Architecture of the BaBar Reconstruction System

E.D. Frank [a,1], Bob Jacobsen [b], Elizabeth Sexton–Kennedy [c,2]
for the BaBar Computing Group

[a] *Dept. of Physics and Astronomy, University of Pennsylvania, Philadelphia, USA*

[b] *University of California, Berkeley CA, USA*

[c] *Fermi National Accelerator Laboratory, Batavia, IL, USA*

The BaBar experiment at SLAC is building its reconstruction system in C++, almost from scratch. We describe here the problems we encountered when designing a system that follows traditional HEP processing paradigms, yet follows sound software engineering precepts. We also describe solutions under investigation. The context of the discussion is the architecture of a reconstruction application.

Key words: Event Reconstruction; Architecture; BaBar; C++

1 Introduction

An overview of the BaBar computing system has been given by Neil Geddes elsewhere in these proceedings. Summarizing that discussion from the point of view of reconstruction, 100 Hz of data, totaling 20 Tera–bytes per year, will be selected by the final trigger and passed to a prompt reconstruction farm. One percent of these events must be reconstructed within ten seconds and returned to an online event display. The remainder of the data must be reconstructed within a few hours of acquisition. The data must be ready for physics analysis at this point with no further passes over the data required: Reprocessing of the data will occur only under unusual circumstances. Thus, the reconstruction system is to operate in "factory mode," with all the reliability, verifiability, and reproducibility that that implies. We describe in this paper the internal architecture of a reconstruction application program.

[1] email: efrank@upenn5.hep.upenn.edu

[2] Operated by the Universities Research Association under contract with the U.S. Department of Energy.

The purpose of the application architecture is to provide a programmer with all the resources needed to complete the required reconstruction tasks. Our picture of a reconstruction application is one in which many coupled, but replaceable, processing components called modules work together to reconstruct the event. The modules are coupled in the sense that a module that matches tracks found in the drift chamber to energy deposits in the calorimeter requires other modules to have already found the tracks and energy clusters. Modules are replaceable in the sense that the matcher–module does not care exactly how these tracks and clusters were found, so the tracking and clustering modules may be chosen as desired by the physicist from a pool of available modules.

Going into more detail, modules obtain their data from a central event and put their results back into the event. The term "data" here really means C++ objects. Thus, a module to do cluster finding in the Silicon Vertex Tracker might obtain objects representing silicon strips from the event, manufacture clusters objects, and place them in the event. Later, another module can apply alignments to the clusters to yield SvtHit objects required by a track finder.

Reconstruction modules need more than just access to the data. They also need access to time dependent data like calibrations and alignments, and they need a control structure that defines their execution order and then calls them according to that definition. We apply several constraints on the architecture in these regards. First, modules are never to decide if data of any kind, especially calibrations, are valid. They simply request the data from the architecture and use them. Second, it must be possible to selectively turn modules off and on prior to running, and it must be possible to rearrange the running order without recoding, relinking, or even restarting the application. These restrictions are intended to ensure that a module is purely an algorithm, with control information localized elsewhere.

This model is very comfortable for physicists. They have been using it for years. Unfortunately, it is difficult to implement this in a way that allows good Object Oriented design of the system. This is explained below and our solutions to the problems are presented.

2 Control Interface: The Application Framework

The control interface allows a user to specify which modules are to be active during a run. It is then responsible for calling the modules in an order given by the user. The software that provides these features in BaBar is called the Application Framework, and is an outgrowth of CDF's Analysis Control program by David Quarrie, Elizabeth Sexton–Kennedy, and Marjorie Shapiro.

A module is implemented as a concrete derivative of an abstract base class, AppModule. The AppModule interface provides a collection of methods that the application framework calls to tell a module to do whatever processing is needed at some particular stage in the execution of the application. A sketch of the AppModule interface is as follows:

```
class AppModule {
   void   begin( AppJob* job ) = 0;
   void   end  ( AppJob* job ) = 0;
   Event* event( Event*  ev  ) = 0;
};
```

A reconstruction application is constructed by defining a class called AppUserBuild, compiling it, and then linking against the appropriate libraries. AppUserBuild simply instantiates all of the modules that a user may wish to use. Note that a different AppUserBuild is written for each application.

When an application starts, AppUserBuild is instantiated, thereby registering available modules with the framework. Next, user commands[3] define an execution path, specifying which modules are to run, and in what order. Other commands start the application running. The framework then calls the modules' begin(AppJob*) methods, indicating that job setup should be done, and subsequently enters an event loop, calling the event(Event*) methods. Finally, end(AppJob*) is called to finish the run. The actual interface includes more than just begin, end, and event, but we omit them here for brevity. For the sake of space, we also do not discuss finer levels of flow control, like subsequences, but simply mention their existence.

This design appears to be a hybrid between traditional SA/SD[4] design techniques and Object Oriented techniques. The event seems to be like the central data–structure and the modules seem like the functions in SA/SD. Even though the modules exchange C++ objects through the event, the modules appear to all depend on the event and therefore, as in SA/SD, any addition to the list of data objects, or modification of existing data objects, will cause the event to change, and thereby cause all modules to recompile.

To prevent this, our event is designed to be an object dictionary. It has no dependence on the objects it contains, not in terms of quantity, kind, or actual properties/interfaces. Consequently, encapsulation still works: addition of a new module that requires new data objects does not cause perturbations that propagate out to all other modules, nor does changing the interface of any of the objects stored in the event. Meanwhile, the old, familiar HEP processing paradigm is maintained. We turn now to the design of the event.

[3] Presently, this is via a Tcl interface.

[4] Structured analysis/Structured Design

3 Event Design

The event design has three constraints:

(i) The event must not depend upon its data, either in number or kind.
(ii) The event must provide a strongly typed interface.
(iii) The event is to encapsulate the data model, hiding any hierarchical structure it might have. [5]

We satisfy these requirements with something called an Interface Dictionary. An interface dictionary is like any other dictionary in that one client stores data with a put(key, data) method which causes the item "data" to be stored and associated with the key. Later, another client uses the same key to retrieve the data with a data = dict.get(key) message. An interface dictionary does this with heterogeneous data, however, while providing a strongly–typed interface, as shown here:

```
Track   *t;
t = Ifd< Track >::get( event );

Cluster *c;
c = Ifd< Cluster >::get( event );
```

In this example, two different classes, Track and Cluster, are being retrieved from the event. The event itself is the interface dictionary. The class Ifd is a helper class that serves as a front end, responsible for the type safety.

Figure 1 shows a dynamic diagram that illustrates the operation of get(). A client calls the get(event) method of the template class Ifd<T>. Ifd<T> maps its type, T, onto a key that is used when invoking the get(key) method in the interface dictionary. The dictionary does a lookup on itself to locate the data. Actually, there is a little twist here, in that the dictionary does not store the data itself; rather, it stores a get–proxy that is located via the lookup. The get() message is forwarded from the dictionary to the proxy, and the proxy returns the data, which is then returned to Ifd<T>. Ifd<T> returns it to the user as a T. This means, of course, that the put method receives not a key and data as arguments, but a key and a get–proxy.

It is vital to note that the dictionary key has a private constructor, and Ifd<T> is the only agent in the system that can create one. Because of this, Ifd<T> is able to enforce a policy that ensures correct manipulation of the types. This is only possible because the compiler has handled the

[5] Otherwise, modules must traverse the structure to obtain data. Changes in the model would then require rewriting all modules.

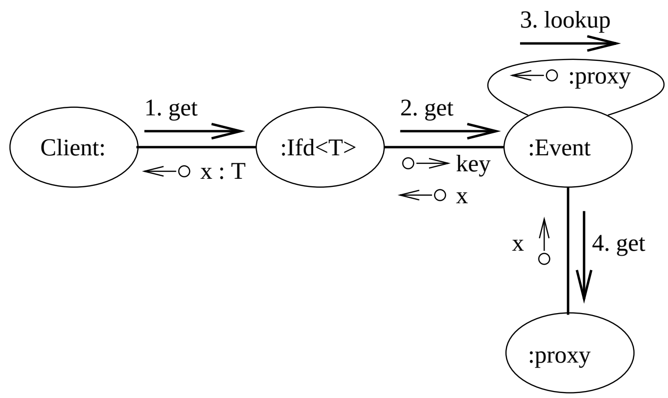

Fig. 1. Dynamic diagram of the get operation on an interface dictionary.

real type issues by virtue of the templated class and method signatures, e.g., T* Ifd<T>::get(Event*). As an aside, we mention here that the actual signature of get() and put() includes a secondary key that may be used to distinguish multiple dictionary entries of the same type.

The proxy is present in the design in order to provide polymorphism. If this polymorphism was provided in the event class itself, the class would couple to the stored data, which is not allowed. The proxy is the agent in the system that will have detailed knowledge of the data model and I/O mechanism. The proxy is completely hidden from the client issuing the get().

The net result of this design is a type–safe dictionary that completely hides the data model and access details. To use this system, an Input Module must be written. This module is specific to the data model and is placed at the top of the execution path. It is responsible for loading the event with get–proxies. Note that the data model has been encapsulated within the input module, which might be selectable at run–time.

4 Access to Time Dependent Data: The Environment

It is wasteful to have every module check the validity of calibration data prior to using it. Doing so also hinders encapsulation of the calibration procedure. We therefore add another entity to the architecture, called the environment, and relegate validation to it.

The environment can be implemented as an interface dictionary. It provides access to calibrations using semantics identical to that of the event, i.e., modules use Ifd<T>::get(Environment) to access objects which they are then to assume are valid, and which they are prohibited from caching between events. We now describe how validation is accomplished.

The time validation is accomplished with the help of a ConditionsModule and a set of proxies. The proxies have several properties: (1) They do not attempt to read their data until a get() is actually done on them, but (2) they cache the results thereafter. (3) They know how to check the validity of the cached data, but normally do not do so. Nonetheless, (4) they can be told to check the validity of the cache when the *next* get() is attempted.

The ConditionsModule is placed after the InputModule. At begin(AppJob* job), the ConditionsModule creates all the proxies for all the various calibrations that might be requested and places them into the environment. At each event(Event*) call from the framework, the ConditionsModule tells the environment to put all of its proxies into a "checkValidity" state. Since they do not check the cache validity at this time, this is an inexpensive operation.

Whenever a module accesses the environment during event processing, e.g., Ifd<List<EmcGains>>::get(environment), three possible states and responses are possible: (1) If this is the first request *ever* for the data, then the data item is built. (2) If this is the first request while processing *this particular event*, then the checkValidity flag is honored and the cache validity is checked. If needed, the data are rebuilt. In any case, clear the checkValidity flag. (3) In all other cases, return the cached data immediately. No validity test is needed.

The consequence of this design is that no module ever needs to worry about the validity of its data, yet validity tests are never done more than once. No calibration data are read, unless it is actually used. Notice that the algorithm lives outside the reconstruction modules and also outside the environment, i.e., it resides in the behaviors of the proxies. Thus, it is fully encapsulated.

5 Remaining Services of the Architecture

There are a few more things that the architecture must provide to meet the needs of the reconstruction modules, and we enumerate them here. Modules report error status by writing text to a C++ stream. Particle data, like masses and lifetimes, are available from a package [6] that is a lookup table for these data. A common system of units is not enforced with code: it is handled via a paper specification and project management. Histograms are provided by HepTuple from CLHEP. Relationships between objects, e.g., recording which tracks HitOnTrack objects are associated with, are recorded with C++ pointers, although this may change when we begin to use an object oriented database.

[6] PDT, Luca Lista. INFN Sezione di Napoli, Naples, Italy

Modules that need execution parameters, e.g., road widths or other cut values, instantiate a derivative of class AbsParm and own this object. For example, there are derivatives that provide bool, string, double, long, and filename parameters. The module simply assumes that an external agent will set the values of these parameters, and accesses them via a value() method that returns the appropriate type. The framework is the agent responsible for setting the values, and does so prior to the event loop.

Some modules will need information about the geometrical location of detector elements and their material properties. This is handled by a package called DetectorModel. Clients look up detector elements in a DetectorSet object by asking for a list of elements that a trajectory in space crosses. The returned objects correspond to physical entities in the detector, e.g., silicon wafers. These can be queried for geometrical properties and the result will include any needed alignment corrections, thus hiding alignment procedures from the client. It is also possible to ask how a particle on this trajectory would interact with material, e.g., energy loss and scattering data can be obtained. Note that the interface only provides quantities immediately useful to tracking and not things like material compositions. Therefore, the material model and interaction model are fully encapsulated.

6 Conclusions

We have presented the architecture of a BaBar reconstruction application. In doing so, we have shown how we are attempting to support the traditional event based HEP processing paradigm while avoiding the broad software interdependencies that such centralized data structures normally produce. We follow an iterative design/test/analyze cycle in BaBar. The design presented here is under development and will almost certainly change with further experience. Our purpose here is not to present a final, tested, system, but rather to share the lessons we have learned so far.

7 Acknowledgments

Stephen Gowdy [7] designed the internals of the environment and David N. Brown [8] designed the DetectorModel package. One of us (EDF) is supported under DOE contract DE–FG02–95ER40893 and is also grateful to Harvard University/HEPL for their generous support during this work.

[7] Dept. of Physics, Univ. of Edinburgh, Edinburgh EH9 3JZ, Scotland, UK
[8] LBNL, Berkeley CA, USA