IC/HEP/98-06

Object oriented reconstruction software for the BaBar calorimeter

Stephen J. Gowdy[a], Helmut Marsiske[b] and Philip Strother[c1]
(a) *Dept. of Physics, University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, Scotland. SGowdy@lbl.gov*
(b) *SLAC, P.O. Box 4349, Stanford, CA94309, USA. marsiske@slac.stanford.edu*
(c) *The Blackett Laboratory, Imperial College, Prince Consort Road, London SW7 2AZ, UK. pd.strother@ic.ac.uk*

Abstract

The object oriented software used in reconstruction for the BaBar electromagnetic calorimeter is described. The basic reconstruction objects and their abstractions are presented. Techniques employed in the design of clustering, cluster division, track matching, particle identification and global calibration software are discussed in terms of their ability to provide a core interface while allowing for improvements in physics capability.

1 The BaBar Calorimeter

The BaBar Calorimeter is a high precision Thallium doped Caesium Iodide device consisting of a 48×120 crystal barrel and an 8 row forward endcap graded between 80 and 120 crystals. The expected resolution is $1\%/\sqrt[4]{E(GeV)} \oplus 1.2\%$. The operating conditions at the PEP-II asymmetric collider for which it is built are expected to be harsh, with between 1–3 GeV of soft photon background (resulting from lost beam particles) being deposited in the calorimeter per μs. The combination of high precision and difficult operating conditions creates the need for flexible software with good scope to allow for improvements in physics capability. The approach taken has been to use the principles of object oriented design to give a C++ implementation that defines the basic observables and reconstruction framework in such a way as to permit the external clients of the calorimeter software to code to a well defined interface, whilst allowing for improved algorithms and implementations which do not force clients to recode.

2 Basic Objects

The Digi class represents the most basic front end electronics output. Virtual accessors to the energy and time are provided, along with a channel identifier. Two subclasses of Digi exist. SharedDigi overrides the energy function to provide a weighted energy. This concept is used in cluster division, which will be discussed

[1]Contact author.

later. CalibratedDigi overrides the energy and time accessors to incorporate an individual crystal calibration given at construction time.

The Cluster class is designed to encapsulate the aspects of an energy deposit in the calorimeter. The principal interface provides information on the energy, position and error matrix associated with the cluster. Its implementation is in terms of a collection of pointers to Digi objects. Most clustering algorithms insist that this collection be contiguous, but this is not enforced at this level. More complicated information on the distribution of crystals within the cluster, and different estimations of the position of the cluster are provided by the derivatives of an AbsClusterProperty class, which are constructed by the Cluster on demand. This allows for the extension of the number of quantities so calculated without having to recompile external code which depends on the Cluster class.

The Bump class is a derivative of Cluster and typically contains only one electromagnetic maximum, although this is not enforced at the design level. A Bump is constructed with SharedDigis, leading to re-use of the Cluster position and energy calculation code, and complete transparency between the two classes. A Bump will usually have as many SharedDigis as the Cluster from which it was formed.

The Cand (candidate) class is intended to represent a generic particle observation in the calorimeter. It contains a collection of pointers to Clusters and also contains a pointer to a track object [1] which may be null. The functions of position, energy and error matrix provide calibrated forms of these quantities using information in a *AbsCalibrator* object handed to the Cand at construction time. Access to a PidInfo object, discussed below, is provided.

The PidInfo class allows for access to information regarding the compatibility of a given observation to the expected observation for a given particle hypothesis. The implementation places no restrictions on the number of hypotheses tested, nor on the number of of algorithms employed to identify a given particle type.

2.1 Basic Object Abstraction

The Cand and Cluster classes, and consequently the Bump class, all inherit from an abstract base class AbsRecoCalo. This class is designed to provide uniformity within the calorimeter software and also in its use to describe reconstructed objects in BaBar's instrumented flux return software, to provide cross system abstraction. Pure virtual position, four momentum and error matrix functions are provided, as well as rudimentary genealogy for use in analysis software. Almost all external clients of the calorimeter code make use of the AbsRecoCalo abstraction, and do not refer to specific implementations of calorimeter observations directly.

3 Algorithm framework

The above objects are generated by a series of Modules in the BaBar Framework[2]. These modules perform the basic sequence of cluster formation, cluster division, track matching and particle identification. With the exception of cluster formation

which is considered a sufficiently low level operation as not to necessitate complicated design, each of these tasks use abstract classes to define the basic operation(s) of the algorithm in question. The Framework modules are then configured at run time using particular subclasses of these. This approach is intended to provide a very simple method of algorithm upgrade and to minimise the coupling between the clients of the calorimeter Framework Modules and the specific algorithms employed.

3.1 Cluster division

It is desirable to separate the clusters into regions containing one and only one local maxima. Two distinct operations are involved in this division: the location of crystals which are potential maxima, and the sharing of the energies of the crystals in the cluster among these. These operations are mapped onto two abstractions which are implemented in a Strategy pattern[3]: firstly an abstract local maxima finder which takes a cluster as input and outputs a list of the crystal indices of the local maxima. Secondly an abstract bump splitter whose purpose is to take the list of local maxima, divide the cluster by the creation of Shared Digis, and return a list of newly created bumps. A Factory method [3] is used to configure the Framework module with a particular pair of maxima finder and bump splitter at run time.

Two local maxima finders and three splitting algorithms have been implemented to date, the majority of these by non-expert C++ programmers. This seen as an indication that the abstraction is at the right level to achieve both decoupling and the rapid deployment of new algorithms once new understanding is gained.

3.2 Track–cluster matching

The association of a given energy deposit to a track is not always unambiguous. For this reason it was decided that the default output should not be lists of "charged" and "neutral" calorimeter observations, but rather that the output for use in physics analysis should be associations between tracks and clusters at some confidence level. Further, it is envisaged that several algorithms for track matching will be developed over the course of the experiment.

The abstraction of the mechanisms used to match tracks to clusters is achieved by a class, *AbsTrkMatchMethod* which defines two pure virtual methods, one for matching tracks to clusters, and one for the reverse operation. These return association maps (see below) between the tracks, clusters and objects containing information on the quality of the match to the event. It has not yet been demonstrated that the flexibility of allowing different algorithms for the two different operations is necessary. A Framework module is again configured with differing algorithms by the use of a Factory method. Three progressively more efficient algorithms have already been encoded in this way.

Association map objects are used to provide links between clusters, tracks and *TMinfo* objects which contain information about the quality of the match, entry and exit points of the track and the track cluster separation. These are implemented as templated three-way, many to many association tables based on hash dictionaries.

The flexibility afforded by the use of such maps allows for decisions about the level of match quality used to decide between neutral and charged to be determined at run time. This flexibility has been found to be useful, but not always wanted by analysts. The provision of lists of calorimeter observations, divided into charged and neutral at some confidence level, and derived from the association maps, is also allowed for.

3.3 Particle identification

The central components of the design are the *CandidateMaker* class which is responsible for constructing the Cand objects that may be appropriate to the hypothesis under consideration and the *Identifier* class, an abstraction incorporating the concept of giving information about a given energy deposit pertaining to a certain particle hypothesis. Derivatives of each are linked together by a Subject-Observer pattern which ensures that each Cand constructed is passed to every active Identifier subclass. In general Identifier subclasses look at only one discriminating variable. In order to allow for multiple discriminating variable identification, and for neural network analyses, a specific abstract subclass of Identifier, CompositeIdentifier, is used, which employs a Template pattern [3] to allow the particular implementation to look at the results of several different Identifier derivatives and form a new conclusion based on this input.

Again a Framework module is configured with specific CandidateMakers and Identifiers via Factory patterns. This, in conjunction with the extensibility of the PidInfo object, allows for the extension of the number of particle types identified and the number of algorithms used to identify them with complete decoupling of all clients of the code from this extension.

4 Offline calibration framework

The energy deposits observed in the calorimeter will represent only some fraction of the energy of the incident particle at its point of creation. The experiences of other experiments suggest that the correction for this will become a function of more and more variables as knowledge of the detector improves. Furthermore, an analyst running over a channel with a small branching ratio may wish to reprocess events with the best possible calibration, i.e. one that takes in all the known functionality, even if this were not available when the data were first taken. A final requirement is that the calibration system be such that the origin of the correction function be allowed to change such that the default is always the best most general purpose correction, but that specific corrections remain available.

The requirement then is for a system that supports multiple algorithms, is extendible in its functional dependence and is retrospectively applicable. The coefficients of the functions used for the energy correction must be stored in the object-oriented database used by BaBar—Objectivity/DB[4]. This does not allow for storage of the functional form itself. The approach is to store the coefficients in the database, along with knowledge of the name of a transient class which contains the

encoding of the functional form appropriate to those coefficients. In addition, the usage of the database in BaBar dictates that the database be completely decoupled from client code.

The AbsCalibrator class contains the pure virtual functions which take a Cluster as input and return the corrected energy or position of that cluster according to the calibration information contained within the class. This class is then used by Cand objects to calibrate the Clusters they contain. The functional form of the calibration is described by subclasses. Verification of new versions of a particular calibrator is performed by means of a virtual function whose default implementation is to check that the new calibrator is of the same type, has the same length and the constants are within 20% of the previous set. More complicated verification procedures may be implemented by subclasses.

The CalibratorP class contains a persistent array of coefficients for a given calibration function, a persistent string which contains the name of the transient class (the AbsCalibrator derivative) to which these constants pertain, and a third string to describe the algorithm used to derive the constants. Subclasses of CalibratorP pertain to different algorithms and differ from the base class only in the content of the algorithm name string.

The interface between clients of the AbsCalibrator class and the database is provided by the CalibProxy class[2]. The proxy mechanism is such that a given AbsCalibrator subclass is held by the proxy and returned on demand until either the subclass relating to a different source of calibration data (referred to as an algorithm below) is requested, or the event time indicates that the current set of constants is out of date. Under these circumstances, the constants for a new derivative are retrieved from the database and a new transient AbsCalibrator derivative constructed appropriate to these constants, using the information stored along with them.

A CalibratorDictionary class is used to provide a persistent map between algorithm names and the persistent CalibratorPs. This removes the need for clients to know about the subclasses of CalibratorP. The CalibratorDictionary is responsible for returning the relevant AbsCalibrator derivative to the CalibProxy. The string contained within the CalibratorP derivative is translated into an AbsCalibrator derivative by means of a Factory. The dictionary then loads the constants contained in the CalibratorP object into the transient AbsCalibrator derivative.

There are two aspects to the permanent storage. One is the ability to store the coefficients of the functional forms encoded in the AbsCalibrator derivatives. This is a relatively trivial exercise. The other is the ability to keep track of available calibration algorithms and of the default calibration. To this end, a persistent capable class AlgDescription containing the algorithm name and a flag indicating whether or not that algorithm is the default is stored in the database inside an AlgBank object, while the transient equivalent used by client code is an AlgList. These are nothing more than persistent (Bank) and transient (List) vectors. They are retrieved by an AlgListProxy class is responsible which obtains from the database the relevant AlgBank object pertaining to the event time. This operation is essentially the same

[2]The Proxy mechanism borrows heavily from an initial implementation for the Silicon Vertex Tracker geometry by David Nathan Brown of the Lawrence Berkeley laboratory.

as that of the CalibProxy class.

Derivatives of the AbsCalibAlgorithm class is responsible for generation of the constants to be stored in the database in the form of an AbsCalibrator derivative. This object is then passed to the singleton class StoreCalibrator which performs the mechanics of translating the transient AbsCalibrator derivative into the appropriate CalibratorP derivative. Each subclass of AbsCalibAlgorithm must register itself with CalibAlgRegister which is responsible for keeping record of the available algorithms and an indication of the default.

4.1 Offline calibration: summary

The rather complicated design described above achieves several things at once. Firstly it separates the client code (coded in terms of AlgLists and AbsCalibrators) from the database implementation, as required. Secondly, in the form of the AlgList/Bank object, and the CalibDictionary's map between the AlgDescription objects and the CalibratorP derivatives, the extensibility of the calibration scheme when new and more complicated algorithms become available is achieved. Thirdly, since these algorithms make no reference to time, they are by definition retrospectively applicable providing the information on which they are based is recorded in the database at the appropriate time. Fourthly, by providing differing implementations of the calibration functions while retaining the external code's dependence on an abstraction, new and more complicated calibration functions can be applied without needing to change external code.

References

[1] S.F.Schaffner and J.M. LoSecco, "Object oriented tracking and vertexing at BaBar" presented at CHEP 1997, to appear in proceedings, and S.F.Schaffner, "BaBar's Object Oriented Tracking System", in these proceedings

[2] E. D. Frank, R. G. Jacobsen and E. Sexton-Kennedy, "Architecture of the BaBar Reconstruction System", presented at "CHEP 1997", to appear in proceedings.

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns", (Addision-Wesley, California, 1995)

[4] Objectivity/DB, Object Oriented Database product supplied by Objectivity Inc., Mountainview, California