing transient and persistent representations, a set of XML converters (one per data type) have been built using the SAX [10] interface of the XML parser. The current implementation of the converters provides read-only XML data but work has been started to allow the converters to write to the persistent store.

#### 6.2.2 Hyperlinks

XML supports hyperlinks in a similar way to HTML. This feature helps to split the XML-based database into multiple files that can be maintained independently of each other, minimizing coupling between developments of different sub-detectors. In addition it is possible to define in XML the persistent references between different pieces of data in a way similar to C++ or C. One can thus refer to objects inside the same file or objects located somewhere on the network and it is possible to use standard internet protocols such as http or ftp to retrieve them.

### 6.3 User Customization

A description of a specific detector can be made available to algorithms by customizing a generic detector element. Customization is done by inheritance of the *DetectorElement* base class. Algorithms can request specific information concerning a particular detector and this is then calculated from the generic data describing the overall geometry, orientation and material and from more specific details describing the particular structure of the detector. The answer is coded using the minimal number of parameters specific to the detector being described. For example, an *Algorithm* may need the local position of a calorimeter cell knowing its cell ID. In this case, probably, a simple parameterized equation in a *detector element* method can give the answer.

## 7 Status of GAUDI and Outlook

Five incremental releases of the GAUDI software framework have been made and it is now being used to construct real applications. Development is proceeding by expanding the number of basic services, including a service that integrates the GEANT4 toolkit and manages the simulation of the detector, and a data visualization service for building event and geometry displays. In the area of data persistency, work is continuing by further development of ”generic” data converters that can be used for more than one data class in order to minimize the work that has to be done by end-users. In addition, we are incorporating the creation and handling of event tag collections within the framework, which will allow selection of events to be processed based on complex selection criteria without the need to download the complete event. The framework is also being extended to include a Detector Conditions Database that contains all the information of the detector (calibration, alignment, environmental data, etc.). Data of this type is always tagged according to its time validity period. The framework should provide an easy way for algorithms to access up to date calibration and alignment data in a transparent manner.

## References


1 The GAUDI architecture and software are documented on `http://lhcb.cern.ch/computing/Components/html/GaudiMain.html`

2 S.Amato et al., *LHCb Technical proposal*, CERN/LHCC 98-4.

3 S.Gowdy et al., *Hiding Persistency when using the BaBar Database*, CHEP 1998, Proceedings.

4 R.Brun, *ZEBRA - Reference Manual - RZ Random Access Package*, Program Library Q100. CERN, 1991
5 R.Brun and F.Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86
6 *Objectivity Technical Overview*, Objectivity Inc., `http://www.objectivity.com`
7 S. Giani et al., *GEANT4: An Object-Oriented Toolkit for Simulation in HEP*, CERN/LHCC/98-44 (see also `http://wwwinfo.cern.ch/asd/geant4/geant4.html`).
8 *CVS - Code Versioning System*, `http://www.cyclic.com/`
9 The World Wide Web Consortium, *Extensible Markup Language (XML)*, `http://www.w3.org/XML`
10 *SAX - The Simple API for XML*, `http://www.megginson.com/SAX/index.html`