DØ Offline Reconstruction and Analysis Control Framework

J.Kowalkowski[1], *H.Greenlee*[1], *Q.Li*[1], *S.Protopopescu*[2],*G.Watts*[3], *V.White*[1], *J.Yu*[1]

[1] Fermi National Accelerator Laboratory, PO Box 500, Batavia, Illinois 60510, USA
[2] Brookhaven National Laboratory, Upton, NY, USA
[3] University of Washington, Seattle, Washington, USA

Abstract

Offline analysis and reconstruction programs are controlled and configured using the DØ † offline framework package. This package incorporates many object-oriented techniques and patterns to make it highly extensible in terms of algorithms and operational characteristics with very low coupling to outside packages. Activities and algorithms are run-time configurable without recompilation. Several subsystems work together to provide the execution environment: a dynamic algorithm registration system, grouping and sequencing components for state machine-like behavior, a user extendible, dynamic message distribution system, and a high level control interface. A set of interactive components that make use of multithreading are being developed along with a CORBA ORB for distributed operation, mixed language use, and support of other services such as database interactions.

Keywords: DØ,framework

1 Introduction

The event [1] processing and control framework[1] at DØ is a C++ application that provides a common interface for analysis and reconstruction algorithms. A standard interface allows all algorithms to be configured in a common way. A standard set of event handling interfaces allows the framework to control and coordinate the flow of collider data and other events through the algorithms. Selection, ordering, and configuration of the algorithms for a particular job are all completed at run time. The framework handles user communications with the algorithms using several interactive components. This paper will point out several important and interesting C++ object-oriented designs that were used in the creation of this system. These techniques should be considered when developing a new framework.

Infrastructure The framework is one of many infrastructure components from DØ. The ***Event*** data is stored in an *Event* [2] object, it is managed by a the EDM [2] package. The EDM ***Event*** object is used directly by the algorithms as the source of the ***Event*** data. Algorithm configuration parameters persist outside the algorithms objects; they are the persistent part or overall state of the algorithm. The parameters are manipulated and made persistent using a package called RCP [3] (Run Configuration Parameters). The RCP package tags each unique set of algorithm parameters, every time a framework job is run. The framework is directly coupled to the RCP package; it relies on the RCP package to identify algorithms that need to be run and the configuration of the

† Multipurpose high energy proton/antiproton collider detector at Fermilab,USA

[1] In this document, an event is normally triggered by the existence or occurrence of an object. The event that is the data acquired from the detector, will be shown as ***Event***.

[2] Concepts that have real C++ class associated with them will be shown in italics

algorithms. Using the RCP package allows one to recreate the exact conditions that were used to run a job.

2 Goals

This system allows for growth and expansion as requirements change during the development cycle without impacting the already existing reconstruction and analysis algorithms. This system has extremely low coupling [4] to the rest of the system, especially the algorithms, can survive in the farm, analysis, and the detector monitoring [5] environment, can handle asynchronous event types [3] , such as run number changes, and synchronous events such as process collider ***Event*** data. A simple example of growth would be the addition of new event objects. Frameworks are sometimes designed to manage a fixed set of events, such as collider ***Event*** data, and run number changes. This framework has no fixed event types; applications are free to introduce new types at any time without impact to existing algorithms. Other goals include:

- **New algorithms** can be introduced on the fly, with no changes to framework.
- **Strong Typing** of event objects. Algorithms are presented with the objects in the exact types they manipulate. The strong typing of C++ is used as much as possible.
- **GUIs** such as the event display and framework control can execute independently of the main event process loop. The framework is not tied to a particular interpreter or GUI product.
- **Control** of reconstruction activities by users is done at run time. An example is histogram filling; the various histograms kept by algorithms can be filled after the ***Event*** has been reconstructed and before the it is put into its persistent form.
- **User Programs** never require a main routine or any C++ code to be written in order to run a framework executable with a custom configuration. The users can link an executable by selecting the algorithms they want to have included. The dynamic load library version allows for one executable that makes algorithms available at run time.

3 Concepts

Algorithms A framework compliant algorithm is created by deriving a new class from the abstract base class *Package*. The algorithm is presented with an *RCP* object in the constructor call. Each algorithm instance will be assigned a unique name and a parameter set (multiple instances of the same algorithm type are allowed). Current interactive components manipulate algorithms at the *Package* level. Before users create a new algorithm, they must decide what events they want the algorithm to response to. A typical algorithm will respond to ***Events*** (as part of the event loop) and to run number changes. Algorithms become event handlers by inheritance from an appropriate event handling class. Each event handler class is derived from a abstract base class called *Interface*. An event handler is simply a interface class with typically one pure virtual method. The event handling classes perform a simple form of double dispatch [6]. Examples of event handling classes are *Analyze*, *Process*, and *RunInit*.

Interface Points An event is really a pair, consists of action ID an object. It is identified by an action ID and is always accompanied by an object. The framework does not know the type of the object; it is application specific and is defined outside the framework library. Event handling classes self register at run time and are assigned an action ID by the framework. Event handlers

[3] An event type constitutes a state, states or event types in the framework are referred to as *Actions*.

use C++ templates [7] to present objects as the type that the algorithm expects to manipulate. This feature allows the framework to pass around objects that it has no relationship with.

Event handlers are similar to states in that they define a time in which processing occurs and appear in two flavors [8]: synchronous and asynchronous. The synchronous type get processed automatically in a sequence within the event loop and must all have accept the same type of object, (i.e. an EDM ***Event***); the asynchronous ones accept a specific type of object and can be triggered at any time by the active algorithms in the framework.

Factory A pluggable factory [9] [10] is used for creating algorithm instances at run time using string names. There is a simple procedure that users must follow when a new algorithm is introduced that will cause the factory to be populated automatically at run time. The factory has no coupling to the algorithms that it creates. The names in the framework configuration file read in at run time refer to registered algorithm types.

4 Event Processing

Actions and WorkQueues Each event handler class is identified by a string or state name. It automatically registers itself at run time into a global action table and is assigned an action ID. The string name to action ID mapping allows event handlers to be assigned to processing states at run time by name. The place or time that reconstruction tasks occur is be determined at run time.

The framework processes information using an event queue (*WorkQueue*). Algorithms that implement special event handlers add events to a queue. The event loop in the framework continually requests that events be added to a *WorkQueue* and processed. The program terminates when the queue is empty.

Data Managers A customized version of the observer pattern [11] [10] is used to organize actions and event handlers. The primary purpose of the *DataManager* is to propagate objects like the ***Event*** through event handlers using this observer-like design pattern. As the system initializes, event handler instances are created and registered by name into *DataManagers*. The event handlers can be thought of as observers of a particular type of event. The *DataManager* object is a collection of observers of a particular action ID.

Controllers The *Controller* holds *Packages* instances, *DataManagers* and *WorkQueues*. It manages the state machine and controls the sequencing of events through its *DataManagers* and its *WorkQueue*. Algorithm instances live inside *Controllers*. A lookup table maps action IDs to *DataManager* instances. A very simple loop is performed here: for each entry in the *WorkQueue*, use the action ID to locate a *DataManager* that will be used to process the queued object. A flow in the DØ framework is defined as a fixed set of transitions that a particular event must go through. The event loop is a list of *DataManagers* that the ***Event*** passes through (or groups of algorithm event handlers). The *Controller* has a flow pattern that can be altered at run time [1], an example is $Generate \rightarrow Decide \rightarrow Filter \rightarrow Process \rightarrow Output$. These types are the synchronous event handler names; they always occur for each event in the order specified by the flow. The *Decide* event handlers can trigger asynchronous events by adding them to the work queue (e.g. Run change objects). Since everything in the *DataManager* and *Controller* is governed by string names and action IDs, new event handlers objects can be introduced easily to the system. The framework knows nothing about the actual event handlers and objects being passed around,

The *Controller* is derived from the *Package* class, therefore it is configured using RCP files. It is configured by a list of algorithm type names, instance names, and associated RCP

file names. Algorithm event handlers are registered in the order that they appear in the list. The *Controller* implements the framework interface *Interface*. Since interface instances are registered in *DataManagers* using the state or action names, *Controllers* can appear to the system as any other event handler. By default it appears as *Process*. *Controllers* can be placed inside other *Controllers* to an arbitrary depth. This feature allows a series of complex tasks to be grouped together - the tasks may involve running many algorithms in a particular sequence.

5 Interactive Components

The interactive abstractions can be used to inspect the results of algorithms and to control the processing and flow of events through the framework. This is essential for debugging and tuning the algorithms, along with viewing intermediate results. The interactive components typically live in a different thread from the event loop to prevent the GUI from freezing while reconstruction activities are in progress. The level of thread synchronization is the event handler. This model allows the developers to create algorithms and not worry about thread safety issues. The framework goes so far as to abstract the locks [12], semaphores, and thread classes. There exists a *Framework* class that contains the event loop and the Controller instances. The interactive components reply on its methods for controls such as pausing/resuming processing, and configuring algorithms.

Commanders and Messengers The *Commander* is a simple abstraction that gives the user access to the *Framework* instance. A derived class is where a command interpreter or GUI would live. Living under a separate thread, the GUI can have its own event loop. Currently a couple of *Commanders* are defined: a simple command reader, a root GUI [13], and an ORB. The ORB extends the framework controls to clients GUIs written in languages such as Java.

The *Messenger* abstraction allows the interactive component developer to catch output from the algorithms. The algorithms do not know about interactive components, so their output goes to cout/cerr. The utilities that surround this component redirect cout/cerr. A few implementations of this interface exist: a console printer and one that uses the ORB to distribute output to all registered clients.

ORB With the ORB[14] *Commander*, the program runs as a daemon and advertises services using the CORBA name server facilities. Client programs attach to the ORB and register output handlers to receive information from the algorithms. Clients can directly control the framework from here. It is planned that each algorithm will optionally be able to advertise an interface; this would allow the client GUI to directly manipulate the algorithms through method calls. With a Java applet client remote users will be able to use the web to start jobs on the central servers, then attach to them for control and manipulation. This is an important feature for the detector monitoring programs[5].

Having a distributed distributed framework allows GUI programs such as event displays to exist in complete isolation from the reconstruction engine. Algorithm development is hindered by the release cycle and distribution of a product such as Open Inventor when it is built directly into the executable. The ORB allows algorithms to ship event data out to the display executables, which are build using the commercial GUI products, allowing development of GUI programs and reconstruction/analysis programs to take place independently.

GUI programs normally require a large amount of resources - memory and CPU. Consuming these resources on a central analysis server is not a desirable quality. Using the ORB and the distributed framework, GUI resources can potentially be allocated on a user's private machine, or only when the user is monitoring a job's progress.

6 Conclusion

The DØ framework does not force the use of specific external products on the user. This allows it to expand as new GUI and communications libraries become available. Multithreading and use of CORBA allow for distributed use of the framework executables, increasing its flexibility. The algorithms remain independent of GUI and communications technologies. The ability for users to easily add new event handlers and processed objects allows for stable algorithms and interfaces. The flexible run time configuration ability allows the user to simply run jobs, not prepare custom programs and compile them into executables. This reduces the amount of software infrastructure expertise an experimenter will need to know. If a new framework is to be developed, one should consider the elements that are making this framework a success:

- A simple algorithm abstraction and a good abstract factory that make it painless for the user to incorporate a new algorithms (*Package*).
- An extensible event handling and dispatching system (*Interface*).
- Use of templates in the event handlers to take advantage of RTTI and strong typing of C++ (*Process, Filter, Output, etc.*).
- The inclusion of state machine behavior (*Controller, DataManager, Action*).
- A simple to use run-time algorithm configuration library (*RCP*).
- A set of abstractions that allow different control or GUI application to attach operate the framework (*Commander, Messenger*).

References

1 J.Kowalkowski, "Framework Users Guide," `http://home.fnal.gov/~jbk/docs/user_guide.ps`, 1998.

2 M.Paterno, "Event Data Model (EDM) Tutorial," `http://cdspecialproj.fnal.gov/d0/EDMTutorial/welcome.htm`, 1998

3 M.Paterno, "Run Control Parameters at DO," `http://cdspecialproj.fnal.gov/d0/rcp/D0LocalGuide.html`, 1999.

4 J.Lakos, *Large Scale C++ Software Design*, Addison-Wesley, Reading, MA,1996.

5 J.Yu, J.Kowalkowski, "DØ RunII Online Examine Framework Design and Requirements," DØ Note 3578, Unpublished, 1999.

6 B.Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1997.

7 K.Weihe, "Using Templates to Improve C++ Designs," *C++ Report*, 10(2): 14-21, February 1998.

8 J.Kowalkowski, "DØ Framework," `http://home.fnal.gov/~jbk/docs/index.html`, 1998.

9 T.Culp, "Industrial Strength Pluggable Factories," *C++ Report*, 11(9): 21-26, October 1999.

10 E.Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

11 M.Piehler, "Adapting Observer for Event-Driven Design," texitC++ Report, 11(6): 36-42, June 1999.

12 D.Schimdt, "Stategized Locking, Thread-Safe Decorator, and Scoped Locking," *C++ Report*, 11(8): 34-42, September 1999.

13 J.Snow, et.al., "Use of Root in DØ Online Monitoring System," these proceedings.

14 J.Siegel, *CORBA Fundamentals and Programming*, Wiley, New York, NY, 1996.