Framework User's Guide

Jim Kowalkowski
Dhiman Chakraborty
Herb Greenlee
Qizhong Li
Gordan Watts

Abstract

The D0 framework provides a common, well-defined, structured way to build software packages for the purpose of event reconstruction. Software packages that adhere to the framework interface can be arbitrarily chained together to perform a desired type of event analysis without rebuilding the framework. The framework allows for a common method of configuring, initializing and managing software packages that conform to the standard interface it provides.

This document contains information for version 2 of the framework. Version 2 RCP files are 100% compatible with version 1. The new features of version two will not be available in version 1 compatibility mode. Running the framework executable from the command prompt is different in version 2 then in version 1. The error messages produced and the options for directing program output have changed significantly.

1 Introduction

The framework is an object-oriented system that provides the user with a common interface and methodology for doing event analysis. Through a set of classes, the framework allows all users to generate application program modules with a common way in which they receive event data and communicate results. The framework defines the way in which event data flows through user application programs. It provides a set of interface points, which can be utilized to register hooks or functions in an application program. The purpose of the framework is to pass event data from user package interface point to user package interface point, in the order and configuration specified by the user at run time.

1.1 Rationale

Having a framework that defines rules governing the creation and use of application program modules allows then to be reused in different configurations. It allows the user to break down problems into small, well-understood pieces that can be reused used by others. By breaking down a problem, testing periods can be shortened, problem and bug analysis times can be shortened, and functionality that is common across difference areas does not need to be rewritten by each area. Having a framework allows users to concentrate on the analysis or programming problem they are trying to solve without getting bogged down in operating system related details such as opening files, producing output, configuring the environment. It allows determining how to arrange program segments with others to do an actual analysis.

1.2 Purpose

The purpose of document is to describe the framework design and all of the major components. It will explain how to use them, including configuring and reuse of existing modules along with creating and configuring new modules. The available processing hook interfaces will be explained and simple sample programs will be shown.

1.3 Definitions

Package: A user-written application program usually designed to perform some kind of reconstruction activity using data in an event. An application program that conforms to the framework package interface definition. An application program that is derived from the framework base class Package.

RCP: Run Control Parameters. RCP is a list or set of parameters that are used to configure a user package. This is a list of parameters that are used to configure components of the framework or the framework application itself.

Interface: A user optionally implements a set of interfaces. An interface defines where a user's code will be executed in the context of the event stream. Many interfaces exist; each specifies a particular place, point, or location within the event stream in which user code must run.

Hook: A term describing an interface.

2 Overview

The framework manages the flow of events through a series of user written and configured software packages in a well-defined manner. The framework provides a standard way to combine application software packages into an event-processing stream. Users write application software packages that adhere to a package interface; this allows the packages to be "plugged" in anywhere in the event processing stream using an RCP file. Inside the application program, the user specifies how, when, and where various sections of code get executed by using the interface classes. A package will query RCP information to allow flexibility for users to control the behavior of the package. There are two basic users of the framework, the first is the person creating an individual package, and the second is the person running the framework using a set of packages, which they have configured. The idea behind separating processing into packages using a well defined interface is simple: put common or complex processing into single components such as packages so they can be reused and tested once for many different applications. The framework itself has no knowledge of an event object that is processed by user packages; it passes event data from package interface to package interface as a pointer to data - or in C++ terms, a void pointer.

2.1 Framework Library User

To a user that is creating event-processing packages, the framework appears as a library. A user will utilize a set of C++ classes to provide reconstruction algorithms and other processing logic. By using the framework library, the user, effectively "advertises" their package automatically for others to use when configuring event-processing streams. Some examples of packages would be an event-reader, an event-writer, and a jet finding algorithm, a simulation module, and an electron finding package.

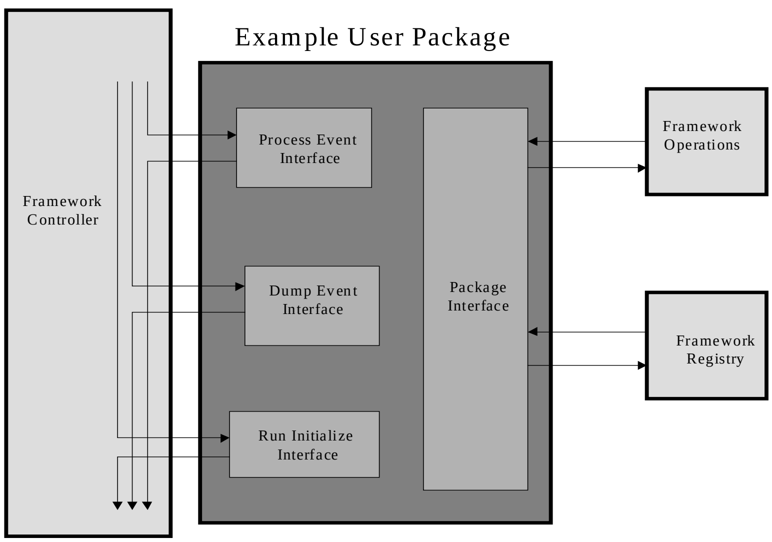

Figure 1

The library user has no knowledge of other packages in the system or where it lives within the execution environment. It conceptually has one path for data to flow into it (the event), and one piece of data it can return (a status code). A package does inform the framework what type of event processing it requires through the use of interfaces. Examples of interfaces would be "call me when the run number changes", or "call me when it's time to process an event", or "call me when the job is complete". Figure 1 shows a simplified view of the system from a user package developer's perspective. In figure 1, the user has chosen to implement three interfaces. In addition, the user package is derived from Package and is registered with the framework.

The only way a user package's processing code gets invoked is though an interface. Each interface defines a particular type of processing that the user wants to do. Each interface will get invoked at a particular place in the event processing stream. It can be said that a package *implements* a set of interfaces.

2.2 Framework Executable Program User

To someone running the framework, it appears as a single executable. The executable takes as arguments an RCP configuration file and in the future, a list of package libraries. The RCP configuration file instructs the framework program as to what packages will be used in the run and the order in which data will flow through the packages, along with all the required configuration information for all the individual packages. The user is expected to create and maintain many framework configuration files. Each set of files would specify a particular type of event analysis using a set of packages. Each package that the user configures can be viewed as a black box with a name. The black box is fed an event at a particular time, which it acts upon, such as adding data to the event based on what the algorithm it implements finds. Figure 2 shows the parts visible to the user when running the framework program. The user creates an RCP file using a standard text editor; it contains the configuration rules, which the framework executable reads in. The use of RCP will be described in a later section. The basic job of RCP is to specify the packages that will be used in the run and the order in which events should pass through then. The user also gives names to each of the package instances they create along with all the parameters required by the package. The framework will also require in the future, a list of shared object libraries that will be used to pull package code out of. This arrangement will allow users to make new packages available to the framework without relinking the framework executable. Currently a framework executable must be built that includes all the potential packages that a user may want to configure. In Figure 2, the user has requested three package instances to be created by the framework: a generator package instance with name GE, a ConeJets package instance with name CJ3, and a second ConeJets package instance with name CJ5. The ConeJets package instances are differentiated by the radius parameter. The parameters allow the user to adjust or change the behavior of the package.

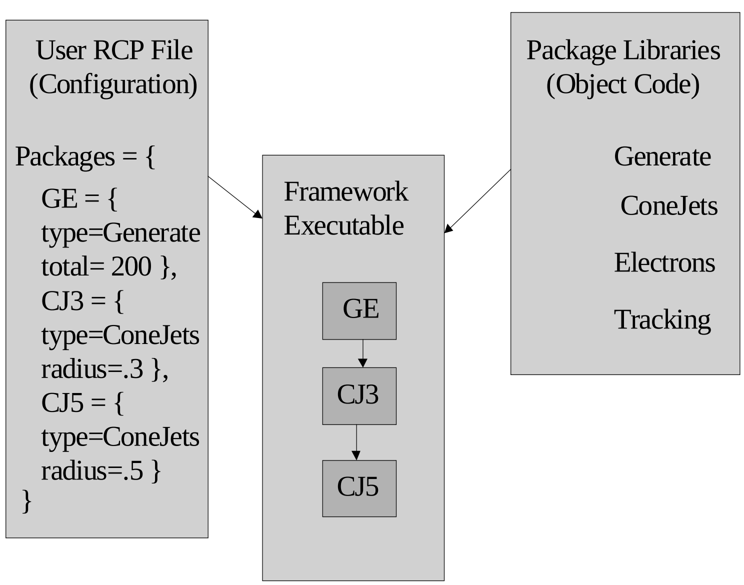

Figure 2

2.3 Building Blocks

The framework allows the user to configure packages in arbitrary ways; linking them together in any configuration via RCP to achieve the desired analysis. To facilitate this, the framework provides a group processing function or controller. Packages instances are always created and exist with a controller. The controller is responsible for passing the event through all the interfaces that user packages have defined. To state another way, each package contained within the controller defines interfaces in order to tell the system *when* it wants to be invoked, the position within the controller defines *where* the interface should be invoked. The controller organizes all the interfaces into a big chain; when the controller receives the event for processing, it just passes the event through all the interfaces in the chain. An important property of controllers is that they can contain child controllers, so processing can be organized in a hierarchical fashion.

The behavior of controllers is a bit complicated because of the framework requirements. There are really two important concepts to remember concerning groups. The first is that package instances defined in an RCP file will exist within a controller and be owned by the controller. Second is that the controller defines the organization of the interfaces implemented by each of the packages within it and event processing is carried out through the interfaces – not the packages. This package/interface relationship can be confusing and leads to unexpected behavior if the user is unaware of the way a controller handles interface processing. The next section describes the controllers is more detail.

The default controller implementation clusters like-interfaces from all the packages it owns. It positions the clusters sequentially according to RCP information defined for the group. The controller passes the event through clusters of like-interfaces. The RCP for a controller or the framework orders processing based on package position within a list. The controller internally orders interfaces within a cluster in the same order that the packages are defined in the RCP file. The interfaces within each cluster are ordered the same as the package ordering defined in the RCP. As mentioned earlier, cluster ordering is accomplished through a different RCP variable – and this is where the confused can occur.

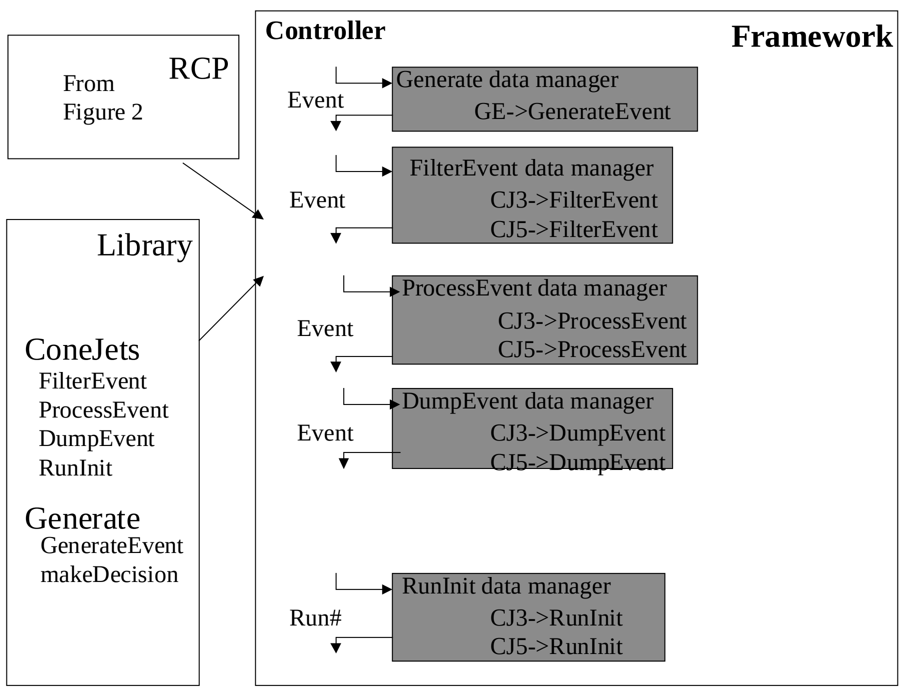

Figure 3

Figure 3 shows the example described in the previous section in more detail. An addition behavior exists within a controller that is very important. Controllers contain controllers and effectively form a tree. In many instances a controller will attempt to put an interface into a cluster, but will fail because it does not understand the interface. The Controller interface clustering is controlled by an RCP variable, that variable probably does not contain and should not contain all the interfaces available in the system – just the ones important to event processing within the group. In the situation that a group fails to add an interface to any cluster it knows about, the group will pass the interface up to it's parent group. The parent group attempts to add the interface to a cluster. The interface will propagate up the tree until it reaches the top. If the interface is not added to any group along the way, the program will immediately stop and report a configuration error. This behavior is necessary to support global interfaces such as job-summary processing and run-initialize processing.

3 Design

The easiest way to understand the framework is to understand the components from which it is composed. The pieces that make up the framework can be grouped into three categories: framework interface components, internal framework components, and utility components. This section will go through a high-level explanation of the components without giving the specific implementation details of the classes.

3.1 Framework Interface Components

The components in this area include the Package class and all the interface classes. These include the classes the user interacts with using inheritance or template instantiation.

3.1.1 Packages

Users must derive a new class from a base class named "Package" for their code to become a framework package. This class identifies the package with a string name, and provides a set of utility functions for the framework to interact with the user package, and for the user package to interact with the framework. The primary job of the package class is to identify the user class as a valid framework piece of software and make it available for use in RCP files. In the future it will also serve as a common way for tools, such as the framework GUI, to extract information from user package instances and reset operating parameters. The list of available packages and tools for creating package instances is managed by the factory object, which is described below. By following the framework package creation method, a user's package will automatically become accessible by the factory object. Each package instance becomes registered automatically with the factory.

3.1.2 Interfaces

The framework comes with a set of interface classes (or hooks) which define different points at which a method in the user's package will get called. A user's package implements a set of interfaces by deriving their package class from them. Currently this is achieved through multiple inheritance. Every user package must be derived from the package class, and optionally derived from a set of interface classes. Each interface defines a protocol by using a C++ pure virtual method. The method defined in the interface class has a particular place in the event processing chain in which it is called. The place that a particular interface is actually invoked is defined by the controller component, which will be described later in this document. When a set of package instances are created by the framework through an RCP configuration file, like-interfaces (of the same interface class) are grouped together and invoked sequentially in the order that the packages appear in the RCP file and according to the defined controller flow. See the Controller description later in this document for further information about flow. The framework does not execute user package code via the package class; it uses the interface object instances. Interface object instances are registered in the proper place in the event processing chain based on their string *name*. The creator of the interface defines the name of an interface; this name will typically be found in the interface header file. The framework libraries comes with a set of available interface classes:

- Generator – Create an event by reading a file or database, or by using a simulation module and determine what processing needs to be done on the event. User implements interface ***Generator*** with name "generator" and writes method ***generateEvent***.

- Decide **-** Look at the current event and make a decision. One such decision could be to do special processing because the run number has changed. The user implements interface **Decision** with the name "decide" and writes methods **makeDecision**.

- Analyze **-** Look at data in the current event without the intention of modifying the data. The user implements interface **Analyze** with name "analyze" and writes method **analyzeEvent**.

- Modify **-** Use the data in the event to create and add more data to the event. The user implements interface **Modify** with name "modify" and writes method **modifyEvent**.

- Process – Perform processing activities on the event. The user implements interface named ***Process*** and with name "process" and writes method ***processEvent***.

- Filter – Check the validity of the current event and determine if further processing should be performed on it. The user implements interface named ***Filter*** with name "filter" and writes method ***filterEvent***.

- Builder – Add data to the current event by reading a data file or database (or simulation). User implements interface named ***Builder*** with name "builder" and writes method ***buildEvent***.

- Dump – Dump information contained in the current event. User implements interface named ***Dump*** with name "dump" and writes method ***dumpEvent***.

- JobSummary – Perform processing at the job end, when there is no more events to process. User implements interface named ***JobSummary*** with name "jobSummary" and writes method ***jobSummary***.

- RunInit – Invoked at the start of a new run number. User implements interface named ***RunInit*** with name "runInit" and writes method ***runInit.***

- RunEnd – Invoked when the run number changes. User implements interface named ***RunEnd*** with name "runEnd" and writes method ***runEnd.***.

It is very easy to add interface classes to the framework without affecting any of the user's packages. All interfaces can be invoked. That is, an event can be passed to then in a common way, and they in turn call a specific method within the user's package. Invoking an interface is an important concept used throughout this document.

All the classes described above are derived from a common base class called Interface, the framework only knows about the abstract base class Interface. All data destine for user package code through the above-described interfaces passes through an instance of Interface.

3.1.3 Registry

The framework registry allows automatic registration of users packages. This facility makes user-written packages available to the framework without coupling the framework library to them. Registration is carried done by creating a source file with a single line of code in it. The single line is a macro provided by the framework using the user's class name as an argument. The macro gives the class a name and a version. The version is provided by the RCS variable $Name. Registration of the package occurs before the main routine is invoked. The instances of the Registry class are recorded in the factory so the factory can create instances of the user packages by name.

3.2 Internal Framework Components

The framework contains several important internal components that are not readily seen by the user. These components define how processing of events is organized and carried out. They are important to understand when configuring a system at run time.

3.2.1 Controller

A *Controller* is a *Package* that implements a simple *Interface*. Within the framework, package instances always exist within controllers. By defining the Controller to be a Package, instances of it can be created and configured by using the RCP package. Interface and Package instances that are implemented by users exist and are executed within the context of a Controller. A controller is invoked just like any other interface. Invoking the controller will typically cause the event to be passed through all of the interface instances registered within that controller. As mentioned in a previous section, each interface class has a name associated with it. The controller has a name associated with it; this name is the name of one of the interface classes and is assigned in the RCP file. Each controller has it's own RCP file.

Implementing the low-level framework Interface class means that the controller must assign a name to the interface. The name given to the interface comes from the controller's RCP file by assigning "InterfaceName". By setting with RCP variable, one can make the controller appear to the system as another interface - in other words, the controller appears to the framework as another, well-known interface such as "process" or "filter". This means that controllers can be inserted into the event-processing stream anywhere the user wants. This implies that processing can be organized in a tree-like fashion, where branches are entered or by-passed based on filter conditions.

The user is also required to set the RCP variable "Interfaces". This variable specifies all the interface names that the controller is willing to register and work with. The highest level controller would typically define the names of all the interfaces that could be implemented by packages. A lower level controller may choose only to act on certain interfaces, such as filter and process, and leave processing such as runInit and runEnd to a higher-level branch in the tree.

Each controller must have a variable named "Flow" defined. This variable tells the controller how event data will flow though it. The value of this variable is a list of interface names that will be participating in the data flow. When a controller is invoked through it's interface, the data passed in will be processed by the packages implementing the first interface in the flow list. If the packages process the data successfully, the data will be passed sequentially through the interfaces described in the "Flow" list.

The fourth important RCP variable for the controller is "Packages". This variable defines a name for each of the package instance that will be created and live within this controller. As the packages for this controller are created, their interfaces will be added to lists - one for each of the names defined in the "Interfaces" variable.

The Controller performs work based on entries in a queue. All Controllers basically run until there is no work on their queue. A work queue entry consists of an interface-id and a chunk of data. Each interface name is assigned an id that is essentially an index into a table of objects (see DataMgr below). An example of a chunk of data is an event. The controller pulls an entry off the queue and gives the chunk of data to the object registered at the index recorded in the queue entry. It is important to understand the rules that drive a Controller:

- The head controller inserts an entry on the queue specifying the first interface in the flow as the receiver and no data (null) until the first-in-flow interface returns end-of-job as a result.

- All controllers loop until there is no work on their queue, at which point they return control to their parent.

- An entry on the queue with no data will only be processed by the first interface registered in that group. It will not be propagated to the other interfaces.

3.2.2 DataMgr

The DataMgr is an internal component of the framework. The DataMgr manages a list of interfaces of a given type. The Controller creates one DataMgr instance for each of the interfaces it knows about. The Controller gives the DataMgr a chunk of data to process. The DataMgr takes the chunk of data and passes through all the interfaces that is holds.

3.2.3 Factory

A package factory exists inside of the framework. By using the Registry, user packages are made available to the factory. Each user package is identified by a string name. The factory can create an instance of a user package given the string name of the package. The framework uses the factory in conjunction with the RCP files to identify and create the package instances that the user wants at run time. In the RCP file, a user will specify the string name of the package. The framework will ask the factory to create a package instance of type "name" as it reads the RCP files. This component is really hidden from the user. The only time a user will be aware of this component is when packages are not available which have been named in an RCP file. The global registry variable also gives some visibility to the existence of this facility.

3.3 Framework Utility Components

There are a few framework library components that are used to trigger some kind of behavior in the framework as it is running. This section describes those components.

3.3.1 Action

The framework contains a single table where all actions are registered. The queue entries in the controller specify an action. The action is really a number or ID that represents an interface name. There is one action ID for each interface named registered by the system. Given any interface name, the action ID associated with it can be returned. Along with the global action registry is a templated class that will manage an action table. Given any object type, the action table will create a look-up table that maps an action ID to an instance of the object type template parameter. This feature is used by the controller to quickly map queue entry actions to objects such as the DataMgr, which knows how to process the data on the queue for that action ID.

3.3.2 Result

Every interface method must return a value indicating success or failure. The Result class encapsulates the return code. The Result can be set to several values which include success, fail, and end-of-job.

3.3.3 RCP

The Run Control Parameter (RCP) library is external to the framework. It is used to configure a framework run. RCP variables go into files; the typical arrangement is one RCP file per package instance. The next section will describe the actual syntax used to configure the framework program. The framework requires each package instance RCP file to indicate the name of the package to be created. All other variables and values in a package instance RCP file are determined by each package.

3.3.4 Arg

Chunks of data are put on the Controller's queue. This chunk of data can be anything the user wants. The Arg set of classes allows any user defined data to be pushed onto the queue and managed. Basically the Arg class is templated on object type and it just hold a pointer to an object of that type. A set of template utility functions creates, manage, and insert the Arg instances onto a queue. The Arg classes also have utilities to allow the real object to be extracted safely by the interface classes. To summarize, the Arg classes allow interfaces to safely pass arbitrary data around inside the framework and safely extract it before giving it to the user. All data passed around in the running framework is wrapped in an Arg<> which is derived from CommonArg. The framework only knows now to really pass CommonArg pointers around the system from package interface to package interface.

4 Miscellaneous Diagrams

Now that all most of the important framework components have been described briefly, a simple diagram will pull them all together. Figure 4 shows the relationship between the various controller parts.

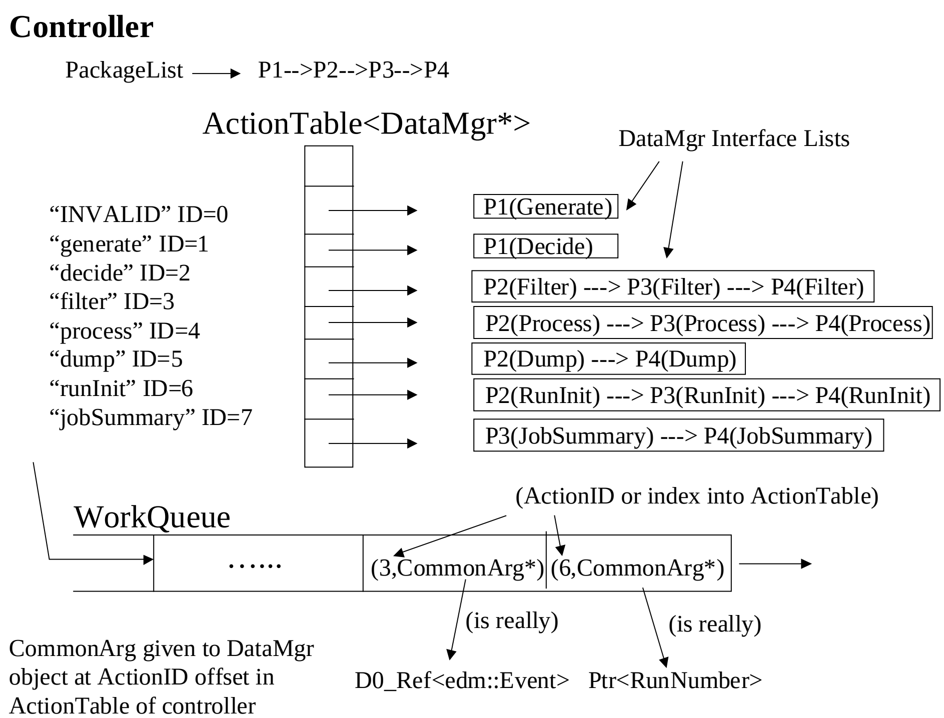

Figure 4

Figure 4 shows a controller that about seven different interfaces. The controller has made one DataMgr for each of the interface types and added the package interfaces to the correct DataMgr. Processing entries off the queue is very simple; just use the ActionID as an index into the ActionTable and give the associated DataMgr the CommonArg pointer. The DataMgr will push the CommonArg through all the interface instances that are register within it. It is the job of the specific type of interface to extract the correct data type out of the CommonArg and give it to the package. The DataMgr will use the controller "Flow" list to determine what to do with the entry on the queue. If there exist a next-in-flow ActionID, then the DataMgr will push the current entry back onto the queue with a new ActionID - the next-in-flow ID. If the DataMgr is the last in a flow, the CommonArg is deleted, which causes the managed pointer class to go out of scope, which may actually delete the object it holds. In figure 4, the queue contains two items, one destine for "filter" and one for "runInit". As one can see, the managed pointer classes are different as well as the type of data that the interface works with. It is up to the interface classes to verify that the data they receive from the framework is really what they expect to get before giving it to the user.

5 Framework Library Use

This section describes the actual C++ classes that the user will need in interact with to create packages and use the framework library.

5.1 The Package

The Package class is an abstract class. All user code must exist in a subclass of Package. To the framework, all user packages are handled as Package objects. Two sets of methods exist in the Package interface. The public methods are meant for the framework to get information about the package instance from the user. The protected methods are meant for the user package to get information from and interact with the framework.

```
class Package {
    Package(Context* hidden_info);
    …
}
```

The user is required to supply a constructor that takes a Context pointer as an argument. The Context class is used in name only. The user is required to pass the pointer down to all base classes of the user package. The context instance passed into the Package contains management information, which the user does not have direct access to, such as the assigned instance name.

```
class Package {
    …
    virtual void reinitialize(const RCP&);
    virtual std::string packageName();
}
```

The method *reinitialize*() can optionally be supplied by the user in the derived package class. This method will be invoked by the framework to inform the user that one or more of the RCP values has changed and the user package needs to reset it's working values.

```
Class Package {
    …
    const RCP& packageRCP();
    const std::string instanceName();
    const std::string packageName();
    …
    const RCP& frameworkRCP();
    ostream& out();
    ostream& out(const std::string& reporting_stream_name);

    ErrorLog& log;
}
```

The method packageName() returns the name as recorded in the Registry. Every user package instance has a fixed RCP object associated with it. The Package method *packageRCP*() allows access to this information. The RCP object returned by this method represents the information from the RCP file for this particular package instance. The method *instanceName*() allows the user package to peek at the actual name of the package instance. The instance name actually comes from the RCP configuration file for the framework, which is read at the beginning of the run. The framework program has an RCP file associated with it, which contains global information concerning defaults and the environment; this information may be of interest to a user package. The framework RCP can be obtained using the *frameworkRCP*() method. A user package can produce any type of printable reporting information using the *out*() and *log*() methods. The out method comes in two forms, the ones with no parameters which send output a default location, and ones that take a string argument, which directs output to a particular destination. Output stream names are assigned in the framework RCP file; each is associated with a particular file. The *out*() method will produce the exact message that the user send it. The *log* will surround the message with administrative information useful to a system-side logger, such as time stamp, node name, and user ID. The implementation of log is designed so that it can be hooked directly into a system logging facility.

It is important to remember that every user package instance created by the framework has three name methods associated with it. The instance name generated at run-time from an RCP file (*instanceName*) would be one. The package-type name accessible to the framework factory (*packageName*) used to create instances of user packages. The package-type name is available to any object in the system given a pointer to a Package.

5.2 Registration

The new system supports two macros used for registering user packages:

```
FWK_REGISTRY( class_name, variable_name )
FWK_PACKAGE( class_name, "package_name", variable_name)
```

The FWK_REGISTRY macros first argument is the user package class. The FWK_PACKAGE can be used to give the class a specific name. The name is the string that appears in RCP files in the "string PackageName = name" variable entry. When using the FWK_REGISTRY macro, the name in the RCP file is the name of the class. Following is an example for a user package with name EMreco.

- Using the FWK_REGISTRY macro:

```
RegEMreco.cc:
      #include "framework/Registry.hpp"
      #include "EMreco/EMreco.hpp"
      using namespace emreco;
      FWK_REGISTRY(EMreco, dummy)

EMreco.rcp:
      string PackageName = Emreco
      ...
```

- Using the FWK_PACKAGE macro:

```
RegEMreco.cc:
      #include "framework/Registry.hpp"
      #include "Emreco/EMreco.hpp"
      FWK_PACKAGE(emreco::EMreco, "MyEMreco", dummy)

EMreco.rcp:
      string PackageName = MyEMreco
      ...
```

If you use the FWK_REGISTRY, do not include the namespace in the macro argument of it will cause problems. Here is an example of how NOT to use the macro. If the namespace must be used, then register the package with the FWK_PACKAGE macro.

```
FWK_REGISTRY(emreco::EMreco, dummy)
```

The standard place to implement these macros is in a file all by itself. The examples above represent a typical package registration. Put the FWK_REGISTRY macro into a file that prepends "Reg" in front of the real class header file name. The use of these macros causes the framework to automatically register a user package with the framework factory using global constructors.

5.3 Commonly Used Interfaces

As mentioned in an earlier section, a user package must be derived from interface classes in order to be called or processed during the event processing cycle. All interface classes are abstract classes and force the user to implement their methods. Most of them only provide one method specific to the type of processing they are meant to perform. A typical user package will want to implement the Process, RunInit, and JobSummary interfaces. Remember, the framework at a precise location in the event processing cycle, will invoke each type of interface that is implemented by the package.

```
class ANY_INTERFACE : public Interface {
     …
     ANY_INTERFACE(Context* hidden_info);

     virtual Result PROCESSING_FUNCTION(Event&) = 0;
}
```

Above is an example of how all Interface class constructors look. All of them take as an argument the Context pointer. This is the same Context pointer used of the Package class. The method PROCESSING_FUNCTION() is an example of a typical method found in an Interface class. As shown, most, if not all, Interface class methods return an instance of class Result to indicate success or failure. Many of the standard Interface classes will have as an argument as the current event.

```
class Processor : public Interface {
     …
     virtual Result processEvent(Event&) = 0;
}
```

The Processor is the main way for a user package to process an event and add information to it. A user package must implement the *processEvent*() method. This interface will get invoked in the order it is specified in the RCP file for the controller in which it belongs. It gets invoked after all the filters in the group and before all the dump event methods. The scope of this interface is a group. Whether or not the *processEvent*() method of a package gets invoked depends on the group in which it belongs. The filterEvent interfaces that are run before processEvent within the controller must have all succeeded, along with all the builder interfaces.

```
class Filter : public Interface {
     …
     virtual Result filterEvent(const Event&) = 0;
}
```

The Filter Interface is implemented by a user class is order to check out an event before actual processing occurs on the event. Returning failure indicates to the framework that the event has failed the test and no further processing should occur on this event in the current controller. The purpose of this interface is to determine if processing should be done on an event. Its scope is typically a controller. It will only impact processing within a single controller. It typically occurs in the processing cycle after buildEvent (which is at the top) and before processEvent. Again, the main purpose of it is to short-circuit or stop the entire processEvent stage within a group. Returning "fail" from filterEvent will cause the parent controller to skip to the end and return control to its parent. Returning success indicates that processing should continue normally.

```
class Dump : public Interface {
     …
     virtual Result dumpEvent(const Event&) = 0;
}
```

The Dump interface is implemented by a user package to dump out a report about the current event when called. The report is meant to be printable information sent to an output stream. This interface is invoked within a group. It is typically called after processEvent. It is typically the last interface to be called within a group processing cycle.

```
class RunInit : public Interface {
     …
     virtual Result runInit(const RunNumber&) = 0;
}
```

The RunInit interface is implemented by a user package so that the user package is invoked at the start of a new run number. A RunNumber number object will be present as an argument when it is called. This interface has global scope. The highest level controller in the framework typically handles it. All packages in the system will have their RunInit interface registered at the highest level so the framework will run all RunInit interfaces when run numbers change. The running of this interface is typically not affected by the presence of multiple controllers in a running system. All RunInit interface instances from all the package instances are gathered into one place so they are handled correctly.

```
class RunEnd : public Interface {
     …
     virtual Result runEnd(const RunNumber&) = 0;
}
```

The RunEnd interface is implemented by a user package if processing is required at the end of a run, before the new run is initialized. Once again, a RunNumber object will be present. This interface has global scope. The highest level controller in the framework typically handles it. All packages in the system will have their RunEnd interface registered at the highest level so the framework will run all RunEnd interfaces when run numbers change or a job completes. The running of this interface is typically not affected by the presence of multiple controllers in a running system. All RunEnd interface instances are gathered into one place so they are handled correctly.

```
class JobSummary : public Interface {
     …
     virtual Result jobSummary() = 0;
}
```

The JobSummary interface is implemented by a user package if processing is required at the end of the entire job. This interface will inform the package that the entire job is done. This interface typically has global scope; the instance of it is invoked at the highest level controller in the framework. Typically the all of the framework packages jobSummary interfaces will be registered by the highest controller in the framework, so they get invoked at the right time.

```
class Builder : public Interface {
     …
     virtual Result buildEvent(Event&) = 0;
}
```

A user package will implement the Builder interface if the user package is designed to add data to an existing event at some time during the event processing cycle. This interface class is used to request processing during the event *builder* phase. This phase is usually first in a processing cycle. The buildEvent method is used to add data to the event, perhaps by reading a data file and merging the data from the file into the event that came in on the argument to the method.

5.4 Advanced Interfaces

An advanced interface is one that exposes the work queue of a controller. Interfaces in this section are essentially allowed to command the controller to perform actions.

```
class Generator : public Interface {
     …
     virtual Result generateEvent(WorkQueue&) = 0;
     virtual Result addDataToEvent(Event&) = 0;
     virtual std::string location() = 0;
}
```

The Generator is implemented by any package that can be a source of events (or other data). The framework will invoke *generateEvent*() at the start of the event processing cycle. An instance of a package implementing this interface must be present at the beginning of flow for the head controller. Packages implementing this interface must provide *generateEvent*(), *addDataToEvent*(), and *location*(). A person configuring the framework using an RCP file can place more than one Generator package at the front of the event processing package list. The difference between generateEvent() and addDataToEvent() is important.

- The head controller will call generateEvent() for every generator package registered within it. Each generator will add a new event to the work queue to be processed by later packages.

- If generator packages are used in a sub-controller, then addDataToEvent() will be used. Each registered generator here will get passed the same event - the one at the front of the queue.

The location() method will be called by the framework when as error occurs and it needs to know information such as the current event number or current run number (if that information is known by the generator package. It is the responsibility of the generator package to signal end of job processing. This is typically done by asking the system for the action ID associated with "jobSummary" and pushing an entry on the queue.

```
class Decide : public Interface {
     …
     virtual Result makeDecision(WorkQueue&) = 0;
     virtual Result makeDecision(const Event&, WorkQueue&) = 0;
}
```

The decide interface is used as a place to make decisions about actions that should happen as a result of information in the event. The typical processing in a Decide interface implementation is run number changes. The method makeDecision() is given the current event, if the run number changes, then the decider package can push a RunNumber number object on the work queue with action ID for "runInit". An actual package performing this function would usually push an entry on the queue with the old run number and action ID for "runEnd". If the queue is treated as a stack, and the decide package uses the "pushQueueFront()" functions to add work, then the "runInit" must be pushed first, followed by the "runEnd" entry. The controller allows processes queue entries from the front of the queue, so "runEnd" must be in front of "runInit". The Decide functionality and implementation does not need to be a separate package, it can be part of the generator package, or included as a data member of the generators. The latter is preferred because the decider code can then be reused by all the different generators instead of embedded directly in a particular generator. By integrating the generator and decider, the uses do not need to know about the decider package inside the RCP file, this is important because typically the decider is always the same.

6 Environment

The information contained in this section is associated with an actual running example in the framework release area of SRT. To use the example, go to the SRT framework area and look in the examples directory. There are sample generators and deciders in directory generate. All the example programs use these simple generators. The directory simple contains a very easy framework program that counts. The filter directory contains an example program that executes a branch for odd numbers within an event and a different branch for even numbers. The filter example demonstrates nesting controllers.

6.1 Running the Framework

The long-term goal of the framework is to have dynamically loaded user packages. In this mode, all user packages are installed in a directory that the framework executable knows about. The framework executable will be smart enough to look into this package installation directory for user packages at run time in order to dynamically load the code required to create instances of framework packages. Until all the issues concerning a fully dynamic environment are worked out, the framework executable will need to be built by the user which statically links in all the packages the framework user will need in their analysis. The user package code that is statically linked into a framework executable is not used unless package instances are requested in the RCP file. In order to allow the framework package registry to work correctly with static linking, the user packages must provide the registry code in object code format instead of archive library format to the linker. The object code for user packages will be installed into a release library directory. To use the framework, a user is required to generate an executable, specifying all the package registry object files that need to be included in the executable. The examples illustrate how to do this.

A framework RCP file must be prepared to run the framework program. The framework program takes an RCP file as an argument. As previously stated, the RCP contains the instructions on how to configure an analysis run. The framework RCP file will always contain the following:

```
  string InterfaceName = “process”
  string Interfaces = “generator decide builder filter process dump
runInit runEnd jobSummary”
  string Flow = "generator decide builder filter process dump"
  string Packages = “list of packages names contained within this
controller”
  int DumpPeriod = frequency in event per call
  string OutputStreams = “list of output stream names”
```

In addition to these five variables, there will be one variable per item in the *Packages* and *OutputStreams* lists. A simple example will illustrate the use of Packages and OutputStreams:

```
string InterfaceName = “process”
string Interfaces = “generator decide filter process dump runInit runEnd”
string Flow = "generator decide filter process dump"
string Packages = “createEvents processEvents dumpEvents”
int DumpPeriod = 5
string OutputStreams = “logFile reportFile errorFile”

RCP createEvents = <createEvents.rcp>
RCP processEvents = <processEvents.rcp>
RCP dumpEvents = <dumpEvents.rcp>

string logFile = logFile.txt
string reportFile = reportFile.txt
string errorFile = errorFile.txt
```

In the above example, each of the items in the *Packages* list must refer to an RCP variable. Each RCP variable in turn refers to the file where the RCP information is kept for that package instance. Each of the items in the *OutputStreams* list refers to another variable in the file that identifies an output file name. The above example implies that there will be three RCP files present for the framework to read in, in addition to this framework RCP file.

```
string PackageName = package-type-name-here
```

Each of the package instance RCP files must contain the *PackageName* variable. This variable is required by the framework and identifies the type of package that must be created. A simple example of createEvents.rcp using the previous framework RCP file would be:

```
String PackageName = createEventPackage
int totalEvent = 100
int inputFile = eventFile.dat
```

This example will cause the framework to create a *createEventPackage* instance and give it this RCP information. The *createEventPackage* instance will automatically have the RCP variable *totalEvent* and *inputFile* available in its constructor by calling the *packageRCP*() method. A nested controller RCP file must also contain the PackageName variable. The package name in this case must be "controller".

The latest version of the framework still works with the previous version RCP files. The framework will issue a warning indicating that an old format RCP file is being used. Nested controllers cannot be used in old version compatibility mode.

WARNING: The syntax of RCP files is expected to change soon. This is expected to enhance the readability and use of the framework RCP files.

6.2 Creating An Example Package

Currently framework user packages and executable must be created under SRT. See the framework module in the framework/example/simple directory for a GNUmakefile to creating user packages and executables. In the future this section will contain examples creating user packages within the SRT environment. All of the interface classes mentioned in the previous sections are contained is the header file named edmD0om.hpp under framework/interfaces in the SRT framework packages tree. The header files are documented to help use them correctly. See framework/interfaces/Generic.hpp for documentation concerning the use of interfaces. The Package class header file is also fully documented and is located in the framework header file directory of SRT.

```
// stuff in UserPackage.hpp
#include “framework/Package.hpp” /*(1)*/
#include “framework/interfaces/edmD0om.hpp” /*(2)*/
class UserPackage : public Package, public Processor { /*(3)*/
public:
    UserPackage(Context*); /*(4)*/
    Result processEvent(Event&); /*(5)*/

private:
    int something_useful;
};

// stuff in RegUserPackage.cpp
#include “UserPackage.hpp” /*(6)*/
#include “framework/Registry.hpp” /*(7)*/

FWK_REGISTRY(UserPackage,MyUserPackage) /*(8)*/

// stuff in UserPackage.cpp
#include "UserPackage.hpp" /*(6) */
/*(9)*/
UserPackage::UserPackage(Context* x):Package(x),Processor(x) {
    something_useful=packageRCP().getInt(“SomethingUseful”);
}

/*(10)*/
Result UserPackage::processEvent(Event& e) {
    out() << e.getBlob() * something_useful << endl;
}
```

The above example illustrates creating an extremely simple event-processing package. It shows the requirements that need to be filled in order to be a registered package within the framework.

Notes about lines in UserPackage.hpp:

1. Include the Package base class header file.

2. Include the standard D0 header file that contains interface class definitions that the user package will implement. In this case it is Process.

3. Create a new user package, derived from Package and all the Interfaces that will be implemented.

4. Define a constructor that takes a Context pointer as an argument.

5. Define the interface class method that must be implemented, in this case it is processEvent().

Notes about lines in UserPackage.cpp and RegUserPackage.cpp:

6. Include the UserPackage header file.

7. Include the Registry header file.

8. Use the Registry macro required to register this user package with the factory.

9. Write the constructor. Pass the Context pointer to the base classes Package and Process. Use the packageRCP() method of Package to load member data of the user package from the RCP file.

10. Implement the hook or interface functions. In this case, it is processEvent(). This simple example uses the Package method out() to print information.

6.3 Making an Executable

An example GNUmakefile follows. Important parts are the vpath, which informs gnumake where to find the object files used to build the executable. The force_objects variable lists all the object files that are required to create the framework executable. The object files in force_object will be in the user's build directory and in the SRT library directory. The object file framework.o will always be required, it is the main framework routine. In this example, it is assumed that an event generator package is available in libdir or in the current directory. The variable libs lists all the libraries required to create the framework executable, in this minimal example, only the bare minimal are required – the framework library, and the rcp and identifiers libraries. Typically the user GNUmakefile will have the edm, d0om, and other libraries. This GNUmakefile is also set up to build a library named libfexample.a in the usual SRT fashion, skipping over the object files that are used to directly build the framework executable. For a real, working example, see framework/example/simple and io_packages/test/genTest.

```
LIB := libfexample.a
BINS = fexample

# Top level package file
# This is the registration source file with FWK_REGISTRY in it
# These files will be installed into the lib area as object files
OBJCPPFILES := RegProcessor.cpp

# Source files
LIBCPPFILES := $(filter-out $(OBJCPPFILES), $(wildcard *.cpp))

# be sure to include all other .o for packages you will use
# see the framework/example/simple area for better information
OBJS = framework.o RegProcessor.o

# Tell gmake where to find link objects
vpath %.o $(libdir) $(BFDIST)/releases/$(BFCURRENT)/lib/$(BFARCH)

# Executable - code for packages in library gLib and pLib
# be sure to include all libraries required by the various packages
FLIBS = -lgLib -lpLib -lframework -lrcp -lidentifiers -lErrorLogger \
	-lZMtools -lZMutility -lExceptions -lm

$(bindir)/$(BINS): $(OBJS) $(FLIBS)
	/bin/rm -f $@
	$(CXX) -o $@ $(CXXFLAGS) $(CPPFLAGS) $(LDFLAGS) \
		$(filter-out %.a,$^) $(FLIBS)

include SoftRelTools/standard.mk
```

7 Advanced Facilities

The framework library allows a user to add functionality to it in the form of interfaces and controller packages. This can be done without modifying the existing framework library or interfering with the release of the framework library.

7.1 Creating New Interfaces

Interfaces are fairly complex from the framework library point of view, but very easy to work with as a user. The basic premise used in the interface design is that the framework will not know anything about the interfaces that are available. The framework also knows nothing about the type of data passed through the interfaces. There is a base class for all interfaces that the framework knows how to manipulate:

```
Class Interface {
Public:
     …
     Interface(Context&);
     virtual Result invoke(Action::Id nextinflow, CommonArg* ca) = 0;
     virtual const std::string interfaceName() const  = 0;

     WorkQueue& getQueue();
     Const RCP& getRCP();
     Const Context& getContext();
Private:
     Void addName(std::string);
}
```

This is a complex interface and is really only required for complex functionality such as the Controller. The framework passed around objects that it knows nothing about, the second argument to invoke() is the data destined for the interface. It is up to the class where invoke() is written to properly extract and cast the data in the CommonArg to the object the user expects. The framework library supplies a set of utilities to make this job easy. The framework also supplies a generic set of interface templates, all of which are derived from interface. This generic set of template classes extract data from CommonArg automatically and deliver to the user though another virtual function call. All the interfaces discussed in this document are actually indirectly derived from Interface. Every commonly used interface that a package will implement has been generalized into the following groups:

- Interfaces that take one hunk of data and can modify that hunk.
- Interfaces that take one hunk of data and can only look at it.
- Interfaces that take no hunks of data.
- Interfaces that need access to low-level facilities such as the work queue.

The last group includes the generator and decider interfaces. The general form of interfaces is a template class with arguments for the hunk-of-data type and a managed pointer type for that hunk. Every interface class has a lower-level general template class associated with it (prefixed by the word "Generic"). In fact, all the commonly used interfaces such as Process, Dump, and RunInit, are typedefs to the generic templated class with parameters that specify the exact types used. In the D0 case the types used are edm::Event for the event and d0_Ref<edm::Event> for the manager. Figure 4 shows the inheritance relationships between all generalized interface components. The interface classes the user appears to derive from are really template instantiations that describe to the framework what type of data the user expects to process. The templated interface classes take arbitrary data from the framework, cast it into the type the user expects, and deliver it via the interfaces virtual function call such as "processEvent(edm::Event&)".

Interface Class Relationships

A=argument type or data hunk type
M=managed pointer type for pointer to A

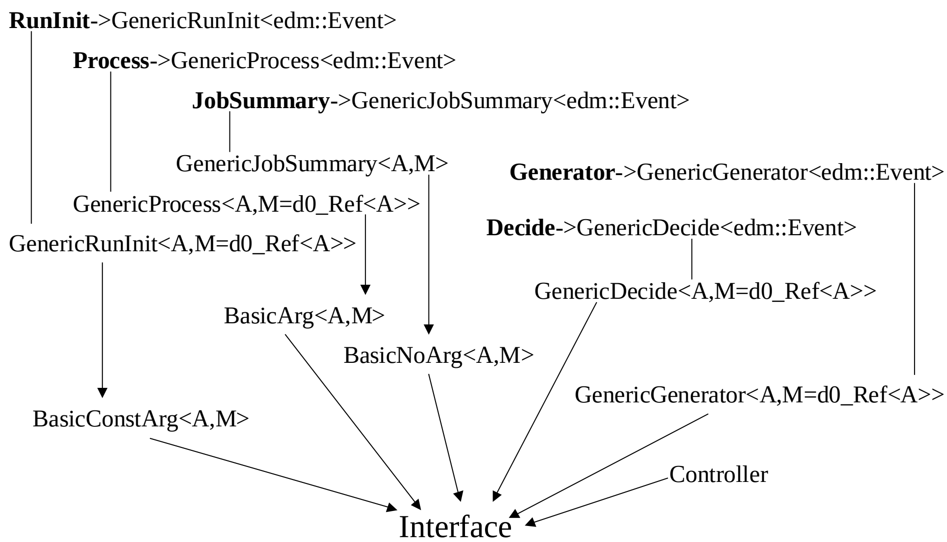

Figure 5

Adding a new interface to the set of interfaces will typically only require a typedef to be generated. Defining a new set of interfaces that process difference event containers or use different managers will typically involving creating a file analogous to edmD0om.hpp. The edmD0om.hpp header file is just a list of typedefs or names for generic template instantiations involving edm::Event and d0_Ref<>. The examples define new interfaces and use different event containers and managers from the standard ones in edmD0om.hpp. Reusing the basic interface templates or the generic template interface should so suffice for most new required interfaces.

This section will be expanded and rewritten as time permits.

7.2 Creating New Controllers

Section to be added at a later time.

8 Testing Utilities

The new version of the framework includes a set of functions that can be used to create a Package outside the context of the framework. To use the test functions, include header file "Testing.hpp". Two functions exist that create user packages

- makePackage(package_pointer, string main_rcp_file, string package_rcp_file)
- makePackage(package_pointer, string package_rcp_file)

The first argument is a pointer variable which will hold the address of the newly created package. The string main_rcp_file is the name of the framework RCP file. Some packages may require environment information from the main RCP file. The string package_rcp_file is the RCP file for the package. These utility functions will initial the RCP system and open the RCP files for the package, create the framework private context, and finally create the user package.

```
Example(string rcp_file)
{
    UserPackage* pkg;
    makePackage(pkg,rcp_file);
    pkg->testMe();
    delete pkg; // you own the pointer!!
}
```

9 Standard Packages

Several standard packages are supplied for use with the D0 framework in SRT package io_packages.

9.1 Available Packages.

The currently available packages and their capabilities are as follows:

- NewEvent.

  Hooks: Generator, Decide, JobSummary.

  The NewEvent package generates a new edm Event. The generated event contains only a collision id. with no chunks. The run number stored in the collision id. is specified in the package rcp file. Event numbers increase sequentially, beginning with one.

- ReadEvent.

  Hooks: Generator, Decide, JobSummary

  The ReadEvent package reads edm events using d0om. The event format (DSPACK or D0MSQL), and the name of the input file(s) are specified in the package rcp file. The input file format may be autosensed by giving an empty string for the input format. Filename indirection via environment variables is supported. Environment variables are preceded by a $ (dollar sign). Multiple input files may be specified using shell style wildcards or file lists. To read a list of input files, specify the file name as "listfile:*filename*," where *filename* is the name of a file containing a list of files to read, one per line.

- WriteEvent.

  Hooks: Process, JobSummary

  The WriteEvent package writes edm events using d0om. The name and format of the output file are specified as rcp parameters.

- DumpEvent.

  Hooks: Dump, RunInit, RunEnd

  The DumpEvent package dumps an edm event to the default framework text output stream. The frequency and number of event dumps are selectable by rcp parameters. Note that the DumpEvent package rcp parameters are separate from the DumpPeriod rcp parameter in the main framework rcp file.

9.2 Linking with the standard packages.

Here is a summary of the object files and libraries needed to link a typical program that includes d0om I/O packages, such as ReadEvent and WriteEvent.

- Include the package object (.o) files from the SRT lib area in the link statement (e.g. ReadEvent.o, WriteEvent.o).

- Include the link objects and libraries needed by d0om and its I/O mechanisms.

  - DSPACK I/O mechanism.

    LoadDSPACK.o -lstream_ds -ld0om_ds –ldspack –lftn

  - D0MSQL I/O mechanism

    LoadD0MSQL.o -ld0omMSQL –lMSQL

  - d0om:

    -lstream –ld0om –lcint-lite –lNameTrans –ld0_util

- Include objects and libraries needed by various standard packages and by the framework.

  framework.o -lio_packages -lframework –ledm -lrcp –lidentifiers -lExceptions –lZMutility.