PATHS: SPECIFYING MULTIPLE JOB OUTPUTS VIA FILTER EXPRESSIONS

C. D. Jones*, Cornell University, Ithaca, NY 14853 USA

Abstract
A common task for a reconstruction/analysis system is to output different sets of events to different permanent data stores (e.g., files). This allows multiple related logical jobs to be grouped into one process and run using the same input data (read from a permanent data store and/or created by an algorithm). In our system, physicists can specify multiple output 'paths', where each path contains a group of filters followed by output 'operations'. The filters are combined using a physicist-specified boolean expression; only if the expression evaluates to true will the output operation be performed for that event. Paths do not explicitly specify the order in which data objects should be created, since our system uses a 'data on demand' mechanism, that causes data to be created the first time the data are requested. Separating the data dependencies from the event selection criteria vastly simplifies the task of creating a path, thereby making the facility more accessible to physicists.

INTRODUCTION

A common feature of a reconstruction/analysis system is the ability to output different sets of events to different permanent data stores (e.g., files). This allows multiple related logical jobs to be grouped into one process and run using the same input data (read from a permanent data store and/or created by an algorithm). In this paper we discuss the CLEO data processing system's [1] solution to this task: paths.

Anatomy of a Data Processing System

Most HEP experiments use data processing systems that subdivide the system into software modules where each module can perform one, or more, of the following tasks:

- **Source**: provides data from persistent media: e.g., read found tracks from a file
- **Sink**: stores data to persistent media: e.g., write fitted tracks to a file
- **Producer**: runs an algorithm to create new data: e.g., fit the tracks that have been found
- **Filter**: selects events for further processing: e.g., reject events with less than three fit tracks
- **Analyzer**: performs an analysis on the selected events: e.g., histogram the momentum of the found tracks

Data Processing

Processing data is typically accomplished via the following algorithm. First, a new event is obtained from a Source. Second, groups of Filters are run to determine if the event passes the selection criteria. If the event fails the criteria, the event is abandoned and the algorithm returns to step one. If the event passes the criteria, it is handed off to the Analyzers and Sinks. Producers must be run before the data they supply is used. This can be accomplished by either explicitly placing Producers in the processing chain so they are called before any other Producer, Filter, Analyzer or Sink that depends on those data, or implicitly by using a 'data on demand' system.

Some data processing jobs require that different events be processed by different modules. The standard example of this is the need to write events into different output files based on the events' characteristics. Therefore a data processing system needs a way to specify how event filters are grouped together to form a logical decision about which Analyzers and/or Sinks will be allowed to process a given event. In this paper we will describe how the CLEO data processing system handles this requirement via its use of 'paths'.

THE CLEO DATA PROCESSING SYSTEM

The simple-to-use mechanism for declaring and using paths in the CLEO data processing system is made possible by several key features of the system.

General Purpose Tools

The CLEO data processing system has been designed to accommodate the various data processing tasks of the experiment: online software trigger, online event monitoring, online event display, offline calibration, reconstruction, detector simulation, analysis and offline event display. A central design goal was to make the system easy to use. To accommodate ease of use and flexibility, we needed to build the system using a small number of types of components (so users would not need to learn very many new concepts) but have those components embody general purpose tasks. We ultimately decomposed the system into four general purpose components: Source, Sink, Producer and Processor. The first three perform the tasks of the same names as described in the Introduction. A Processor is actually a combined Producer, Analyzer and Filter. We decided to combine the three tasks into one software module because when physicists do an analysis they would rather edit one software class than (potentially) three. In addition, it can be conceptually useful to be able to intersperse Analyzers and Filters in the same

*cdj@mail.lepp.cornell.edu

processing sequence. Making them the same class makes it easier to build such a sequence in software.

Data on Demand

One novel feature of the CLEO data processing system is its use of 'data on demand' [2]. When configuring a data processing job, a user does not explicitly state when a Producer's algorithm should be run. Instead, the user just specifies which Producers they want to use in the job. At run-time, when a software module requests a particular type of data the Producer associated with that data type will be called automatically. If multiple Producers in the job can deliver the same data type, we choose the last one selected as being the source for that data. Using 'data on demand' dramatically simplifies the specification of a job since users are not required to know the actual data dependencies between modules. It also aids in run-time performance optimization, since only algorithms whose results are actually needed will be run.

Dynamic Loading

A data processing job is made up of many small software modules combined in a set sequence. When started, the CLEO data processing application knows nothing about the specific software tasks it needs to perform. Instead, the binary code for each software module is dynamically loaded into the application at run time under the direction of the user's command (either interactively or through a script). Multiple instances of the same type of module can be loaded into the system by specifying a unique prefix to add to the module's name. This allows multiple instances of an algorithm, each with different run-time configuration, to be run in the same job.

The benefits of dynamic loading are manifold. First, it decreases the time to reconfigure a job, since there is no need to change makefiles and then relink a large executable. Second, since only the software modules the user actually wants to use in that job are loaded into the job, there is no need for a way to disable and enable software modules. To enable a module, the user just loads one instance into the application and to disable a module, the user just removes it from the application. (A 'remove' is not as flexible as a 'disable', since the state of the module will be lost. However, we have never encountered a problem with this in practice.) This also means that we can enforce an 'if it is loaded, it must be used' rule to avoid any user mistakes that would cause their jobs to run for a long time but fail to do what they want because they forgot to enable a module.

PATH SPECIFICATION

A Path is a sequence of Processors that work as a filter to decide if an event should be passed to a terminating set of Sinks and/or Processors. It is important to note that Producers are not specified in a Path. This is due to the system's use of 'data on demand'. Therefore, the path is purely an event filtering mechanism and is not needed to ensure proper data creation sequencing (e.g., that tracks are found before the job attempts to fit the tracks). This concentration on filtering allows users to only have to specify what only they can answer (i.e., what filters do they want to apply to decide if an event should be processed). The system deduces what is necessary to make the job run properly (i.e., when a piece of data should be created in order to satisfy the first request for that data).

Conceptual Model

The conceptual model of the path system is described below. A job is made up of one or more paths. Each path has two parts: a filter and an operation. A path's filter is a Boolean expression made up of a sequence of Processors combined together using Boolean operators (*and*, *or*, *xor* and *not*). Since it is common for the same Processor Boolean expression to be used in different paths, a user can create a 'named filter' to hold that expression. That 'named filter' can be used in the same place as a Processor's name in the specification of a path's filter or in the creation of an additional 'named filter'. A Processor or 'named filter' that appears multiple times in one or more path specifications is guaranteed to be run at most once per event. A path's operation is the activity the user wants to have happen if the filter succeeds. Functionally an operation is an unordered list of Processors and/or Sinks. When a Processor is used as part of an operation, the decision of the Processor (i.e., whether the event passes or fails) is ignored.

Notation

The syntax to explicitly create a path is

path create *path-name filter-exp* **>>** *operation*

where

filter-exp := [**not**] *fname* [**and**|**or**|**xor** *filter-exp*]
operation := *oname* [*oname*]
fname := *proc-name* | *filter-name*
oname := *proc-name* | *sink-name*

In plain English, the **path create** method takes as its first argument the name of the path being created. The second argument is a filter expression. The filter expression is terminated with the >> symbol. The last arguments are the list of Processors and/or Sinks that make up the operation. The filter expression is made up of Processor and/or filter names joined together by the Boolean operations **and**, **or** and **xor**. If a name is preceded by **not** then the value returned by the named object is inverted. Parenthetical sub-expressions are not supported since sub-expressions can be achieved using 'named filters'.

The syntax to create a named filter is

path filter create *filter-name filter-exp*

The **path filter create** method takes the name of the filter as its first argument and the rest of the arguments for the filter expression, which uses the same syntax as the filter expression in the **path create** method.

Enforced Rules

To avoid common user mistakes, the system enforces the following set of rules. First, any Processor or Sink loaded into a job must be used in a path. This avoids accidentally forgetting to use a Processor. With dynamic loading, there is no need to have unused modules in the job. Second, all Sinks and Processors that appear in a path's operation must only appear once, and only in one path designation. This guarantees that the purpose of a path (which is to make it easy to set up the criteria needed to specify when to apply the operation) is enforced. If items in the operation were allowed to appear elsewhere, it would be difficult to decipher the condition under which that item would be given an event.

Expression Evaluation

The filter expressions are parsed left to right, with the result being an expression graph (see Figure 1). The use of 'filter names' creates sub-graphs in the expression graph. If several consecutive parts of the expression share the same operation (e.g., **and**), only one filter object is created for those parts.

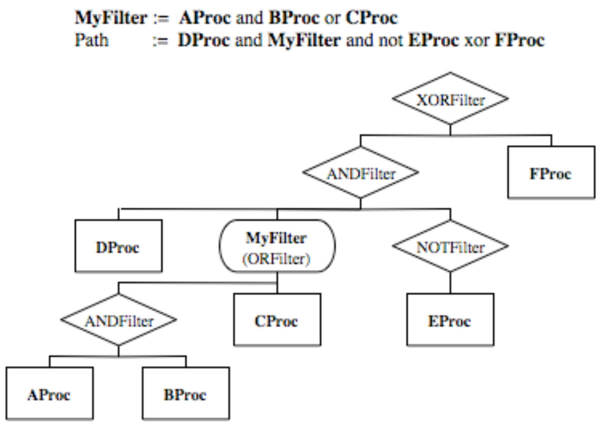
MyFilter := AProc and BProc or CProc
Path := DProc and MyFilter and not EProc xor FProc
XORFilter
ANDFilter
FProc
DProc
MyFilter (ORFilter)
NOTFilter
ANDFilter
CProc
EProc
AProc
BProc

Figure 1: Path example as an expression graph

Simple Path

The simplest case is also the most common: the user wants only one path made up of all the Processors in the job (which is often just one), and only if all the Processors pass the event should that event be written to the Sink. In this case, users are not required to explicitly create a path. Instead, the system implicitly creates a default path where the order of the Processors in the path is the same as the order in which they were loaded into the job. This way, the majority of users never have to learn about paths in order to do their work.

EVENT PROCESSING

During event processing, each path is treated separately. Only if the path's filter expression evaluates to 'true' will the path's operation be processed for that event.

Optimization

If a Processor or 'named filter' appears multiple times (either in one path or in multiple paths) it will only be executed the first time its value is requested and that value will be cached. All subsequent requests will return the cached value.

If the value of a sub-expression (or the full expression) of the filter expression can be determined while evaluating the filter expression from left to right, then the remaining 'right' terms of the sub-expression will not be evaluated (since they can have no impact on the value of the sub-expression). For example, say we have the expression:

AProc and BProc and CProc

If AProc returns false for a particular event, then the entire expression must evaluate to false. Therefore, for this event, BProc and CProc will not be called.

A more complicated example is:

AProc and BProc or CProc

If AProc returns false for a particular event then the value returned by BProc will not affect the result of the expression, but the value of CProc could. Therefore BProc will not be evaluated but CProc will. If AProc returns true then CProc will only be evaluated if BProc returns false. In contrast, different Processors will be run for the equivalent expression:

filter Filter := AProc and BProc
CProc or Filter

if CProc returns true, then the sub-expression *AProc and BProc* will not be evaluated. If CProc returns false then BProc will only be evaluated if AProc returns true.

Error Handling

In the CLEO data processing application, a run time error that happens during event processing throws a C++ exception. If that exception propagates up all the way to the event processing loop, one of two things can happen: the job will end with an error or the job will proceed. The choice depends on how the user configured the job.

If the user has specified that a job should proceed when an uncaught exception occurs, then the path that was being evaluated at the time of the exception will be aborted and the next path in the sequence will be allowed to process that event. It is possible for multiple paths to be aborted because of the same underlying problem (e.g., they all ask for non-existent data). In all cases, the

```
#create the path used to monitor the job
path create monitor >> DataMonitorProc PrintEventNumberProc WatchDiskSpaceProc

#create event selection filters
path filter create qcd EventTypeFilterProc@qcd
path filter create bhabha EventTypeFilterProc@bhabha
path create qcd_data  qcd                  >> qcd.data
path create qcd_index qcd                  >> qcd.index
path create bha_data  bhabha and not qcd   >> bhabha.data

path create bha_index bhabha               >> bhabha.index
```

Figure 2: Reconstruction Job Output Paths Example

aborted paths will be executed for any subsequent events that are processed.

EXAMPLE JOB

A real life example is our use of paths to cluster and index events as the final step of reconstruction. CLEO's new EventStore [3] allows users to access all our events from one simple interface. The data for the events are clustered by physics category, with each event is stored in only one data file. Indexes (which determine what events to process) are allowed to index events that are stored across multiple data files. The data files and index files are created using paths. A simplified version of the path specification for the reconstruction job is shown In Figure 2.
This job creates five paths. The first path has no filter and instead runs three Processors that monitor the reconstruction collation job. The Processor EventTypeFilterProc is used to filter events based on their gross physical characteristics. Multiple instances of this Processor are used where the postfix following the @ sign denotes the type of events the filter will pass. Therefore, the filter named 'qcd' will only pass events that are believed to be caused by qcd processes, while the filter named 'bhabha' will only pass bhabha events. The paths that fill the qcd data and index files use the same filter since both files are meant to hold information about all qcd events. The filters for the paths for filling bhabha data and index files are different. The reason is that any event that passes both the qcd and bhabha selections should have its data stored in the qcd.data file and not in the bhabha.data file. However, such an event should appear in both the qcd.index and bhabha.index files.

CONCLUSION

Event processing control flow is an advanced but common feature of modern data processing systems. The CLEO design leverages CLEO's use of dynamic loading and 'data on demand' to simplify the specification of the control flow. This allows users to concentrate solely on specifying the task they wish to accomplish and therefore makes them more productive.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation.

REFERENCES

[1] C.D. Jones and M. Lohner. *CLEO's user centric data access system. In International Conference on Computing in High-Energy Physics and Nuclear Physics (CHEP 2000), Padova, Italy*, Feb 2000

[2] C.D. Jones, Reconstruction and Analysis on Demand: A Success Story. *2003 Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, Ca, USA*, March 2003

[3] C.D. Jones, EventStore: Managing Event Versioning and Data Partitioning using Legacy Formats. In *International Conference on Computing in High-Energy Physics and Nuclear Physics (CHEP 2004), Interlaken, Switzerland*, Sept 2004