GAUDI - The Software Architecture and Framework for building LHCb 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] *B. Jost*[3], *I. Last*[3], *P. Maley*[3], *P. Mato*[3], *S. Probst*[3], *F. Ranjard*[3], *A. Tsaregorodtsev*[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 the strategy that has been adopted for the development of the software system for the LHCb experiment. This strategy follows an architecture-centric approach as a way of creating a resilient software framework that can withstand changes in requirements and technology. The software architecture, called GAUDI, covers event data processing applications in all processing stages 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

The goal of the GAUDI project [1] is to build a framework which can be applied to a wide range of applications, covering all stages of the data processing of a high energy physics experiment: physics and detector simulation, high level software triggers, reconstruction, physics analysis, visualization, etc., in a wide range of different environments: interactive and batch applications, off-line and on-line programs, etc. A framework is an implementation of an architecture. It is therefore important that the underlying architecture can work in the different environments and data processing stages. The framework should be customisable, so that it can be adapted to the different tasks and integrated with components from other frameworks, thus permitting re-use of the major functional elements. As a consequence, the resulting system could be easily adaptable to experiments other than LHCb, for which it is being initially developed.

2   Development Strategy

The first step of the development was to identify the main system requirements. The inputs were our own experience of previous HEP experiments, and (very informal) discussions with some future users of the system (mainly physicist developers). We studied some simulation, reconstruction and analysis processes to extract example usages of the system (use cases). Ideas from, and limitations of, the software systems of previous experiments were also an important input.

The design of the framework is driven by the requirements of the physicists who will develop reconstruction and simulation code and who will analyse the data. Although they know in broad terms what they want to do, the details are as yet unknown, so it is crucial that the development does not proceed in isolation but intimately involves the future users, to avoid producing a framework which does not match with their needs. Another constraint is that physicists need to

continuously study the detector and the physics. If a framework is not provided, they will either keep working in FORTRAN, thereby generating more and more legacy code, or they will look for a framework themselves, leading to fragmentation and duplication of the computing effort.

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 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 strategic decisions taken and the main design criteria adopted in the development of the GAUDI software architecture.

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.

Three basic categories of data objects: event, detector and statistical data.
- *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).

Separation between *persistent data* 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.

Data store -centered architectural style. In this design, the flow of data between algorithms proceeds via the transient data storage. This minimises 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 somehow "registered" into the data store so that other algorithms may identify them by some "logical" addressing scheme.

Encapsulated *User code* localized in a few specific places. The framework needs to be customised when used by different data processing applications in various environments. Usually this customisation 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 sub-detector specific code will eventually be added. We envisage two main places: ***Algorithms*** and ***Converters***.

Well defined component *interfaces*, as *generic* as possible. 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 customising 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.
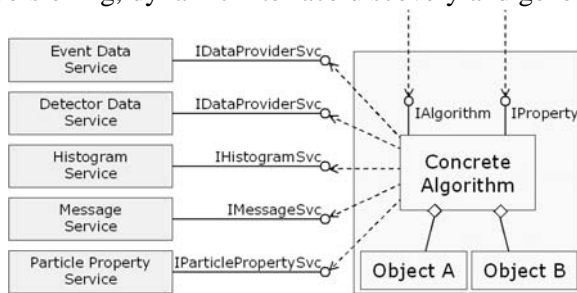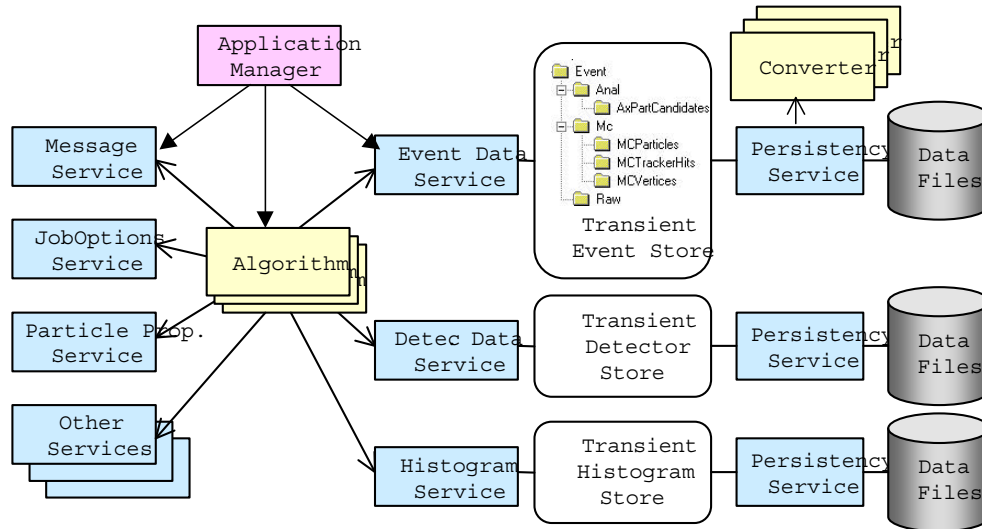

Figure 1:
Example of interaction via interfaces.
The Algorithm class *provides* the
IAlgorithm and *IProperty* interfaces,
and *uses* the various Service interfaces.

Re-use standard components wherever possible. 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

We introduce our description of the architecture by an object diagram showing the main components of the system (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 represent the structure (class hierarchy) of the software.

Algorithms and Application Manager. The essence of the event data processing applications are the physics algorithms, which we encapsulate into a set of components that we called 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.

Figure 2: Object Diagram of the GAUDI Architecture

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.

Event data, valid only during the time it takes to process one event, is stored in the transient event store. Detector data, generally with a lifetime of many events, is stored in the transient detector store. Statistical data, generally with a lifetime of the complete job, is stored in transient histogram and n-tuple stores. The four stores behave slightly differently, at least with respect to the data lifetime (e.g. the event data store is cleared for each event), but their implementations have many things in common and are based a common component.

Services are a category of components which should offer all the services directly or indirectly needed by the algorithms, so that physicists can concentrate on developing the physics content of the algorithms and not on the technicalities. Some examples can be seen in the object diagram:

- 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 [2], [3] 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   Status of GAUDI

We have developed an architecture for high energy physics data processing software. Three incremental releases of a C++ software framework based on this architecture have been made and the framework is now being used by the physicists of the LHCb collaboration.

References

1   The GAUDI architecture and software are documented on WWW at the URL:
    `http://lhcb.cern.ch/computing/Components/html/GaudiMain.html`
2   G.Barrand et al., Data Persistency solution for LHCb, CHEP2000 proceedings.
3   G.Barrand et al., The LHCb Detector Description Framework, CHEP2000 proceedings.