ATLAS

Athena

The ATLAS Common Framework

Developer Guide

Version: 8
Issue: 0
Edition: 0
Status: Draft
ID: 1
Date: 9 February 2004

DRAFT

European Laboratory for Particle Physics
Laboratoire Europ en pour la Physique des Particules
CH-1211 Gen ve 23 - Suisse

Table of Contents

Chapter 1
Introduction 9
1.1 Purpose of the document 9
1.2 Athena and GAUDI 9
1.2.1 Document organization 10
1.3 Conventions 10
1.3.1 Units 10
1.3.2 Coding Conventions 10
1.3.3 Naming Conventions 10
1.3.4 Conventions of this document 10
1.4 Release Notes 11
1.5 Reporting Problems 11
1.6 User Feedback 11

Chapter 2
The framework architecture 13
2.1 Overview 13
2.2 Why architecture? 13
2.3 Data versus code 15
2.4 Main components 16
2.5 Controlling and Scheduling 18
2.5.1 Application Bootstrapping 18
2.5.2 Algorithm Scheduling 19

Chapter 3
Writing algorithms 21
3.1 Overview 21
3.2 Algorithm base class 21
3.3 Derived algorithm classes 24
3.3.1 Creation (and algorithm factories) 24
3.3.2 Declaring properties 25
3.3.3 Implementing IAlgorithm 26
3.4 Nesting algorithms 28
3.5 Algorithm sequences, branches and filters 29
3.5.1 Filtering example 30
3.5.2 Sequence branching 30

Chapter 4
Scripting 33

4.1 Disclaimer 33
4.2 Overview 33
4.3 Python overview 34
4.4 Using Python scripting 35
4.4.1 Using Python to drive Athena 35
4.4.2 Using a Python script for configuration and control 36
4.4.3 Using a job options text file for configuration with a Python interactive shell 37
4.5 Prototype functionality 37
4.6 Property manipulation 38
4.7 Synchronization between Python and Athena 39
4.8 Controlling job execution 40

Chapter 5
StoreGate - the event data access model 43
5.1 Overview 43
5.2 The Data Model Architecture 43
5.2.1 Data Objects and Algorithms 43
5.2.2 StoreGate: the Atlas Transient Data Store 44
5.3 Data Objects 45
5.3.1 Using Containers as Data Objects 46
5.3.2 Describing Data Objects to SG 47
5.3.3 Data Object Creation and Ownership of Data Objects 48
5.4 Accessing Data Objects 48
5.4.1 Recording a Data Object 49
5.4.2 Retrieving a Data Object 50
5.5 Using DataLinks to persistify references 52
5.5.1 Creating a DataLink to a data object 53
5.5.2 Creating a Link to an Element of a Container 54
5.5.3 ElementLinks to other Containers 54
5.5.4 Accessing DataLinks 55
5.5.5 DataLinks Persistency 55
5.6 History 56

Chapter 6
Data dictionary 57
6.1 Overview 57
6.2 How to write/read data via POOL 57
6.2.1 Creating a data dictionary filler 59
6.2.2 generating converters 64
6.2.3 writing custom converters 65
6.2.4 setting up the joboptions 73
6.2.5 caveats, problems and work-arounds 74

Chapter 7
Detector Description 77
7.1 Overview 77
7.2 About the Geometry Kernel Classes.. 77
7.3 Examples 78
7.3.1 Example 1: Getting the data into the transient represention. 79
7.3.2 Example 2: Getting the data out of the transient representation. 82
7.4 An Overview of the Geometry Kernel. 83
7.4.1 The Detector Store Service and Detector Managers 83
7.4.2 Material Geometry 84
7.4.3 Materials. 85
7.4.4 Detector Specific Geometrical Services. 91
7.4.5 Alignment. 92
7.4.6 On Memory Use 93

Chapter 8
Histogram facilities 95
8.1 Overview 95
8.2 The Histogram service. 95
8.3 Using histograms and the histogram service 96
8.4 Persistent storage of histograms 97
8.4.1 HBOOK persistency 97
8.4.2 ROOT persistency 98

Chapter 9
N-tuple and Event Collection facilities. 99
9.1 Overview 99
9.2 N-tuples and the N-tuple Service. 99
9.2.1 Access to the N-tuple Service from an Algorithm. 100
9.2.2 Using the N-tuple Service.. 100
9.2.3 N-tuple Persistency 103
9.3 Event Collections. 105
9.3.1 Writing Event Collections 105
9.3.2 Event Collection Persistency 107
9.3.3 Interactive Analysis using Event Tag Collections 110

Chapter 10
Framework services. 113
10.1 Overview 113
10.2 Requesting and accessing services 113
10.3 The Job Options Service 115
10.3.1 Algorithm, Tool and Service Properties 115
10.3.2 Accessing and modifiying properties. 118

10.3.3 Job options file format 118
10.4 The Standard Message Service 122
10.4.1 The MsgStream utility 122
10.5 The Particle Properties Service 124
10.5.1 Initialising and Accessing the Service 125
10.5.2 Service Properties 125
10.5.3 Service Interface 125
10.5.4 Examples 127
10.6 The Chrono & Stat service 127
10.6.1 Code profiling 128
10.6.2 Statistical monitoring 129
10.6.3 Chrono and Stat helper classes 129
10.6.4 Performance considerations 130
10.7 The Auditor Service 130
10.7.1 Enabling the Auditor Service and specifying the enabled Auditors 131
10.7.2 Overriding the default Algorithm monitoring 131
10.7.3 Implementing new Auditors 132
10.8 The Random Numbers Service 132
10.9 The Incident Service 135
10.9.1 Known Incidents 137
10.10 The Gaudi Introspection Service 137
10.11 Developing new services 138
10.11.1 The Service base class 138
10.11.2 Implementation details 139

Chapter 11
Tools and ToolSvc 143
11.1 Overview 143
11.2 Tools and Services 143
11.2.1 Private and Shared Tools 144
11.2.2 The Tool classes 144
11.3 The ToolSvc 149
11.3.1 Retrieval of tools via the `IToolSvc` interface 150
11.4 GaudiTools 151
11.4.1 Associators 152

Chapter 12
Converters 157
12.1 Overview 157
12.2 Persistency converters 157
12.3 Collaborators in the conversion process 158
12.4 The conversion process 160
12.5 Converter implementation - general considerations 163
12.6 Storing Data using the ROOT I/O Engine 163

12.7 The Conversion from Transient Objects to ROOT Objects 164
12.8 Storing Data using other I/O Engines 165

Chapter 13
Visualization 167
13.1 Overview 167

Chapter 14
Framework packages, interfaces and libraries 169
14.1 Overview 169
14.2 Athena Package Structure 169
14.2.1 Packaging Guidelines. 169
14.3 Interfaces in Gaudi. 170
14.3.1 Interface ID 171
14.3.2 Query Interface 172
14.4 Libraries in Athena 173
14.4.1 Component libraries. 173
14.4.2 Linker (or installed) libraries 177
14.4.3 Dual use libraries 178
14.4.4 Linking FORTRAN code. 179

Chapter 15
Analysis utilities 181
15.1 Overview 181
15.2 CLHEP. 181
15.3 HTL 181
15.4 NAG C. 182
15.5 ROOT 182

Appendix A
Options for standard components 183
A.1 Obsolete options 189

Appendix B
Design considerations 191
B.1 Generalities 191
B.2 Designing within the Framework 192
B.3 Analysis Phase. 193
B.4 Design Phase 194
B.4.1 Defining Algorithms 194
B.4.2 Defining Data Objects 194
B.4.3 Mathematics and other utilities 196

Appendix C
Job Options Grammar 197
C.1 The EBNF grammar of the Job Options files 197
C.2 Job Options Error Codes and Error Messages 200

Appendix D
The ATLAS Development Model 203
D.1 Overview. 203
D.2 Strategy 203
D.3 Component Libraries 204
D.4 Dual Use Libraries 204

Appendix E
Package and Directory Structure 207
E.1 Subsystem Package Organization. 207
E.2 Utilities Package Directory Structure 208
E.3 Algorithm and Service Package Directory Structure 208
E.4 Data Package Directory Structure. 209

Appendix F
Standard ATLAS Patterns and Variables 211
F.1 Overview. 211
F.2 Platform Environment Variables 211
F.3 Patterns controlling include paths. 212
F.4 Patterns controlling library creation 213
F.5 Patterns controlling linker options 213
F.6 Patterns for establishing a run-time environment. 213

Appendix G
References 217

Chapter 1
Introduction

1.1 Purpose of the document

This document is intended for developers of the Athena control framework. Athena is based upon the GAUDI architecture that was originally developed by LHCb, but which is now a joint development project. This document, together with other information about Athena, is available online at:

http://web1.cern.ch/Atlas/GROUPS/SOFTWARE/OO/architecture

This version of the Athena Developers Guide corresponds to Athena release 7.5.0. This is based upon ATLAS GAUDI version 0.12.1.5, which itself is based upon GAUDI version 12.

1.2 Athena and GAUDI

As mentioned above Athena is a control framework that represents a concrete implementation of an underlying architecture. The architecture describes the abstractions or components and how they interact with each other. The architecture underlying Athena is the GAUDI architecture originally developed by LHCb. This architecture has been extended through collaboration with ATLAS, and an experiment neutral or kernel implementation, also called GAUDI, has been created. Athena is then the sum of this kernel framework, together with ATLAS-specific enhancements. The latter include the event data model and event generator framework.

The collaboration between LHCb and ATLAS is in the process of being extended to allow other experiments to also contribute new architectural concepts and concrete implementations to the kernel GAUDI framework. It is expected that implementation developed originally for a particular experiment will be adopted as being generic and will be migrated into the kernel. This has already happened with,

for example, the concepts of auditors, the sequencer and the ROOT histogram and ntuple persistency service.

For the remainder of this document the name ***Athena*** is used to refer to the framework and the name ***GAUDI*** is used to refer to the architecture upon which this framework is based.

1.2.1 Document organization

The document is organized as follows:

1.3 Conventions

1.3.1 Units

This section is blank for now.

1.3.2 Coding Conventions

This section is blank for now.

1.3.3 Naming Conventions

This section is blank for now.

1.3.4 Conventions of this document

Angle brackets are used in two contexts. To avoid confusion we outline the difference with an example.

The definition of a templated class uses angle brackets. These are required by the C++ syntax, so in the instantiation of a templated class the angle brackets are retained:

```
AlgFactory<UserDefinedAlgorithm> s_factory;
```

This is to be contrasted with the use of angle brackets to denote replacement such as in the specification of the string:

```
"<concreteAlgorithmType>/<algorithmName>"
```

which implies that the string should look like:

```
"EmptyAlgorithm/Empty"
```

Hopefully what is intended will be clear from the context.

1.4 Release Notes

Although this document is kept as up to date as possible, Athena users should refer to the release notes that accompany each ATLAS software release for any information that is specific to that release. The release notes are kept in the `offline/Control/ReleaseNotes.txt` file.

1.5 Reporting Problems

ATLAS uses the Savannah portal for reporting and tracking of problems. The URL for the Athena project is *http://savannah.cern.ch/projects/athena*.

1.6 User Feedback

Feedback on this Developers Guide, or any other aspects of the documentation for Athena, should be sent to the ATLAS Architecture mailing list at *atlas-sw-architecture@atlas-lb.cern.ch.*

Chapter 2
The framework architecture

2.1 Overview

In this chapter we outline some of the main features of the Athena architecture. A (more) complete view of the architecture, along with a discussion of the main design choices and the reasons for these choices may be found in references [6] and [9].

2.2 Why architecture?

The basic requirement of the physicists is a set of programs for doing event simulation, reconstruction, visualisation, etc. and a set of tools which facilitate the writing of analysis programs. Additionally a physicist wants something that is easy to use and (though he or she may claim otherwise) is extremely flexible. The purpose of the Athena application framework is to provide software which fulfils these requirements, but which additionally addresses a larger set of requirements, including the use of some of the software online.

If the software is to be easy to use it must require a limited amount of learning on the part of the user. In particular, once learned there should be no need to re-learn just because technology has moved on (you do not need to re-take your licence every time you buy a new car). Thus one of the principal design goals was to insulate users (physicist developers and physicist analysists) from irrelevant details such as what software libraries we use for data I/O, or for graphics. We have done this by developing an architecture. An architecture consists of the specification of a number of components and their interactions with each other. A component is a block of software which has a well specified interface and functionality. An interface is a collection of methods along with a statement of what each method actually does, i.e. its functionality.

The main components of the Athena software architecture can be seen in the object diagram shown in Figure 2.1. Object diagrams are very illustrative for explaining how a system is decomposed. They represent a hypothetical snapshot of the state of the system, showing the objects (in our case component instances) and their relationships in terms of ownership and usage. They do not illustrate the structure, i.e. class hierarchy, of the software.

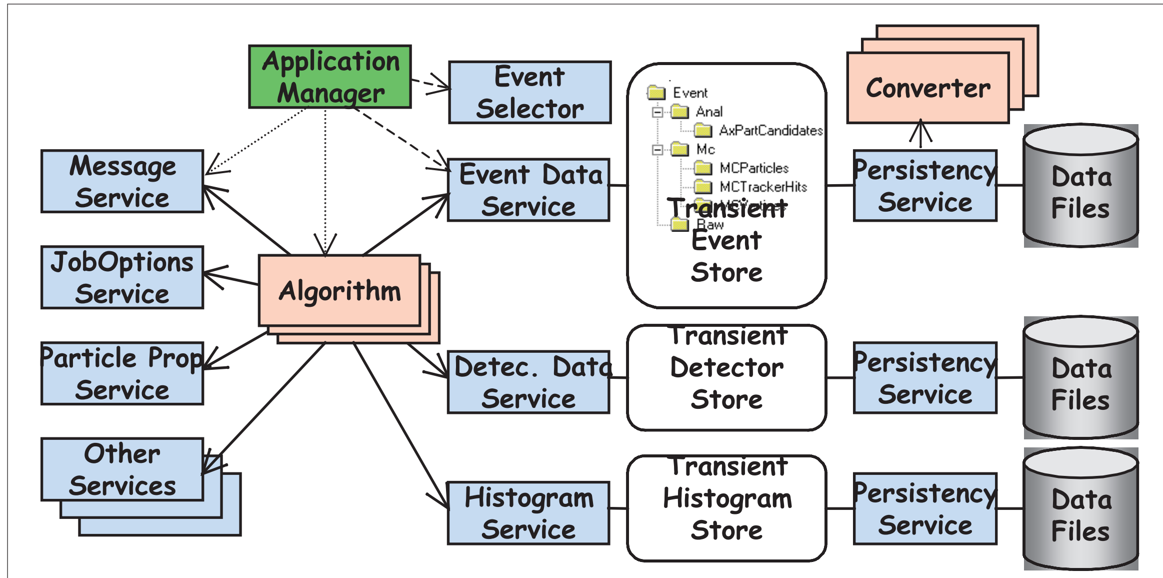

Figure 2.1 Gaudi Architecture Object Diagram

It is intended that almost all software written by physicists, whether for event generation, reconstruction or analysis, will be in the form of specialisations of a few specific components. Here, specialisation means taking a standard component and adding to its functionality while keeping the interface the same. Within the application framework this is done by deriving new classes from one of the base classes:

- DataObject
- Algorithm
- Converter

In this chapter we will briefly consider the first two of these components and in particular the subject of the separation of data and algorithms. They will be covered in more depth in chapters 3 and 7. The third base class, Converter, exists more for technical necessity than anything else and will be discussed in Chapter 12. Following this we give a brief outline of the main components that a physicist developer will come into contact with.

2.3 Data versus code

Broadly speaking, tasks such as physics analysis and event reconstruction consist of the manipulation of mathematical or physical quantities: points, vectors, matrices, hits, momenta, etc., by algorithms which are generally specified in terms of equations and natural language. The mapping of this type of task into a programming language such as FORTRAN is very natural, since there is a very clear distinction between data and code . Data consists of variables such as:

```
integer n
real p(3)
```

and code which may consist of a simple statement or a set of statements collected together into a function or procedure:

```
real function innerProduct(p1, p2)
real p1(3),p2(3)
innerProduct = p1(1)*p2(1) + p1(2)*p2(2) + p1(3)*p2(3)
end
```

Thus the physical and mathematical quantities map to data and the algorithms map to a collection of functions.

A priori, we see no reason why moving to a language which supports the idea of objects, such as C++, should change the way we think of doing physics analysis. Thus the idea of having essentially mathematical objects such as vectors, points etc. and these being distinct from the more complex beasts which manipulate them, e.g. fitting algorithms etc. is still valid. This is the reason why the Athena application framework makes a clear distinction between data objects and algorithm objects.

Anything which has as its origin a concept such as hit, point, vector, trajectory, i.e. a clear quantity-like entity should be implemented by deriving a class from the `DataObject` base class.

On the other hand anything which is essentially a procedure , i.e. a set of rules for performing transformations on more data-like objects, or for creating new data-like objects should be designed as a class derived from the `Algorithm` base class.

Further more you should not have objects derived from `DataObject` performing long complex algorithmic procedures. The intention is that these objects are small .

Tracks which fit themselves are of course possible: you could have a constructor which took a list of hits as a parameter; but they are silly. Every track object would now have to contain all of the parameters used to perform the track fit, making it far from a simple object. Track-fitting is an algorithmic procedure; a track is probably best represented by a point and a vector, or perhaps a set of points and vectors. They are different.

2.4 Main components

The principle functionality of an algorithm is to take input data, manipulate it and produce new output data. Figure 2.1 shows how a concrete algorithm object interacts with the rest of the application framework to achieve this.

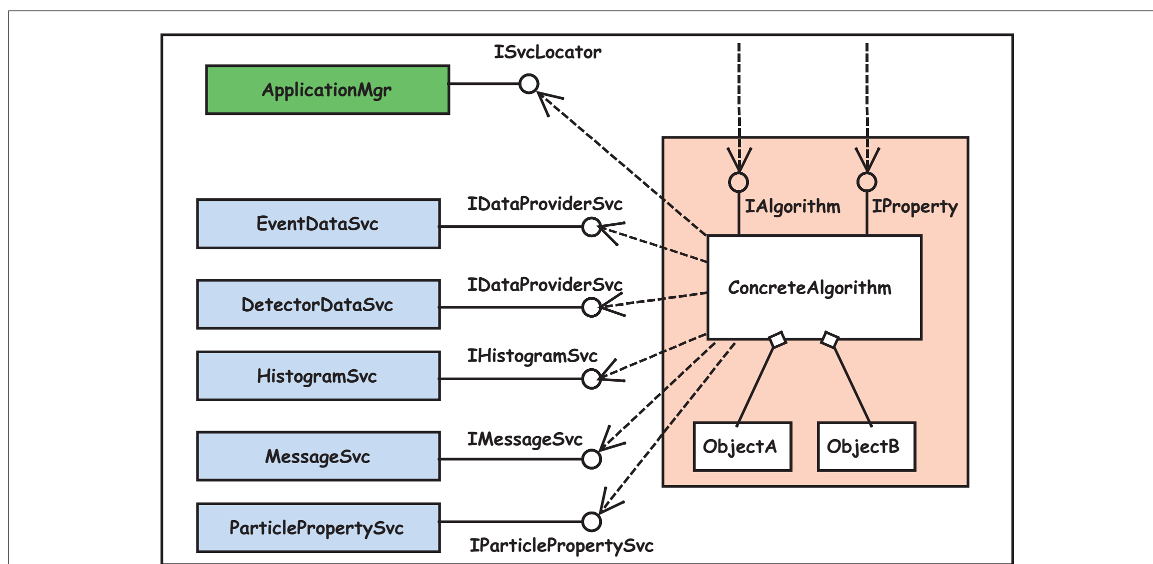

Figure 2.1 The main components of the framework as seen by an algorithm object

The figure shows the four main services that algorithm objects use:

¥ The event data store

¥ The detector data store

¥ The histogram service

¥ The message service

The particle property service is an example of additional services that are available to an algorithm. The job options service (see Chapter 10) is used by the `Algorithm` base class, but is not usually explicitly seen by a concrete algorithm.

Each of these services is provided by a component and the use of these components is via an interface. The interface used by algorithm objects is shown in the figure, e.g. for both the event data and detector data stores it is the `IDataProviderSvc` interface. In general a component implements more than one interface. For example the event data store implements another interface: `IDataManagerSvc` which is used by the application manager to clear the store before a new event is read in.

An algorithm s access to data, whether the data is coming from or going to a persistent store or whether it is coming from or going to another algorithm is always via one of the data store components. The `IDataProviderSvc` interface allows algorithms to access data in the store and to add new data to the store. It is discussed further in Chapter 7 where we consider the data store components in more detail.

The histogram service is another type of data store intended for the storage of histograms and other statistical objects, i.e. data objects with a lifetime of longer than a single event. Access is via the `IHistogramSvc` which is an extension to the `IDataProviderSvc` interface, and is discussed in Chapter 8. The n-tuple service is similar, with access via the `INtupleSvc` extension to the `IDataProviderSvc` interface, as discussed in Chapter 9.

In general, an algorithm will be configurable: It will require certain parameters, such as cut-offs, upper limits on the number of iterations, convergence criteria, etc., to be initialised before the algorithm may be executed. These parameters may be specified at run time via the job options mechanism. This is done by the job options service. Though it is not explicitly shown in the figure this component makes use of the `IProperty` interface which is implemented by the Algorithm base class.

During its execution an algorithm may wish to make reports on its progress or on errors that occur. All communication with the outside world should go through the message service component via the `IMessageSvc` interface. Use of this interface is discussed in Chapter 10.

As mentioned above, by virtue of its derivation from the `Algorithm` base class, any concrete algorithm class implements the `IAlgorithm` and `IProperty` interfaces, except for the three methods `initialize()`, `execute()`, and `finalize()` which must be explicitly implemented by the concrete algorithm. `IAlgorithm` is used by the application manager to control top-level algorithms. `IProperty` is usually used only by the job options service.

The figure also shows that a concrete algorithm may make use of additional objects internally to aid it in its function. These private objects do not need to inherit from any particular base class so long as they are only used internally. These objects are under the complete control of the algorithm object itself and so care is required to avoid memory leaks etc.

We have used the terms interface and implements quite freely above. Let us be more explicit about what we mean. We use the term interface to describe a pure virtual C++ class, i.e. a class with no data members, and no implementation of the methods that it declares. For example:

```
class PureAbstractClass {
  virtual method1() = 0;
  virtual method2() = 0;
}
```

is a pure abstract class or interface. We say that a class implements such an interface if it is derived from it, for example:

```
class ConcreteComponent: public PureAbstractClass {
  method1() { }
  method2() { }
}
```

A component which implements more than one interface does so via multiple inheritance, however, since the interfaces are pure abstract classes the usual problems associated with multiple inheritance do not occur. These interfaces are identified by a unique number which is available via a global constant of

the form: `IID_InterfaceType`, such as for example `IID_IDataProviderSvc`. Interface identifiers are discussed in detail in Chapter 14.

Within the framework every component, e.g. services and algorithms, has two qualities:

- A concrete component class, e.g. `TrackFinderAlgorithm` or `MessageSvc`.
- Its name, e.g. `KalmanFitAlgorithm` or `MessageService`.

2.5 Controlling and Scheduling

2.5.1 Application Bootstrapping

The application is bootstrapped by creating an instance of the *ApplicationMgr* component. The *ApplicationMgr* is in charge of creating an initializing a minimal set of basic and essential services before control is given to specialized scheduling services. These services are shown in Figure 2.1. The *EventLoopMgr* is in charge controlling the main physics event[1] loop and scheduling the top algorithms. There will be a number of more or less specialized implementations of EventLoopMgr which will perform the different actions depending on the running environment, and experiment specific policies (clearing stores, saving histograms, etc.). There exists the possibility to give the full control of the application to a component implementing the `IRunable` interface. This is needed for interactive applications such as event display, interactive analysis, etc. The *Runable* component can interact directly with the *EventLoopMgr* for triggering the processing of the next physics event.

The essential services that the *ApplicationMgr* need to instantiate and initialize are the *MessageSvc* and *JobOptionsSvc*.

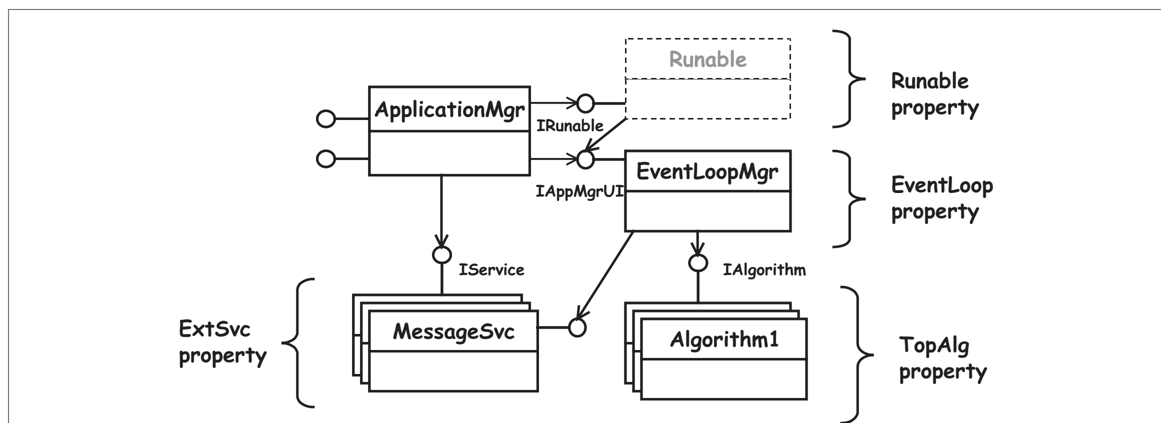

Figure 2.1 Control and Scheduling collaboration

1. We state physics event to differentiate from what is called generally an *event* in computing.

2.5.2 Algorithm Scheduling

The Gaudi architecture foresees explicit invocation of algorithms by the framework or by other algorithms. This latter possibility allows for a hierarchical organization of algorithms, for example, a high level algorithm invoking a number of sub-algorithms.

The *EventLoopMgr* component is in charge of initializing, finalizing and executing the set of Algorithms that have been declared with the `TopAlg` property. These *Algorithms* are executed unconditionally in the order they have been declared. This vary basic scheduling is insufficient for many use cases (event filtering, conditional execution, etc.). Therefore, a number of *Algorithms* have been introduced that perform more sophisticated scheduling and can be configured by some properties. Examples are: Sequencers, Prescalers, etc. and the list can be easily extended. See Section 3 for more details on these generic high level Algorithms.

Chapter 3
Writing algorithms

3.1 Overview

As mentioned previously the framework makes use of the inheritance mechanism for specialising the `Algorithm` component. In other words, a concrete algorithm class must inherit from (be derived from in C++ parlance, extend in Java) the `Algorithm` base class.

In this chapter we first look at the base class itself. We then discuss what is involved in creating concrete algorithms: specifically how to declare properties, what to put into the methods of the `IAlgorithm` interface, the use of private objects and how to nest algorithms. Finally we look at how to set up sequences of algorithms and how to control processing through the use of branches and filters.

3.2 Algorithm base class

Since a concrete algorithm object *is-an* `Algorithm` object it may use all of the public and protected methods of the `Algorithm` base class. The base class has no protected or public data members, so in fact, these are the only methods that are available. Most of these methods are provided solely to make the implementation of derived algorithms easier. The base class has two main responsibilities: the initialization of certain internal pointers and the management of the properties of derived algorithm classes.

A part of the `Algorithm` base class definition is shown in Listing 3.1. Include directives, forward declarations and private member variables have all been suppressed. It declares a constructor and destructor; some methods of the `IAlgorithm` interface; several accessors to services that a concrete

algorithm will almost certainly require; a method to create a sub algorithm, the two methods of the `IProperty` interface; and a whole series of methods for declaring properties.

Listing 3.1 The definition of the Algorithm base class.

```
 1: class Algorithm : virtual public IAlgorithm,
                     virtual public IProperty {
 2: public:
 3: // Constructor and destructor
 4: Algorithm( const std::string& name, ISvcLocator *svcloc );
 5: virtual ~Algorithm();
 6:
 7: // IAlgorithm interface only partially implemented
 8: StatusCode sysInitialize();
 9: StatusCode sysExecute();
10: StatusCode sysFinalize();
11: StatusCode beginRun();
12: StatusCode endRun();
13: const std::string&  name() const;
14:
15: virtual bool isExecuted() const;
16: virtual StatusCode setExecuted( bool state );
17: virtual StatusCode resetExecuted();
18: virtual bool isEnabled() const;
19: virtual bool filterPassed() const;
20: virtual StatusCode setFilterPassed( bool state );
21:
22: // Service accessors
23: template<class T> StatusCode service( const std::string& name, T*& svc,
    bool createIf = false );
24: void setOutputLevel( int level );
25: IMessageSvc*        msgSvc()          const;
26: IAuditorSvc*        auditorSvc()      const;
27: IDataProviderSvc*   eventSvc()        const;
28: IConversionSvc*     eventCnvSvc()     const;
29: IDataProviderSvc*   detSvc()          const;
30: IConversionSvc*     detCnvSvc()       const;
31: IHistogramSvc*      histoSvc()        const;
32: INtupleSvc*         ntupleSvc()       const;
33: IChronoStatSvc*     chronoSvc()       const;
34: IRndmGenSvc*        randSvc()         const;
35: IToolSvc*           toolSvc()         const;
36: ISvcLocator*        serviceLocator()  const;
37:
38: StatusCode createSubAlgorithm( const std::string& type,
                      const std::string& name, Algorithm*& pSubAlg );
39: std::vector<Algorithm*>* subAlgorithms() const;
40:
```

Listing 3.1 The definition of the Algorithm base class.

```
41:  // IProperty interface
42:  virtual StatusCode setProperty( const Property& p);
43:  virtual StatusCode setProperty( std::istream s& );
44:
45:  virtual StatusCode setProperty( const std::string& n,
                                 const std::string& v);
46:  virtual StatusCode getProperty( Property* p ) const;
47:  const   Property&  getProperty( const std::string& name) const;
48:  virtual StatusCode getProperty( const std::string& n,
                                      std::string& v) const;
49:  const std::vector<Property*>& getProperties() const;StatusCode
     setProperties();
50:  template <class T>
       StatusCode declareProperty(const std::string& name, T& property);
51:  StatusCode declareRemoteProperty(const std::string& name,
               IProperty* rsvc, const std::string& rname = "") const;
52:  /// Methods for IInterface
53:  unsigned long addRef();
54:  unsigned long release();
55:  StatusCode queryInterface(const IID& riid, void**);
56:
57:  protected:
58:    bool isInitialized( ) const;
59:    void setInitialized( );
60:    bool isFinalized( ) const;
61:    void setFinalized( );
62:  private:
63:  // Data members not shown
64:  Algorithm(const Algorithm& a);   // NO COPY ALLOWED
65:  Algorithm& operator=(const Algorithm& rhs); // NO ASSIGNMENT ALLOWED};
```

Constructor and Destructor The base class has a single constructor which takes two arguments: The first is the name that will identify the algorithm object being instantiated and the second is a pointer to one of the interfaces implemented by the application manager: `ISvcLocator`. This interface may be used to request special services that an algorithm may wish to use, but which are not available via the standard accessor methods (below).

The `IAlgorithm` interface The base class only partially implements this interface: the three pure virtual methods `initialize()`, `execute()` and `finalize()` must be implemented by a derived algorithm: these are where the algorithm does its useful work and are discussed in more detail in section 3.3. The base class provides default implementations of the methods `beginRun()` and `endRun()`, and the accessor `name()` which returns the algorithm s identifying name. The methods `sysInitialize()`, `sysFinalize()`, `sysExecute()` are used internally by the framework; they are not virtual and may not be overridden.

Service accessor methods Lines 25 to 35 declare accessor methods which return pointers to key service interfaces. These methods are available for use only <u>after</u> the Algorithm base class has been initialized, i.e. they may not be used from within a concrete algorithm constructor, but may be used from within the `initialize()` method (see Section 3.3.3). The services and interface types to

which they point are self explanatory. Services may be located by name using the templated `service()` function in line 23 or by using the `serviceLocator()` accessor method on line 36, as described in Section 10.2. Line 24 declares a facility to modify the message output level from within the code (the message service is described in Section 10.4).

Creation of sub algorithms The methods on lines 38 to 39 are intended to be used by a derived class to manage sub-algorithms, as discussed in section 3.4.

Declaration and setting of properties A concrete algorithm must declare its properties to the framework using the templated `declareProperty` method (line 50), as discussed in Section 3.3.2 and Section 10.3.1. The Algorithm base class then uses the `setProperties()` method (line 49) to tell the framework to set these properties to the values defined in the job options file. The methods in lines 42 to 49 can later be used to access and modify the values of specific properties, as explained in Section 10.3.2.

Filtering The methods in lines 14 to 19 are used by sequencers and filters to access the state of the algorithm, as discussed in Section 3.5.

3.3 Derived algorithm classes

In order for an algorithm object to do anything useful it must be specialised, i.e. it must extend (inherit from, be derived from) the `Algorithm` base class. In general it will be necessary to implement the methods of the `IAlgorithm` interface, and declare the algorithm s properties to the property management machinery of the `Algorithm` base class. Additionally there is one non-obvious technical matter to cover, namely algorithm factories.

3.3.1 Creation (and algorithm factories)

A concrete algorithm class must specify a single constructor with the same parameter signature as the constructor of the base class.

In addition to this, a concrete algorithm factory must be provided. This is a technical matter which permits the application manager to create new algorithm objects without having to include all of the concrete algorithm header files. From the point of view of an algorithm developer it implies adding three lines into the implementation file, of the form:

```
#include "GaudiKernel/AlgFactory.h"
...
static const AlgFactory<ConcreteAlgorithm>  s_factory;
const IAlgFactory& ConcreteAlgorithmFactory = s_factory;
```

where `ConcreteAlgorithm` should be replaced by the name of the derived algorithm class (see for example lines 10 and 11 in Listing 3.2 below).

3.3.2 Declaring properties

In general, a concrete algorithm class will have several data members which are used in the execution of the algorithm proper. These data members should of course be initialized in the constructor, but if this was the only mechanism available to set their value it would be necessary to recompile the code every time you wanted to run with different settings. In order to avoid this, the framework provides a mechanism for setting the values of member variables at run time.

The mechanism comes in two parts: the declaration of properties and the setting of their values. As an example consider the class `TriggerDecision` in Listing 3.2 which has a number of variables whose value we would like to set at run time.

Listing 3.2 Declaring member variables as properties.

```
 1: //------- In the header file --------------------------------------//
 2: class TriggerDecision : public Algorithm {
 3:
 4: private:
 5:   bool m_passAllMode;
 6:   int m_muonCandidateCut;
 7:   std::vector m_ECALEnergyCuts;
 8: }
 9: //------- In the implementation file ------------------------------//
10: static const AlgFactory<TriggerDecision>  s_factory;
11: const IAlgFactory& TriggerDecisionFactory = s_factory;
12:
13: TriggerDecision::TriggerDecision(std::string name, ISvcLocator *pSL) :
14:   Algorithm(name, pSL), m_passAllMode(false), m_muonCandidateCut(0) {
15:   m_ECALenergyCuts.push_back(0.0);
16:   m_ECALenergyCuts.push_back(0.6);
17:
18:   declareProperty(“PassAllMode”, m_passAllMode);
19:   declareProperty(“MuonCandidateCut”, m_muonCandidateCut);
20:   declareProperty(“ECALEnergyCuts”, m_ECALEnergyCuts);
21: }
22:
23: StatusCode TriggerDecision::initialize() {
24: }
```

The default values for the variables are set within the constructor (within an initialiser list). To declare them as properties it suffices to call the `declareProperty()` method. This method is templated to take an `std::string` as the first parameter and a variety of different types for the second parameter. The first parameter is the name by which this member variable shall be referred to, and the second parameter is a reference to the member variable itself.

In the example we associate the name `PassAllMode` to the member variable `m_passAllMode`, and the name `MuonCandidateCut` to `m_muonCandidateCut`. The first is of type boolean and the second an integer. If the job options service (described in Section 10.3 on page 115) finds an option in the job options file belonging to this algorithm and whose name matches one of the names associated

with a member variable, then that member variable will be set to the value specified in the job options file.

3.3.3 Implementing IAlgorithm

Any concrete algorithm must implement the three pure virtual methods `initialize()`, `execute()` and `finalize()` of the `IAlgorithm` interface. For a top level algorithm, i.e. one controlled directly by the application manager, the methods are invoked as is described in section 2.6. This dictates what it is useful to put into each of the methods.

Initialization Figure 3.1 shows an example trace of the initialization phase.In a standard job the application manager will initialize all top level algorithms exactly once before reading any event data. It does this by invoking the `sysInitialize()` method of each top-level algorithm in turn, in which the framework takes care of setting up internal references to standard services and to set the algorithm properties (using the mechanism described in Section 10.3.1 on page 115). At the end, `sysInitialize()` calls the `initialize()` method, which can be used to do such things as creating histograms, or creating sub-algorithms if required (sub-algorithms are discussed in Section 3.4). If an algorithm fails to initialize it should return `StatusCode::FAILURE`. This will cause the job to terminate.

Execution The guts of the algorithm class is in the `execute()` method. For top level algorithms this will be called once per event for each algorithm object in the order in which they were declared to the application manager. For sub-algorithms (Section 3.4) the control flow may be as you like: you may call the `execute()` method once, many times or not at all.

Just because an algorithm derives from the `Algorithm` base class does not mean that it is limited to using or overriding only the methods defined by the base class. In general, your code will be much better structured (i.e. understandable, maintainable, etc.) if you do not, for example, implement the `execute()` method as a single block of 100 lines, but instead define your own utility methods and classes to better structure the code.

If an algorithm fails in some manner, e.g. a fit fails to converge, or its data is nonsense it should return from the `execute()` method with `StatusCode::FAILURE`. This will cause the application manager to stop processing events and end the job. This default behaviour can be modified by setting the `<myAlgorithm>.ErrorMax` job option to something greater than 1. In this case a message will be printed, but the job will continue as if there had been no error, and just increment an error count. The job will only stop if the error count reaches the `ErrorMax` limit set in the job option.

The framework (the Algorithm base class) calls the `execute()` method within a try/catch clause. This means that any exception not handled in the execution of an Algorithm will be caught at the level of `sysExecute()` implemented in the base class. The behaviour on these exceptions is identical to that described above for errors.

Finalization The `finalize()` method is called at the end of the job. It can be used to analyse statistics, fit histograms, or whatever you like. Similarly to initialization, the framework invokes a `sysFinalize()` method which in turn invokes the `finalize()` method of the algorithm and of any sub-algorithms.

Figure 3.1 Algorithm initialization.

Optionally, the methods `beginRun()` and `endRun()` can also be implemented. These are called at the beginning and the end of the event loop respectively.

Monitoring of the execution (*e.g.* cpu usage) of each Algorithm instance is performed by *auditors* under control of the Auditor service (described in Section 10.7 on page 130). This monitoring can be turned on or off with the boolean properties `AuditInitialize, AuditExecute, AuditFinalize`.

The following is a list of things to do when implementing an algorithm.

¥ Derive your algorithm from the `Algorithm` base class.

¥ Provide the appropriate constructor and the three methods `initialize()`, `execute()` and `finalize()`.

¥ Make sure you have implemented a factory by adding the magic two lines of code (see Section 3.3.1).

3.4 Nesting algorithms

The application manager is responsible for initializing, executing once per event, and finalizing the set of top level algorithms, i.e. the set of algorithms specified in the job options file. However such a simple linear structure is very limiting. You may wish to execute some algorithms only for specific types of event, or you may wish to loop over an algorithm s execute method. Within the Athena application framework the way to have such control is via the nesting of algorithms or through algorithm sequences (described in section 5.5). A nested (or sub-) algorithm is one which is created by, and thus belongs to and is controlled by, another algorithm (its parent) as opposed to the application manager. In this section we discuss a number of points which are specific to sub-algorithms.

In the first place, the parent algorithm will need a member variable of type `Algorithm*` (see the code fragment below) in which to store a pointer to the sub-algorithm.

```
Algorithm* m_pSubAlgorithm;    // Pointer to the sub algorithm
                               // Must be a member variable of the parent class
std::string type;              // Type of sub algorithm
std::string name;              // Name to be given to subAlgorithm
StatusCode sc;                 // Status code returned by the call
sc = createSubAlgorithm(type, name, Algorithm*& m_pSubAlgorithm);
```

The sub-algorithm itself is created by invoking the `createSubAlgorithm()` method of the `Algorithm` base class. The parameters passed are the type of the algorithm, its name and a reference to the pointer which will be set to point to the newly created sub-algorithm. Note that the name passed into the `createSubAlgorithm()` method is the same name that should be used within the job options file for specifying algorithm properties.

The algorithm type (i.e. class name) string is used by the application manager to decide which factory should create the algorithm object.

The execution of the sub-algorithm is entirely the responsibility of the parent algorithm whereas the `initialize()` and `finalize()` methods are invoked automatically by the framework as shown in Figure 3.1. Similarly the properties of a sub-algorithm are also automatically set by the framework.

Note that the `createSubAlgorithm()` method returns a pointer to an `Algorithm` object, not an `IAlgorithm` interface. This means that you have access to the methods of both the `IAlgorithm` and `IProperty` interfaces, and consequently as well as being able to call `execute()` etc. you may also change the properties of a sub-algorithm during the main event loop as explained in Section 10.3.1. Note also that the vector of pointers to the sub-algorithms is available via the `subAlgorithms()` method.

3.5 Algorithm sequences, branches and filters

A physics application may wish to execute different algorithms depending on the physics signature of each event, which might be determined at run-time as a result of some reconstruction. This capability is supported in Athena through sequences, branches and filters. A *sequence* is a list of Algorithms. Each Algorithm may make a *filter* decision, based on some characteristics of the event, which can either allow or bypass processing of the downstream algorithms in the sequence. The filter decision may also cause a *branch* whereby a different downstream sequence of Algorithms will be executed for events that pass the filter decision relative to those that fail it. Eventually the particular set of sequences, filters and branches might be used to determine which of multiple output destinations each event is written to (if at all). This capability is not yet implemented but is planned for a future release of Athena.

A `Sequencer` class is available in the `GaudiAlg` package which manages algorithm sequences using filtering and branching protocols which are implemented in the `Algorithm` class itself. The list of Algorithms in a Sequencer is specified through the `Members` property. Algorithms can call `setFilterPassed( true/false )` during their `execute()` function. Algorithms in the membership list downstream of one that sets this flag to `false` will not be executed, *unless* the `StopOverride` property of the Sequencer has been set, or the filtering algorithm itself is of type `Sequencer` and its `BranchMembers` property specifies a branch with downstream members. Please note that, if a sub-algorithm is of type `Sequencer`, the parent algorithm must call the `resetExecuted()` method of the sub-algorithm before calling the `execute()` method, otherwise the sequence will only be executed once in the lifetime of the job!

An algorithm *instance* is executed only once per event, even if it appears in multiple sequences. It may also be enabled or disabled, being enabled by default. This is controlled by the `Enable` property. Enabling and disabling of algorithm instances is a capability that is designed for a future release of Athena that will include an interactive scripting language.

The filter passed or failed logic for a particular Algorithm instance in a sequence may be inverted by specifying the `:invert` optional flag in the `Members` list for the `Sequencer` in the job options file.

A Sequencer will report filter success if either of its main and branch member lists succeed. The two cases may be differentiated using the `Sequencer` `branchFilterPassed()` boolean function. If

this is set true, then the branch filter was passed, otherwise both it and the main sequence indicated failure.

The following examples illustrate the use of sequences with filtering and branching.

3.5.1 Filtering example

Listing 3.3 is an extract of the job options file of the `AlgSequencer` example: a `Sequencer` instance is created (line 2) with two *members* (line 5); each member is itself a `Sequencer`, implementing the *sequences* set up in lines 7 and 8, which consist of `Prescaler`, `EventCounter` and `HelloWorld` algorithms. The `StopOverride` property of the `TopSequence` is set to true, which causes both sequences to be executed, even if the first one indicates a filter failure.

The `Prescaler` and `EventCounter` classes are example algorithms distributed with the `GaudiAlg` package. The `Prescaler` class acts as a filter, passing the fraction of events specified by the `PercentPass` property (as a percentage). The `EventCounter` class just prints each event as it is encountered, and summarizes at the end of job how many events were seen. Thus at the end of job, the `Counter1` instance will report seeing 50% of the events, while the `Counter2` instance will report seeing 10%.

Note the same instance of the `HelloWorld` class appears in both sequences. It will be executed in `Sequence1` if `Prescaler1` passes the event. It will be executed in `Sequence2` if `Prescaler2` passes the event *only* if `Prescaler1` failed it.

Listing 3.3 Example job options using Sequencers demonstrating filtering

```
 1: ApplicationMgr.DLLs  += {  "GaudiAlg" };
 2: ApplicationMgr.TopAlg = {  "Sequencer/TopSequence" };
 3:
 4: // Setup the next level sequencers and their members
 5: TopSequence.Members = {"Sequencer/Sequence1", "Sequencer/Sequence2"};
 6: TopSequence.StopOverride = true;
 7: Sequence1.Members = {"Prescaler/Prescaler1", "HelloWorld",
    "EventCounter/Counter1"};
 8: Sequence2.Members = {"Prescaler/Prescaler2", "HelloWorld",
    "EventCounter/Counter2"};
 9:
10: Prescaler1.PercentPass = 50.;
11: Prescaler2.PercentPass = 10.;
```

3.5.2 Sequence branching

Listing 3.4 illustrates the use of explicit branching. The `BranchMembers` property of the `Sequencer` specifies some algorithms to be executed if the algorithm that is the first member of the branch (which is common to both the main and branch membership lists) indicates a filter failure. In

this example the EventCounter instance Counter1 will report seeing 80% of the events, whereas Counter2 will report seeing 20%.

Listing 3.4 Example job options using Sequencers demonstrating branching

```
1: ApplicationMgr.DLLs  += {  "GaudiAlg" };
2: ApplicationMgr.TopAlg = {  "Sequencer" };
3:
4: // Setup the next level sequencers and their members
5: Sequencer.Members = {"HelloWorld", "Prescaler",
   "EventCounter/Counter1"};
6: Sequencer.BranchMembers = {"Prescaler", "EventCounter/Counter2"};
7:
8: Prescaler.PercentPass = 80.;
```

Listing 3.5 illustrates the use of inverted logic. It achieves the same goal as the example in Listing 3.4 through use of two sequences with the same instance of a Prescaler filter, but where the second sequence contains inverted logic for the single instance.

Listing 3.5 Example job options using Sequencers demonstrating inverted logic

```
1: ApplicationMgr.DLLs  += {  "GaudiAlg" };
2: ApplicationMgr.TopAlg = {  "Sequencer/Seq1", "Sequencer/Seq2" };
3:
4: // Setup the next level sequencers and their members
5: Seq1.Members = {"HelloWorld", "Prescaler", "EventCounter/Counter1"};
6: Seq2.Members = {"HelloWorld", "Prescaler:invert",
   "EventCounter/Counter2"};
7:
8: Prescaler.PercentPass = 80.;
```

Chapter 4
Scripting

4.1 Disclaimer

```
Athena scripting support is available in experimental form. User
feedback is required to define the best approach and backwards
compatibility can not always be guaranteed for future versions of the
scripting facilities. Functionality is likely to change rapidly, so
users should check with the latest release notes for changes or new
functionality that might not be documented here.
```

4.2 Overview

A scripting service for Athena can be provided in several possible ways. In order to gain experience and to find out what is best, the following is available in the prototype which uses Python[4] as the scripting language and interactive interpreter:

¥ A Python script that starts and runs Athena. This way, Python is on par with Athena.

¥ A Python scripting service that is run from Athena, which is closer to the design philosophy of Athena and the underlying GAUDI architecture.

For the second case, in the current implementation, the Python scripting can be started explicitly by specifying it as the runable in your jobOptions.txt file or implicitly by giving the Athena executable a Python script. Athena will then set the proper options for you.

From a technical point of view, more funtionality (read: code) can be put on the Python side in the first case. In the second case, the scripting service will be more restricted and therefore easier replaced by a

different scripting language. However, considering that Athena is written in C++ and considering that Python is a more portable and much more productive programming language, paying the upfront cost of keeping the scripting service generic and as much functionality as possible in C++, is unlikely to pay off in the long run. Unless, of course, Athena ends up supporting a myriad of scripting services.

Each of the methods, that were mentioned, of enabling Python scripting will be described in this chapter. The Python scripting language itself will not be described in detail here, but only a brief overview, sufficient to write scripts for Athena, will be presented.

4.3 Python overview

Python is a full-flexed, interpreted, open source, free programming langauge. In it you ll find all the programming elements you need, either in the language or in so-called Python modules (libraries). The complete Python language is available for use, but the main parts that are needed for writing Athena scripts are basic function calls, object creation, and member selection. Next to standard types such as double and int, the string and list builtin types are often useful. Python allows you to use other builtin collection classes like tuples and dictionaries, and you can define your own functions and classes.

Like many other scripting languages, Python uses dynamic typing (ie. you don t need to declare any of your variables) and is rather generous with implicit conversions. Memory management is done for you. Further, Python provides for many ways of exploring the run-time environment from the interpreter prompt (which looks like >>>). Python is extensible by means of modules and interfaces very easily to other popular programming languages.

When writing code, be aware that Python uses explicit indentation to indicate the scope of a block. Blocks are not needed when you write simple Athena scripts, so just make sure that you start all statements on the first character position. Python does not require an end-of-line nor end-of-statement character, except when opening a block.

A basic function call is very similar to what you would find in other languages: *funcname(arglist)*, where *"arglist"* is a comma-separated list of arguments that are passed to the function. Objects are created by calling the class name like a function and assigning the result to a variable: *varname* = *ClassName*(*arglist*), where *"arglist"* is again a comma-separated list of arguments. Object members are selected using the .operator (dot-operator, like in Java and C/C++). Members may include data types (including objects) as well as functions.

Character strings are constructed by enclosing text in between single ( ) or double (") quotes. Lists are expandable arrays and are constructed by opening with a "[" followed by the comma-seperated elements and closed by a "]" ("[]" is the empty list). Both strings and lists can be concatenated using the +operator, or you can append to a string or list variable by using the +=operator.

Tips: use "#" to indicate the start of a comment. The comment extends to the end of the line, nest, and can not be carried with a backslash. Use the "print" command to sent string output to the screen.

For examples of actual Python code, please refer to the listings that follow or look at any of the .py files in the Athena distribution. More information on Python is available on *http://www.python.org*, which includes an up to date documentation of the available modules, and in many books on the language.

4.4 Using Python scripting

Three different mechanisms are available, in the current implementation of Athena, for enabling Python scripting. The methods are:

1. Use the `athena.py` script to run Athena/GAUDI from Python.
2. Replace, on the command line, the job options text file with a Python script.
3. Use a job options text file which hands control over to the Python shell once the initial configuration has been established.

The 2nd and 3th method are very similar. The only difference being who specifies the Python scripting service as the runable (Athena or the user). The first method is the most powerful and easiest to develop. It s also the recommend method, but it has a few issues when debugging, see below.

4.4.1 Using Python to drive Athena

The `athena.py` script is fully backwards compatible with the `athena.exe` executable: you can use it to drive Athena with a job options text file. Instead, or on top of that, you can also specify any number of Python scripts that need to be executed, see Listing 4.1

Listing 4.1 Using Python to drive Athena

```
athena.py                                                    [ 1]
athena.py myJobOptions.txt                                   [ 2]
athena.py myScript.py [ myScript2.py [ myScript3.py [ ...]]]  [ 3]
```

Notes:

1. When the script is run with no arguments, it will look for the default jobOptions.txt file, start up Athena, and if it found the default file, run it. If it didn t find any default file, it will present you with the Python prompt and you can take it from there.
2. Same as `[ 1]` , except that an explicit job options text file is specified. The file must exist (if it doesn t, the script will exit) and Athena is run in batch mode.
3. Any number of Python scripts can be run, the file extension `.py` is used to detect Python scripts. The scripts are executed in order and the first is expected to call `theApp.setup( MONTECARLO )` or `theApp.setup( ZEBRA )` , depending on your needs, before instantiating/using any services or algorithms. This minor wart is expected to go away in a future installment

If you include a `.txt` file on the command line in [3] it is assumed to be a job options text file and treated accordingly. In all cases, any other kind of file will be seen as an error and will stop the script.

Debugging a script is different from debugging an application because your debugger won t find any symbols in the script, causing it to complain. You can run the debugger with this script in two ways. First, the more "classical" approach (using `gdb` as an example):

1. Start the debugger with `python` as the executable: `gdb python`
2. Specify the program arguments: `(gdb) set args -itx athena.py [ <scripts>]`
3. Run the program like you re used to: `(gdb) run`

Alternatively, you can specify the `-d` option to `athena.py` . This will start Athena, spawn the debugger (default: `dbg`, specify `-d dbx` or anything else you might like on Solaris), and give you the prompt of the debugger just before Athena would be starting to run the algorithms. You can alternate between the Python and `gdb` prompts by hitting ^C in Python or typing `continue` on the `dbg` prompt, unless you re running of a job options text file only, in which case you won t get to the Python prompt.

If your final Python script is not ended with a call to `theApp.exit()` , you ll be left with the Python prompt after execution of all scripts.

4.4.2 Using a Python script for configuration and control

Currently, this implementation is flawed due to a few inconsistensies in GAUDI. You should only use it if you intend to use the McEventSelector. You ll likely have problems if you want to use the ZebraTDR facilities instead. It hasn t been tested, though. It might work.

The necessity for using a job options text file for configuration can be avoided by specifying a Python script as a command line argument as shown in Listing 4.2.

Listing 4.2 Using a Python script for job configuration

```
athena MyPythonScript.py                                                    [ 1]
```

Notes:

1. The file extension `.py` is used to identify the job options file as a Python script. All other extensions are assumed to be job options text files.

This approach may be used in two modes. The first uses such a script to establish the configuration, but results in the job being left at the Python shell prompt. This supports interactive sessions. The second specifies a complete configuration and control sequence and thus supports a batch style of processing. The particular mode is controlled by the presence or absence of Athena-specific Python commands described in Section *4.8.*

4.4.3 Using a job options text file for configuration with a Python interactive shell

Python scripting is enabled when using a job options text file for job configuration by adding the lines shown in Listing 4.3 to the job options file.

Listing 4.3 Job Options text file entries to enable Python scripting

```
ApplicationMgr.DLLs    = { "GaudiPython", "McEventSelector };          [1]
ApplicationMgr.Runable = "PythonScriptingSvc";                         [2]
```

Notes:

1. This entry specifies the component library that implements Python scripting. If other DLLs are specified first, then care should be taken to use the +=operator syntax in order not to overwrite the other component libraries.
2. This entry specifies the use of the Python scripting implementation as the run manager.

Once the initial configuration has been established by the job options text file, control will be handed over to the Python shell. It is possible to run in batch mode, still: simply pipe the Python script, as is shown in Listing 4.4.

Listing 4.4 Specifying a job options file for application execution

```
athena [job options file] < MyPythonScript.py                          [1]
```

4.5 Prototype functionality

The functionality of the prototype is limited to the following capabilities. This list will be added to as new capabilities are made available:

1. The ability to read and store basic Properties for framework components (Algorithms, Services, Auditors) and the main ApplicationMgr that controls the application. Basic properties are basic type data members (int, float, *etc.*) or SimpleProperties of the components that are declared as Properties via the declareProperty() function.
2. The ability to retrieve and store individual elements of array (list) properties.
3. The ability to specify a new set of top level Algorithms.
4. The ability to add new services and component libraries and access their capabilities
5. The ability to specify a new set of members or branch members for Sequencer algorithms.
6. The ability to specify a new set of output streams.

7. The ability to specify a new set of "AcceptAlgs", "RequireAlgs", or "VetoAlgs" properties for output streams.

4.6 Property manipulation

An illustration of the use of the scripting language to display and set component properties is shown in Listing 4.5:

Listing 4.5 Property manipulation from the Python interactive shell

```
>>>theApp.algorithms()
[ 1][ 2]
('TopSequence', 'Sequence1', 'Sequence2')

>>> theApp.services()                                              [ 3]
('MessageSvc', 'JobOptionsSvc', 'EventDataSvc', 'EventPersistencySvc',
'DetectorDataSvc', 'DetectorPersistencySvc', 'HistogramDataSvc',
'NTupleSvc', 'IncidentSvc', 'ToolSvc', 'HistogramPersistencySvc',
'ParticlePropertySvc', 'ChronoStatSvc', 'RndmGenSvc', 'AuditorSvc',
'ScriptingSvc', 'RndmGenSvc.Engine')

>>> TopSequence.properties()                                       [ 4]
{'ErrorCount': 0, 'OutputLevel': 0, 'BranchMembers': [],
'AuditExecute': 1, 'AuditInitialize': 0, 'Members':
['Sequencer/Sequence1', 'Sequencer/Sequence2'], 'StopOverride': 1,
'Enable': 1, 'AuditFinalize': 0, 'ErrorMax': 1}

>>> TopSequence.OutputLevel                                        [ 5]
'OutputLevel': 0

>>> TopSequence.OutputLevel=1                                      [ 6]

>>> TopSequence.Members=['Sequencer/NewSeq1', 'Sequencer/NewSeq1']     [ 7]

>>> TopSequence.properties()
{'ErrorCount': 0, 'OutputLevel': 1, 'BranchMembers': [],
'AuditExecute': 1, 'AuditInitialize': 0, 'Members':
['Sequencer/NewSeq1', 'Sequencer/NewSeq1'], 'StopOverride': 1,
'Enable': 1, 'AuditFinalize': 0, 'ErrorMax': 1}

>>> theApp.properties()                                            [ 8]
{'JobOptionsType': 'FILE', 'EvtMax': 100, 'DetDbLocation': 'empty',
'Dlls': ['HbookCnv', 'SI_Python'], 'DetDbRootName': 'empty',
'JobOptionsPath': 'jobOptions.txt', 'OutStream': [],
'HistogramPersistency': 'HBOOK', 'EvtSel': 'NONE', 'ExtSvc':
['PythonScriptingSvc/ScriptingSvc'], 'DetStorageType': 0, 'TopAlg':
['Sequencer/TopSequence']}
>>>
```

Notes:

1. The ">>>" is the Python shell prompt.
2. The set of existing Algorithms is given by the theApp.algorithms() command.
3. The set of existing Services is given by the theApp.services() command.
4. The values of the properties for an Algorithm or Service may be displayed using the <name>.properties() command, where <name> is the name of the desired Algorithm or Service.
5. The value of a single Property may be displayed (or used in a Python expression) using the <name>.<property> syntax, where <name> is the name of the desired Algorithm or Service, and <property> is the name of the desired Property.
6. Single valued properties (*e.g.* IntegerProperty) may be set using an assignment statement. Boolean properties use integer values of 0 (or FALSE) and 1 (or TRUE). Strings are enclosed in " " characters (single-quotes) or """ characters (double-quotes).
7. Multi-valued properties (*e.g.* StringArrayProperty) are set using "[...]" as the array (list) delimiters.
8. The theApp object corresponds to the ApplicationMgr and may be used to access its properties.

4.7 Synchronization between Python and Athena

It is possible to create new Algorithms or Services as a result of a scripting command. Examples of this are shown in Listing 4.6:

Listing 4.6 Examples of Python commands that create new Algorithms or Services

```
>>> theApp.ExtSvc += [  "ANewService" ]
>>> theApp.TopAlg  = [  "TopSequencer/Sequencer" ]
```

If the specified Algorihm or Service already exists then its properties can immediately be accessed. However, in the prototype the properties of newly created objects cannot be accessed until an equivalent Python object is also created. This restriction will be removed in a future release.

This synchronization mechanism for creation of Python Algorithms and Services is illustrated in Listing 4.7:

Listing 4.7 Examples of Python commands that create new Algorithms or Services

```
>>> theApp.ExtSvc += [  "ANewService" ]
>>> ANewService    = Service( "ANewService" )                            [ 1]
>>> theApp.TopAlg  = [  "TopSequencer/Sequencer" ]
>>> TopSequencer   = Algorithm( "TopSequencer" )                        [ 2]
>>> TopSequencer.properties()
```

Notes:

1. This creates a new Python object of type Sequencer, having the same name as the newly created Athena Sequencer.
2. This creates a new Python object of type Algorithm, having the same name as the newly created Athena Algorithm.

The Python commands that might require a subsequent synchronization are shown in Listing 4.8:

Listing 4.8 Examples of Python commands that might create new Algorithms or Services

```
theApp.ExtSvc            += [ ...]
theApp.TopAlg             = [ ...]
Sequencer.Members         = [ ...]
Sequencer.BranchMembers   = [ ...]
OutStream.AcceptAlgs      = [ ...]
OutStream.RequireAlgs     = [ ...]
OutStream.VetoAlgs        = [ ...]
```

4.8 Controlling job execution

This is very limited in the prototype, and will be replaced in a future release by the ability to call functions on the Python objects corresponding to the ApplicationMgr (theApp), Algorithms, and Services.

In the prototype, control is returned from the Python shell to the Athena environment by the command in Listing 4.9:

Listing 4.9 Python command to resume Athena execution

```
>>> theApp.run( nEvents )                                        [ 1]
```

Notes:

1. This is a temporary command that will be replaced in a future release by a more flexible ability to access more functions of the ApplicationMgr. "`nEvents`" is the number of events that should be processed.

This will cause the currently configured event loop to be executed, after which control will be returned to the Python shell.

Typing Ctrl-D (holding down the Ctrl key while striking the D key) at the Python shell prompt will cause an orderly termination of the job. Althernatively, the command shown in Listing 4.10 will also cause an orderly application termination.

Listing 4.10 Python command to terminate Athena execution

```
>>> theApp.exit()                                                    [ 1]
```

This command, used in conjunction with the theApp.run() command, can be used to execute a Python script in batch rather than interactive mode. This provides equivalent functionality to a job options text file, but using the Python syntax. An example of such a batch Python script is shown in Listing 4.11:

Listing 4.11 Python batch script

```
>>> theApp.TopAlg = [  "HelloWorld" ]
    [ other configuration commands]
>>> theApp.run( nEvents )
>>> theApp.exit()
```

Chapter 5
StoreGate - the event data access model

5.1 Overview

A more detailed version of this chapter is contained in the document The ATLAS Data Model User s Guide, which is available in the ATLAS CVS repository at:

offline/AtlasDoc/doc/DataModel

5.2 The Data Model Architecture

5.2.1 Data Objects and Algorithms

The Gaudi software architecture belongs to the blackboard family: data objects produced by knowledge modules (called Algorithms in Gaudi) are posted to a common "in-memory data base" from where other modules can access them and produce new data objects.

This model greatly reduces the coupling between knowledge modules containing the algorithmic code for analysis and reconstruction, in that one knowledge module does not need anymore to know which specific module can produce the information it needs nor which protocol it must use to obtain it (the "interface explosion" problem described in component software systems). Algorithmic code is known to be the least stable component of software systems and the blackboard approach has been very effective at reducing the impact of this instability, from the Zebra system of the FORTRAN days to the Java Data Objects architecture.

5.2.2 StoreGate: the Atlas Transient Data Store

The Transient Data Store (TDS) is the blackboard of the Gaudi architecture: an Algorithm creates a data object and post it onto the TDS to allow other Algorithms to access it[1].

Once an object is posted on to the store, the TDS takes ownership of it and manages its lifetime according to preset policies, removing, for example, a TrackCollection when a new event is read. The TDS also manages the conversion of a data object from/to its persistent form and provides therefore an API to access data stored on persistent media.

StoreGate (SG) is the Atlas implementation of the TDS. It manages the data objects in transient form, it steers their transient/persistent conversion and it provides a dictionary allowing to identify and retrieve data objects in memory. The SG design and implementation was largely driven by a few design concepts that are worth describing as a way of introduction.

5.2.2.1 Avoid User-defined Keys

The disadvantage of the data/knowledge objects separation is the need for knowledge objects to identify data objects to be posted on or retrieved from the blackboard. It is crucial to develop a data model optimized for the required access patterns and yet flexible enough to accommodate the unexpected ones.

SG addresses this problem with a two-step approach: it defines a natural identifier mechanism for data objects and it transparently associates to each data object a default value of this identifier allowing developers to register and retrieve data objects without having to identify them explicitly.

The first component of the identifier is the data object type. Experience shows that HEP developers tend to group the objects they work on into collections, most often STL vectors. As a result the TDS will often contain a single instance of a data object type (say a `TrackCollection` or several related ones (e.g. a `TrackCollection` for each component of the Inner Detector). The SG retrieve interface covers these two use cases

```
DataHandle<TrackCollection> theTrackColl; //STL forward_iterator
sg->retrieve(theTrackColl);  //get the (default) TrackCollection
DataHandle<TrackCollection> beginTrackColls, endTrackColls;
sg->retrieve(beginTrackColls, endTrackColls); //get all TrackColls
```

Type-based identification is not always sufficient. For example the TDS may contain several equivalent instances of a TrackCollection produced by alternative tracking algorithms. Therefore we need to add a second component to our identification mechanism: the identifier of the Algorithm instance that produced the data object we want[2].In the spirit of working with user types, the SG will allow developers to augment this history identifier with a generic key optimized for their access patterns.

1. To be precise the current TDS implements only a "passive" blackboard, since Algorithms do not (yet) react to TDS events (e.g. executing after a data object is registered into the TDS)

2. Notice that we need to identify the instance rather than the class. In an often quoted use case, clients may want to distinguish among tracks reconstructed by the same tracking algorithm using different jet-cone sizes.

5.2.2.2 Work with User Types

The success of the STL and of other public domain template libraries means that it has become vital to design an open system that can work with generic types that export an interface, in particular the STL containers, rather than forcing data objects to import a common interface. SG adapts its behavior to the functionality each data object exports. The only SG-imposed constraint on a data object[1] is to be an STL *Assignable* type.

5.2.2.3 Control Object Access and Creation

The TDS is the main channel of communication among modules. A data object is often the result of a collaboration among several modules. SG allows a module to use transparently a data object created by an upstream module or read from disk.

A *Virtual Proxy* defines and hides the cache-fault mechanism: upon request[2], a missing data object instance can be transparently created and added to the TDS, presumably retrieving it from a persistent data-base or, in principle, even reconstructing it on demand.

To ensure reproducibility of data processing, a data object should not be modified after it has been published to the store, the same handle/proxy scheme is used to enforce an ‘‘almost const’’ access policy: modules downstream of the publisher are only allowed to retrieve a constant iterator to the published object.

5.2.2.4 Support Inter-object Relationships

SG supports uni-directional inter-objects relationships, or links. A link is a persistable pointer. If the linked object is a data object then the handle/proxy mechanism described above is also used to implement the link. But typically links will refer to objects that are not data objects but are contained within a data object. The SG knows how to get to the container and the container knows how to return an element given its index. The job of the link is to find out the value of the index, persistify it and, later on, pass it on to the container and get back the linked object.

5.3 Data Objects

As we mentioned earlier SG is designed to work with user types rather than requiring them to implement a C++ interface. Basically any STL *Assignable* (i.e. any type which has an `operator =` and/or a copy constructor) can be stored into SG and hence is a Data Object.

1. this does not mean that the data model, simulation and reconstruction groups should not issue design guidelines to ensure that ATLAS data objects behave consistently in terms of memory management and persistability
2. Currently the proxy uses lazy instantiation (i.e. the object is created only when the handle is dereferenced).

A Data Object is a struct or class that encapsulates and "publishes" the result of some arbitrarily complex processing performed by one or more Algorithms. A Data Object should present a predictable, stable and efficient interface to client Algorithms.

A Data Object is often persistable and in this case the interface must be sufficient to allow a Converter to capture the Data Object state on disk and to restore it. The best advice to the Data Object designer is to keep them simple[1].

5.3.1 Using Containers as Data Objects

Experience shows that most Data Objects are containers (of hits, cells, tracks, muons,...). In particular STL containers are perfectly valid Data Objects and can be stored into SG.

The developer of a Data Object container must decide if the container they want to store is a *Value Container* or a *View Container* and then are they are strongly advised to use the tools and policies SG provides to implement them.

5.3.1.1 View Containers

A View Container is a container of object references. The referred-to objects are not owned by the View Container and will, in general, continue to exist after the View goes out of scope. As an example the list of cells which were used to reconstruct a photon is a View on the container(s) of reconstructed calorimeter cells. A View Container that does not need to be persistified can be implemented using plain C++ pointers, e.g. `std::list<const CaloCell*>`. A persistable view should be implemented using `DataLink`s.

5.3.1.2 Value Containers

A Value Container is a container that owns its elements "by-value": the elements cease to exist when the container does. For example the LAr cell recontruction may add the cells it makes to a `LArCellContainer` that is later recorded on SG. When a `LArCellContainer` goes out of scope all `LArCell`s it contains are deleted.

Whenever possible `LArCellContainer` should be implemented as a standard STL container of `LArCell` objects (e.g. as a `std::vector<LArCell>`).

Unfortunately this can not be done when `LArCell` is abstract: a polymorphic container (as containers of abstract elements are called) can only be implemented using STL as a container of pointers, e.g. `std::vector<LArCell*>`. But, as we mentioned in the previous section, a STL container of pointers is not a Value Container: it does not own its elements.

1. As a rule of thumb, if you need to include more than a couple of Atlas-specific header files to define a Data Object interface and its implementation, you should probably move some complexity out of it

To support polymorphic Value containers, SG provides two class templates, `DataList`, `DataVector` and a reference counted pointer `DataPointer` in terms of which the three containers are implemented:

```
std::vector< DataPtr<T> >   DataVector<T>
std::list< DataPtr<T> >     DataList<T>
std::map< DataPtr<K,T> >    DataMap<K,T>
```

`DataPtr<T>` behaves like a plain C++ pointer T* and can be freely assigned to and from a T*.

```
typedef DataVector<CaloCell> CellContainer;
// this is completely equivalent to
// typedef std::vector< DataPtr<CaloCell> > CellContainer;
CellContainer caloCells;
intV.reserve(nCells);
for (int i=0; i<nCells; ++i) {
    CaloCell* pThisCell = fillThisCell(i);
    caloCells.push_back(pThisCell);
}

.....

CellContainer::const_iterator it(caloCells.begin());
CellContainer::const_iterator iEnd(caloCells.end());
while (it!=iEnd) {
    const CaloCell& thisCell = **it;  //vector of pointers!
    std::cout << thisCell.energy() << std::endl;
    ++it;
}
```

when `CellContainer` goes out of scope, the vector destructor will delete the `DataPtr<CaloCell>`. In normal usage[1] this will trigger the deletion of the `CaloCell` instance as desired.

In summary to define a Value Container of instances of a concrete class use plain STL containers. Use containers of DataPtrs for polimorphic Value Containers. If you prefer using the shorthands DataVector and DataList please remember that a `DataVector<CaloCell>` has the semantics of a `std::vector<CaloCell*>` and not of `std::vector<CaloCell>`.

5.3.2 Describing Data Objects to SG

StoreGate uses a compact, technology-independent mechanism[2] to describe object types with two integer identifiers: a `CLID` and a `VERSION`. `CLID` is a 16-bit integer which uniquely identifies an object type across all Atlas software. The `CLID` of say `EventInfo` should not change from one

1. As always with ref-counted pointers care must be take to avoid cyclical dependencies (`a` points to `b` which points back to `a`).
2. Adapted from Gaudi Persistency Framework and capable of interacting with it.

release to another. If EventInfo changes in a non-backward compatible way a new VERSION number must be assigned to it[1].

SG provides a cpp preprocessor macro to define CLID and VERSION for a type

```
EventInfo.h:

class EventInfo {
 ...
}
#ifndef TOOLS_CLASSID_TRAITS_H
#include ``StoreGate/tools/ClassID_traits.h''
#endif
CLASS_DEF(EventInfo, 2101, 0)
```

Although the intention is to have them generated automatically using the Atlas Dictionary, as of release 3.1.0 these macros have to be explicitely added by each Data Object developer to the class header file, hence CLASS_DEF(EventInfo, 2101, 0) should be placed in EventInfo.h. For templated Data Objects (e.g. std::vector<Track>) or for "external" Data Objects {e.g. HepMC::Vertex) of which we can't modify the header file, we recommend adding the CLASS_DEF macros into a separate header file per Data Object package.

```
MyPackage_ClassDefs.h:

#ifndef TOOLS_CLASSID_TRAITS_H
#include ``StoreGate/tools/ClassID_traits.h''
#endif
CLASS_DEF(std::list<MyContObj>, 8003, 1)
CLASS_DEF(std::vector< DataPtr<MyAbstractCell> >, 8004, 1)
```

5.3.3 Data Object Creation and Ownership of Data Objects

Data Objects must be created on the heap using the new command:

```
DataVector<LArCell> *pCells = new DataVector<LArCell>;
```

Data Objects recorded to SG are owned by SG and the creator must not delete them.

5.4 Accessing Data Objects

This section is a tutorial on how to use StoreGate to access DataObjects from user Algorithms. The examples in it are based on the ones in the AthenaExamples/AthExStoreGateExample Tutorial package.

1. As of release 3.1.0 the VERSION number is not yet used by SG.

As a preliminary, each user algorithm must locate the relevant StoreGateSvc instances in its initialize method. For example:

```
StatusCode SGRead::initialize()
{
  StatusCode sc;
   ....
  //locate event store pointer and cache it in a SGread data member
  sc = service("StoreGateSvc", p_eventStore);

  if (sc.isFailure()) {
      log << MSG::ERROR
  << "Unable to retrieve pointer to Event StoreGateSvc"
  << endreq;
      return sc;
  }
  //locate detector store pointer and cache it in a SGread data member
  sc = service(``DetectorStore", p_detectorStore);
  if (sc.isFailure()) {
      log << MSG::ERROR
  << "Unable to retrieve pointer to Detector StoreGateSvc"
  << endreq;
      return sc;
  }
```

5.4.1 Recording a Data Object

To record a Data Object we must provide StoreGateSvc with a pointer to the Data Object created on the heap[1], and, *optionally* with a *key*.

```
StatusCode SGWrite::execute() {
   ...
  MyDataObj *pdobj = new MyDataObj;      // Create a DataObject
  pdobj->val(42);                        // Set its internal state
   ...
  StatusCode sc = p_eventStore->record(pdobj, dataObjKey);
  if ( sc.isFailure() )
  {
    log << MSG::ERROR
<< " could not register object " << dataObjKey
<< endreq;
    return StatusCode::FAILURE;
  }
   ...
}
```

1. i.e. using the operator `new`

here `pdobj` is the pointer to a `MyDataObj` created on the heap (i.e. using `new`), and `dataObjKey` is a reference to a valid key object. Usually a key is an instance of `std::string` but any class that can be converted to and from a `string` can be used as key[1]. The combination of type and key are the SG object identifier and they must be *unique*.

Once a Data Object has been recorded, SG takes ownership of it. *Never delete a recorded Data Object.*

5.4.1.1 Locking a Data Object

We strongly recommend to lock a Data Object once it is ready to be used by client algorithms. The preferred way to achieve this is to add a bool flag to the `record` invocation:

```
static const bool ALLOWMODS(false);
StatusCode sc = p_eventStore->record(pdobj, dataObjKey, ALLOWMODS);
```

Once a Data Object has been locked, downstream clients will not be able to modify its contents (see next section).

One can also lock an already recorded object using

```
StatusCode sc = p_eventStore->setConst(pdobj);
```

where `pdobj` is the pointer to the Data Object in memory.

We are considering to enforce Data Object locking in a forthcoming release, by disallowing to write out "non-const" Data Objects.

5.4.2 Retrieving a Data Object

Data Objects in SG are retrieved by type. SG sets a pointer to the requested object(s) of a given type.

5.4.2.1 Retrieving the default instance of a given type

SG defined the default DataObject of a given type as the last one `record`ed. To retrieve the default instance, one passes a pointer to the Data Object to SG "keyless" *retrieve* method to set

```
  const DataVector<MyElement> *pcoll(0);
  if (m_eventStore->retrieve(pcoll).isSuccess()) {
//use pcoll
  } else {
    log << MSG::ERROR
        << "can't retrieve default DataVector<MyElement>" << endreq;
    return StatusCode::FAILURE;
```

1. In the near future (release 5.x?) we plan to allow to use integers as keys. Eventually any type that can be hashed into an integer will be usable as a key.

```
}
```

Notice that you can retrieve a Data Object which has not yet been read from disk: SG will message the persistency service and the appropriate converter[1] will create the Data Object.

5.4.2.2 Retrieving a keyed instance of a given type

```
const MyDataObj *pdobj(0);
if (m_eventStore->retrieve(pdobj, m_DataObjKey).isSuccess()) {...}
```

Since two Data Objects of the same type can not have the same key, you are assured of being returned a unique Data Object (provided of course it exists).

5.4.2.3 Retrieving a Data Object to modify it

If a Data Object has not yet been locked it is possible to modify it passing to `retrieve` a (non-const) pointer that SG will set

```
DataVector<MyElement> *pcoll(0);
if (m_eventStore->retrieve(pcoll).isSuccess()) {
   collHandle->push_back(new MyElement(....));
} else { ... }
```

if the `DataVector` has been already locked the `retrieve` will fail.

Once again, SG owns all stored Data Objects: never `delete` a Data Object using the pointer set by `retrieve`.

5.4.2.4 Retrieving {\em all} instances of a given type

To retrieve all instances of MyDataObj in the store, create two `DataHandle<MyDataObj>` and let `record` set them

```
    DataHandle<MyDataObj> dbegin;
    DataHandle<MyDataObj> dend;
    StatusCode sc = storeGateSvc()->retrieve(dbegin, dend);
    if (sc.isFailure())
    {
       log << MSG::ERROR << "Error Retrieving MyDataObj's" << endreq;
    }
\end{verbatim}
DataHandle is a standard forward iterator[2]: the pair{\tt dbegin, dend} allows to
iterate over all stored instances of {\tt
```

1. SG mantains a list of DataProxy objects, each one of these managing the life-cycle of the Data Object they represent. Besides containing the transient key of a Data Object and its CLID, a DataProxy has a pointer to the OpaqueAddress of its Data Object which is passed to the appropriate converter when the Data Object has to be created \ref{UG}.

```
MyDataObj}
\begin{ verbatim}
    while(dbegin != dend)    // Loop Over \dobjs
    {
      dbegin->do_something(); //read MyDataObj if needed
      ++dbegin;
    }
```

notice that the Data Object pointed to by a DataHandle will not be accessed (and read from disk if necessary) until the DataHandle is dereferenced.

5.4.2.5 Checking if a Data Object is in the store

SG provides two `contains` method sthat allow to check whether a given Data Object has already been stored. The typical use case is as follows

```
if (!p_SG->contains<MyDataObject>(myKey)) {
   MyDataObject* pMDO = createMDO();
   if(!p_SG.record(pMDO, myKey).isSuccess()) { return StatusCode::FAILURE;}
}
```

as usual the "keyless" version of `contains` is also provided to check whether any Data Object of a given type has been stored.

5.5 Using DataLinks to persistify references

In C++ we describe associations among objects using pointers or, less frequently, references. For example, a cluster object may refer to its list of associated cells by holding a vector of Cell pointers

```
class Cluster {
  ...
 private:
     ...
   std::vector<Cell*> m_myCells;
};
```

Unfortunately a plain C++ pointer can not be simply written out and read back from disk as is: it is valid only within the context of a running job.

To address this limitation we introduced `DataLink` and `EementLink`, two class templates which can be dereferenced like a pointer and can be read and written using various persistency mechanisms. The `DataLink` template allows to point to a data object, using its unique type/key combination.

2. Hence {\tt dend} points past the end of the list of returned objects: dereferencing {\tt dend} will have unpredictable, but most likely fatal, results.

ElementLink is used to point to an element of a container recorded in StoreGate. ElementLink, by default, allows to link to elements of any STL-derived sequence (e.g. std::vector and std::list, but also DataVector and DataList).

5.5.1 Creating a DataLink to a data object

DataLinks can be set to refer to a data object either by providing a pointer to the data object or its StoreGate identifier (type/key).

5.5.1.1 Construct a DataLink using a C++ pointer

Most often the developer filling a DataLink will have a C++ pointer to the data object they want to create an association to. For example let's consider the PileUpEventInfo class. It carries a list of references to the EventInfo data objects of the physics and background events used in the overlay process:

```
class PileUpEventInfo {
  ...
 private:
  DataLink<EventInfo> m_origEvent;
  std::list<DataLink<EventInfo> > m_subEvents;
};
  ...
}
```

we can create a DataLink<EventInfo> from an EventInfo* and add it to m_subEvents

```
  std:list<EventInfo*> pSubEvts;
  ... fill pSubEvts ...
  for (int i=0, i<nSubEvts, i++) {
     m_subEvents.push_back(DataLink<EventInfo>(*pSubEvts[ i]));
  }
```

Of course the pointer used to set the DataLink<EventInfo> must refer to an EventInfo which has been or will be recorded into StoreGate. Both for convenience and for efficiency reasons, DataLink will not look-up the data object is pointing to, until the DataLink itself has to be persistified, or until the user invokes

```
DataLink<{ ...}>::dataID()
```

At that point, if the data object is not found in the store an exception will be thrown.

5.5.1.2 Construct a DataLink using its StoreGate Key

If you know the key identifying a data object in the store you can also use it to create a link to it:

```
  DataLink<EventInfo> m_origEvent;
```

```
...
 m_origEvent.toStorableObject(``PhysicsTDR'');
...
```

5.5.2 Creating a Link to an Element of a Container

Quite often the linked object is not a data object but an element of a data object (typically a STL container). In the example of the `Cluster` class the linked objects are `Cell` objects contained in a `DataVector<CaloCell>`, which behaves like a `std::vector<CaloCell*>` and hence it is an example of STL Sequence. To make the `Cluster` class above persistable, we replace the `Cell` pointers with `ElementLink`s:

```
class Cluster {
   typedef DataVector<CaloCell> CaloCellContainer;
   std::vector< ElementLink<CaloCellContainer> > m_myCells;

   void addCells() {
     const CaloCellContainer* pCont;
     if ((p_eventStore->retrieve(pCont)).isSuccess()) {
unsigned int nCells(pCont->size());
for (unsigned int iCell=0; iCell<nCells; ++iCell) {
           if (weLikeThisCell((*pCont)[ iCell])) {
              ElementLink<CaloCellContainer> linkCell;
              linkCell.toIndexedElement(*pCell, iCell);
              m_myCells.push_back(linkCell);
           }
        }
     }
   }
};
```

Please notice the difference between `DataLink<CaloCellContainer>`, a pointer to a `CaloCellContainer`, and `ElementLink<CaloCellContainer>` which behaves like a pointer to an element of a `CaloCellContainer` (hence like a `CaloCell**`).

5.5.3 ElementLinks to other Containers

The header file `StoreGate/tools/DeclareIndexingPolicy.h` provides two macros that "inform" SG that a given container is (or behaves like) an `std::map` or `std::set`. For example

```
in MyHitMap.h
typedef std::map<Identifier32, LArHit*> MyHitMap;
#include ``StoreGate/tools/DeclareIndexingPolicy''
CONTAINER_IS_MAP( MyHitMap );

in MyUniqueInts.h
typedef std::set<unsigned int> MyUniqueInts;
#include ``StoreGate/tools/DeclareIndexingPolicy''
```

```
 CONTAINER_IS_SET( MyUniqueInts );
```

although it should be redundant, for completeness `DeclareIndexingPolicy` also provides a `CONTAINER_IS_SEQUENCE` macro.

Advanced developers can also specialize `ElementLink` to link to elements of arbitrary containers

```
from GeneratorObjects/McEventLinks.h
 typedef ElementLink<McEventCollection,  \\target container
                     DataProxyStorage<McEventCollection>, \\default
                     GenParticleIndexing> \\HepMC indexing policy
         GenParticleLink
```

A description of how to define a specialized indexing policy like `GenParticleLink` is beyond the scope of this document. You are welcome to contact the authors for help.

5.5.4 Accessing DataLinks

`DataLinks` are dereferenced as pointers: in the `Cluster` class above you can

```
void Cluster::workWithCells() {
  ...
  const Cell& cell33 = *(m_myCells[ 33]);
  ...
  unsigned int i(0), nCells(m_myCells.size());
  while(i<nCells) clusterRawEnergy += (**(m_myCells[ i++])).energy();
}
```

notice that, unlike `DataHandles` and plain pointers, `DataLink` is not an iterator: you can not increment it or perform any "pointer-arithmetic" on it.

Notice also how the `ElementLink<CaloCellContainer>` must be dereferenced **twice**: the `ElementLink` is a pointer to an element of a `vector` of `CaloCell*`, hence it is equivalent to a `CaloCell**`.

5.5.5 DataLinks Persistency

For stream-based persistency (e.g. root, Gaudi generic converters or plain files) SG provides templated inserter (`operator >>`) and extractor (`operator <<`) operators:

```
void Cluster::Streamer(TBuffer& tbuf) {
  typedef ElementLink< std::vector< DataPtr<CaloCell> > > CellLink_t;
  ...
  if (tbuf.IsReading()) {
    std::vector<CellLink_t>::size_type i(0), nCells(0);
    tbuf >> nCells;
    for(;i<nCells;++i) {
       CellLink_t inLink;
```

```
        tBuf >> inLink;
        m_myCells.push_back(inLink);
      }
      ...
    } else if (tbuf.IsWriting()) {
      std::vector<CellLink_t>::size_type i(0), nCells(m_myCells.size());
      tbuf<< nCells;
      while(i<nCells) tBuf << m_myCells[ i++] ;
      ...
    }
  }
```

To support other persistency technologies, `DataLink` provides a method[1]:

```
//StoragePolicy method returning the data object key
const ID_type& dataID() throw(std::logic_error); //ID_type is a string
```

`ElementLink` provides also another method that returns the index of the element inside the container (this is e.g. an `unsigned int` for a `Sequence` and the `key_type` for a `std::map`)

```
//IndexingPolicy method returning the element index inside the data
//object container.
index_type index() const { return m_key;}
//index_type is unsigned int for sequences, key_type for maps and sets
```

5.6 History

This section is incomplete.

1. actually inherited from its StoragePolicy

Chapter 6
Data dictionary

6.1 Overview

One of the central components of Athena is the data dictionary that is used both for coupling together the C++ and Python (scripting) environments, but also as part of the auto-generation of persistency converters for elements of the Event Data Model.

6.2 How to write/read data via POOL

There are basically three steps needed to be able to work with POOL:

1. The objects to be stored in POOL must be "described" in a data dictionary which exists in memory when a program runs. As far as POOL is concerned, this dictionary contains the description of the types of all attributes for the objects to be stored. What is meant by description is just the size and position of an attribute. Other applications, e.g. python scripting, will also need to have methods described in order to be able to call them.

   This description is built by creating a data dictionary filler, which is just a library that when loaded into memory will fill the memory resident dictionary with descriptions for a set of classes.

   Conventions:

   Where to create the dictionaries: Each XXXEvent, YYYDetDescr, ZZZConditions package defining data objects to be stored will create its own dictionary filler library by applying CMT lcgdict pattern. There are exceptions to this rule for packages that do not include framework-related dependencies. For example, there is a DetDescrDictionary package for defining the Identifier-related classes.

   How to use classes defined in different dictionaries: Each package should only "describe" classes that it contains. (This is done in a selection file.) To "use" a description in another library, one simply needs to "load" the other library. This "loading" of the dictionaries is temporarily done by explicitly linking the converters (see next step) to the dictionary filler libs, so that one only specifies to load a particular converter. This must be specified by hand for the converters. An improved automation will eventually be deployed so that this linking will no longer be required.

2. Objects are written and read to POOL via converters of the AthenaPoolCnvSvc. For most objects generic converters are sufficient, thus we have provided a CMT pattern which can be applied. (See generating converters.) However there are situations where the converters need to be customized, for example, to set the values of transient detector description pointers when event objects are read in. (See writing custom converters.)

   Conventions:

   As opposed to the dictionary fillers which are generated in the data packages, the POOL converters are grouped together into separate packages according to subsystem, reconstruction, etc. For example packages that exist today are:

   EventAthenaPool

   RecAthenaPool

   InDetEventAthenaPool

   MuonEventAthenaPool

3. Finally, one needs to specify the job options for reading and writing. This is described in setting up the joboptions.

Last but not least, we maintain a changing list of caveats, problems and work-arounds which hopefully diminish as Atlas and POOL software improves.

6.2.1 Creating a data dictionary filler

To explain how to generate a dictionary filler for a data package, we take as an example the SimpleTrack package. The basic procedure is to use the cmt pattern "lcgdict" to generate a dictionary for a package with data objects. This creates a shared library which is dynamically loadable. There is also a small job option file generated which adds the library name to a list of libs to be loaded by the AthenaSealSvc. If this job option file is included from one's application joboptions, then AthenaSealSvc will load the lib at initialization time.

Note that a corresponding pattern, poolcnv, which today is in the AtlasPOOL package, is used to generate pool converters. This is described in the AtlasPoolUtilities package.

The procedure:

1. Use the SimpleTrack packageas an example. The relevant portion of the requirements file is shown in Listing 6.1. You will need to add a `use` to `AtlasSEAL` and apply the pattern `lcgdict`. To do the latter, you need to create a `<package>Dict.h` file and a `selection.xml` file.

Listing 6.1 Package SimpleTrack Requirements file

```
package SimpleTrack

author  Laurent Vacavant <Laurent.Vacavant@cern.ch>

use AtlasPolicy AtlasPolicy-01-*
use DataModel DataModel-00-* Control
use CLIDSvc CLIDSvc-00-* Control

library SimpleTrack *.cxx

apply_pattern installed_library

private
use AtlasSEAL   AtlasSEAL-00-*   External -no_auto_imports

# Pattern to build the dict lib. User should create a single header
# file: <package>Dict.h which includes all other .h files. See MissingETDict
# A selection file must be created by hand. This file lists the
# classes to be added to the dictionary, and which fields are
# transient. It should be put in ../<package> dir and is conventionally
called
# selection.xml.
apply_pattern lcgdict dict=SimpleTrack selectionfile=selection.xml
headerfiles="../SimpleTrack/SimpleTrackDict.h"
```

2. Create a single `<package>Dict.h` which just includes the other header (.h) files. An example is shown in Listing 6.2:

Listing 6.2 Example <package>Dict.h file

```
#ifndef SIMPLETRACK_SIMPLETRACKDICT_H
#define SIMPLETRACK_SIMPLETRACKDICT_H

#include "SimpleTrack/SimpleTrackCollection.h"

#endif
```

This creates a single C++ file to compile and avoids multiple definitions of symbols in the lib which may arise if each .h is listed separately. Note that the file `<package>Dict.h` is given as an argument for the `lcgdict` pattern (see Listing 6.1).

3. Create a `selection.xml` file in the header directory and specify it as the `selectionfile` argument to the `lcgdict` pattern. The selection file contains a list of classes for each of the data member types. An example of this is shown in Listing 6.3.

Listing 6.3 Example selection.xml file

```
<lcgdict>
  <class name="SimpleTrackCollection"
id="9E3595D6-1362-429A-8BA2-3396C93D6BA0" />
  <class name="SimpleTrack" />
  <class name="DataVector<SimpleTrack>" />
  <class name="std::vector<SimpleTrack*>" />
</lcgdict>
```

4. Not all data members are intended to be written out. For example, a class might have a pointer to an object that is only used in transient memory. If the value is not intended to be written out, then one should declare it transient, e.g. for the `EventInfo` class has a pointer to subEvent for pileup which is only needed in memory. One declares this in the selection file as illustrated in Listing 6.4.

Listing 6.4 Specifying data member overrides

```
 <class name="PileUpEventInfo::SubEvent" >
    <field name="pSubEvtSG" transient="true" />
  </class>
```

One should make sure that the default constructor sets this to a reasonable value for objects that are read back in. (Note: more sophisticated initialization of transient members will be addressed in a future version of this documenation.)

5. You must specify the an "id" for all persistent data objects. (NOTE: we expect that this requirement to create an id will eventually be removed and no id will be needed.) For example,

```
    <class name="SimpleTrackCollection"
id="9E3595D6-1362-429A-8BA2-3396C93D6BA0" />
```

For each "data object" with a CLID, an id number must be added as an "id" attribute. This area is "changing" in POOL. At the moment, POOL 1.2.0, one must use a universally unique identifier (UUID) which can be obtained with

```
 > uuidgen
5089b086-8b04-4696-a254-f5ce380f536e
```

and the resulting number is copied and made into uppercase.

6. To check that the selection file is ok, you should run a check with the AthenaSealSvc which loads the SimpleTrackDict and checks all fields to see that their type is defined.

Listing 6.5 Example of checking the selection file

```
In TestRelease req:

use AthenaCommon AthenaCommon-* Control
use AthenaSealSvc AthenaSealSvc-* Control
# Need something, e.g. the following, to pull in libT_Histoxx.so
use TestEvent TestEvent-* Event
```

do "source setup.sh" and "gmake" and modify Load_AthenaSealSvc_joboptions.txt which you will get in your run directory.

Listing 6.6 Job Options file

```
//
// JobOptions for the loading of the AthenaSealSvc
//
#include "$ATHENASEALSVCROOT/share/AthenaSealSvc_joboptions.txt"

#include "$DETDESCRDICTIONARYROOT/dict/DetDescrDictionary_joboptions.txt"
#include "$SIMPLETRACKROOT/dict/SimpleTrack_joboptions.txt"

// Set to output level to debug for more information
//MessageSvc.OutputLevel      = 2;

// Check the dictionary in memory for completeness
//AthenaSealSvc.CheckDictionary = true;
```

Uncomment "AthenaSealSvc.CheckDictionary = true;", set the output level to 2 and add in includes for any other dictionary fillers that you want to check, or that your dictionary needs, e.g. has inherited or embedded types. NOTE THE .txt IS IN /dict/!!! Note here we have added the DetDescrDictionary for the Identifier classes and SimpleTrack.

This prints out the classes and their fields and lists the classes missing (Listing 6.7).

Listing 6.7 Example session to check for missing classes

```
> athena.exe Load_AthenaSealSvc_joboptions.txt
...
AthenaSealSvc  INFO
AthenaSealSvc  INFO Checking the completeness of the dictionary for all
classes
AthenaSealSvc  INFO
AthenaSealSvc  INFO Checking fields of class DataVector<Pixel1RawData>: ok
AthenaSealSvc  DEBUG Fields of class DataVector<Pixel1RawData>:
AthenaSealSvc  DEBUG     m_ownPolicy          -- offset:   4 -- type: int
AthenaSealSvc  DEBUG     m_pCont              -- offset:   8 -- type:
std::vector<Pixel1RawData*>
AthenaSealSvc  DEBUG
AthenaSealSvc  INFO Checking fields of class DataVector<PixelRDORawData>: ok
AthenaSealSvc  INFO Checking fields of class DataVector<SimpleTrack>: ok
AthenaSealSvc  DEBUG Fields of class DataVector<SimpleTrack>:
AthenaSealSvc  DEBUG     m_ownPolicy          -- offset:   4 -- type: int
AthenaSealSvc  DEBUG     m_pCont              -- offset:   8 -- type:
std::vector<SimpleTrack*>
...
AthenaSealSvc  INFO Checking fields of class SimpleTrack: ok
AthenaSealSvc  DEBUG Fields of class SimpleTrack:
AthenaSealSvc  DEBUG     m_A0Vert             -- offset:  40 -- type: double
AthenaSealSvc  DEBUG     m_BarEnd             -- offset: 256 -- type: double
AthenaSealSvc  DEBUG     m_BremRadius         -- offset: 384 -- type: double
AthenaSealSvc  DEBUG     m_Chi2               -- offset:  32 -- type: double
AthenaSealSvc  DEBUG     m_CotThEnd           -- offset: 240 -- type: double
...
AthenaSealSvc  INFO ----->  NO Missing fields!!
```

This is the output when all fields have been defined. If there is something missing, there will be a message. For example, if the selection file line:

```
<class name="std::vector<SimpleTrack*>" />
```

is missing, one will get:

Listing 6.8 Example session to check for missing classes

```
AthenaSealSvc  INFO Checking fields of class SimpleTrackCollection:
AthenaSealSvc  INFO ****> Missing type for DataVector<SimpleTrack>  m_pCont
AthenaSealSvc  DEBUG Fields of class SimpleTrackCollection:
AthenaSealSvc  DEBUG     m_ownPolicy          -- offset:   4 -- type: int
AthenaSealSvc  DEBUG     m_pCont              -- offset:   8 -- type:
[ unknown]
```

where the attribute with a missing type is listed with type "[unknown]".

Note that attributes declared as transient will have "[unknown - declared transient]".

You can add to the selection file and iterate until all problems are resolved. (It may be useful to look in the dictionary filler cpp file to find the right class name to use, e.g. in ../dict/EventInfo/EventInfoDict_dict.cpp one will see that "std::basic_string" is needed for std::string.)

Any job option that need dict files should have:

Listing 6.9 Example session to check for missing classes

```
#include "$ATHENASEALSVCROOT/share/AthenaSealSvc_joboptions.txt"

#include "$EVENTINFOROOT/dict/EventInfo_joboptions.txt"
... (for each new dict)
```

These just add to the list of AthenaSealSvc, and this service loads the dict libs.

6.2.2 generating converters

Pool converters are automatically generated using the CMT `poolcnv` pattern. By convention, the generation is done in a single package for a number of classes. For example, the Reconstruction classes are done in the Reconstruction/RecAthenaPool package. We use this package as an example.

Rule 1: each class must be declared in a separate .h file. For example, `MissingETEvent/MissingET.h` defines the `MissingET` class and `SimpleTrack/SimpleTrackCollection.h` defines the `SimpleTrack` collection.

In the requirements file of converter package one has:

Listing 6.10 Convert package requirements file

```
package RecAthenaPool

author David Rousseau <rousseau@lal.in2p3.fr>

use AtlasPolicy            AtlasPolicy-01-*
use AthenaPoolUtilities    AthenaPoolUtilities-00-*    Database/AthenaPOOL
use MissingETEvent         MissingETEvent-00-*         Reconstruction
use SimpleTrack            SimpleTrack-00-*            Reconstruction

# temporarily add in explicit link to dictionary
macro_append RecAthenaPool_linkopts " -lSimpleTrackDict -lMissingETEventDict
"

apply_pattern poolcnv files="-s=${MissingETEvent_root}/MissingETEvent
MissingET.h
                             -s=${SimpleTrack_root}/SimpleTrack
SimpleTrackCollection.h "
```

The "`use AthenaPoolUtilities`" is needed to get the `poolcnv` pattern. The other uses should refer to packages with data objects which need converters.

One should provide the list of header files for the pattern "`poolcnv`". These can be taken from a number of packages following the syntax of "`-s=${<package>_root}/<package> <hdr1> <hdr2>`" which can be repeated.

Finally, one should temporarily link against the dictionary libraries containing the object descriptions needed by the converter. This is done by adding the library names in the list of `<package>_linkopts` as seen above. Note that this will be automated in the future so that the dictionaries will be loaded when needed. When this happens the `<package>_linkopts` will need to be removed.

6.2.3 writing custom converters

6.2.3.1 when to use custom converters

There are some situations where one needs to write a custom converter for a class which we divide into two categories:

1. Modifying objects being read/written: For example when one declares attributes as transient in the selection.xml file they are not written out. When reading in the default constructor is called which may initialize the transient attributes. However, if one needs to set the transient attributes externally, e.g. with DetDescr information, after an object is read in, then a custom converter is needed.
2. Fine control over what is written: In some situation, one needs fine control over the I/O. One example is the case of the InDetRawDataContainers for the InDet RDOs. These containers group RDOs into RDO collections, which are in turn stored in the InDetRawDataContainers. These containers have special behavior when reading in from the byte-stream - collections are converted on-demand as clients request them from the container. However for pool, one is interested in a bulk read/write of the container and not interested in writing collections one by one. Custom converters are used here to read/write the containers and as well to initialize them with their required id helper.

For both of these situations one will generate a converter skeleton. The difference will be in the level of modification applied.

6.2.3.2 generate custom converter skeletons

Writing custom converters uses and extends the generated converters described above. To start, one generates these converters to use as skeletons. The procedure is:

1. Specify the header file, e.g. MyClass.h, for which one wants a converter as described in generating converters in a converter package.
2. Run gmake ONCE in the converter package. This will generate two files MyClassCnv.h and MyClassCnv.cxx in the ../pool directory of the package. You should move these files to the ../src directory, modify them as described below and save them in the cvs repository. Once there are MyClassCnv.h and MyClassCnv.cxx files in the src directory, a subsequent gmake should NOT regenerate these files in ../pool, rather the ones in the src will be used. You should Finally, keep MyClass.h in the poolcnv pattern because it is needed for building the component library.

6.2.3.3 customizing the converter skeletons

Newly generated files are quite simple:

Listing 6.11 Converter skeletons

```
MyClassCnv.h:

#ifndef MyClassCnv_H
#define MyClassCnv_H

#include "AthenaPoolCnvSvc/T_AthenaPoolCnv.h"
#include "MyPackage/MyClass.h"

typedef T_AthenaPoolCnv<MyClass> MyClassCnv;

#endif

MyClassCnv.cxx:

#include "MyClassCnv.h"
```

The basic procedure for customization is to derive from the generic templated converter T_AthenaPoolCnv<MyClass>. To do so, rename the typedef to MyClassCnvBase and derive from this class:

Listing 6.12 Customized converter

```
MyClassCnv.h:

//...

// We rename generated typedef to <converter>CnvBase
typedef T_AthenaPoolCnv<MyClassCnv> MyClassCnvBase;

/**
 ** Create derived converter to customize the saving of MyClass
 **/
class MyClassCnv : public MyClassCnvBase
{
   friend class CnvFactory<MyClassCnv >;
public:
    MyClassCnv(ISvcLocator* svcloc);
    virtual ~MyClassCnv();

    /// initialization
    virtual StatusCode initialize();

    /// Extend base-class conversion method to modify when reading in
    virtual StatusCode     PoolToDataObject(DataObject*& pObj,const
std::string &token);

private:
    /// For your private attributes

};
```

Here we have added the constructor, destructor, initialize and PoolToDataObject which are probably the minimal changes needed to set transient attributes when reading objects from pool. initialize can be used to access, for example, StoreGate.

The `PoolToDataObject` method should be implemented as:

Listing 6.13 Customized converter

```
MyClassCnv.cxx:
#include "MyClassCnv.h"
#include "GaudiKernel/MsgStream.h"
#include "StoreGate/StoreGateSvc.h"
#include "SGTools/StorableConversions.h"
// Constructor - call base constructor and initialize local attributes
MyClassCnv::MyClassCnv(ISvcLocator* svcloc) :
    // Base class constructor
    LArCellContainerCnvBase::T_AthenaPoolCnv(svcloc){}
MyClassCnv::~MyClassCnv(){}
StatusCode MyClassCnv::initialize() {
    AthenaPoolConverter::initialize(); // Call base clase initialize
    // Get the messaging service, print where you are
    MsgStream log(msgSvc(), "MyClassCnv");
    log << MSG::INFO << "initialize()" << endreq;
    // get DetectorStore service - if needed
    StoreGateSvc *detStore;
    StatusCode sc=service("DetectorStore",detStore);
    if (sc.isFailure()) {
        log << MSG::FATAL << "DetectorStore service not found !" << endreq;
        return StatusCode::FAILURE;
    } else {
        log << MSG::DEBUG << " Found DetectorStore " << endreq;
    }
    // Get objects from the detector store
    // ...
    log << MSG::DEBUG << "Converter initialized" << endreq;
    return StatusCode::SUCCESS;
}
StatusCode MyClassCnv::PoolToDataObject(DataObject*& pObj,const std::string
&token) {
    // First call base class converter to get DataObject from
    // pool. Then modify as appropriate
    MsgStream log(messageService(), "MyClassCnv::PoolToDataObject" );
    StatusCode sc = MyClassCnvBase::PoolToDataObject(pObj, token);
    if (sc.isFailure()) {
        log << MSG::FATAL << "Unable to get object from pool" << endreq;
        return StatusCode::FAILURE;
    } else {
        log << MSG::DEBUG << " Found DataObject " << endreq;
    }
    // Convert DataObject pointer to MyClass*
    MyClass* obj=0;
    SG::fromStorable(pObj, obj );
    if(!obj) {
        log << MSG::ERROR << "  failed to cast to MyClass " << endreq ;
        return StatusCode::FAILURE;
    }
    // Initialize MyClass
    // ...
    return StatusCode::SUCCESS;
}
```

For more extensive customization for the converters, one may similarly modify an object before written out by implementing:

MyClassCnv.h:

```
    /// Extend base-class conversion method for writing
    virtual StatusCode     DataObjectToPool(DataObject*  pObj,std::string
tname);
```

One may also write out an object of a completely different type than MyClass. This is possible as long as the new class is defined in a Seal Dictionary. But because of the change in type, one will also have to implement the two addition methods:

```
MyClassCnv.h:
    /// Must redefine placement according to type that is stored
    virtual void           setPlacement();

    /// class ID
    static const CLID& classID();
```

where

```
MyClassCnv.cxx:
const CLID& MyClassCnv::classID()
{  return ClassID_traits< MyClass >::ID() ; }
```

must return the CLID of MyClass. For a detailed example, see the InDet RDO converters, e.g. PixelRDO_Container, which are in the InDetAthenaPool package.

6.2.3.4 detailed custom converter examples

For the simple case of initializing objects being read in have a look at the LArCell/LArCellContainer example:

- LArCalorimeter/LArRecEvent - defines the dictionary for LArCell and container
- LArCalorimeter/LArCnv/LArAthenaPool - contains the custom converter
- AtlasTest/DatabaseTest/AthenaPoolTest - contains a simple write/read example which creates dummy LArCells and checks that the same one can be read back. JobOptions: LArCellContWriter_jobOptions.txt and LArCellContReader_jobOptions.txt

The InDet RDOs provide an example where one writes/reads a DataVector instead of an IdentifiableContainer. The custom converter simply copies the RDO collections between the IdentifiableContainer and DataVector before write and after read.

- InnerDetector/InDetRawEvent/InDetRawData - defines the dictionary for the RDOs, collections, containers and extra DataVector
- InnerDetector/InDetEventCnv/InDetEventAthenaPool - contains the custom converters

¥ AtlasTest/DatabaseTest/AthenaPoolTest - contains a simple write/read example which creates dummy RDOs and their collections and checks that the same one can be read back. JobOptions: InDetRawDataWriter_jobOptions.txt and InDetRawDataReader_jobOptions.txt

6.2.4 setting up the joboptions

6.2.4.1 To write out data objects to POOL

To illustrate the job options for writing, we use RecExCommon_jobOptions.txt as an example:

Listing 6.14 RecExCommon_jobOptions.txt

```
//---------------------------------------------------------------
// now write out Transient Event Store content in POOL
//---------------------------------------------------------------
//

#include "AthenaPoolCnvSvc/WriteAthenaPool_jobOptions.txt"

// check dictionary
#include "$ATHENASEALSVCROOT/share/AthenaSealSvc_joboptions.txt"
AthenaSealSvc.CheckDictionary = true;

// Define the output Db parameters (the default value are shown)
PoolSvc.Output    = "SimplePoolFile.root";
// PoolSvc.DbServer  =  "db1.usatlas.bnl.gov";
// PoolSvc.DbAccount =  "athena";
// PoolSvc.DbPassword = "";
// PoolSvc.DbType    = "mysql";
// PoolSvc.ConnectionType = "MySQLCollection";
// PoolSvc.FullConnection =
"mysql://athena:insider@db1.usatlas.bnl.gov/pool_collection";
PoolSvc.DbType     = "root";  // to define ROOT file resident collection
PoolSvc.Collection = "NewPoolTry";

// Converters:
#include "EventAthenaPool/EventAthenaPool_joboptions.txt"
#include "RecAthenaPool/RecAthenaPool_joboptions.txt"

// list of output objects key
// MissingET
Stream1.ItemList+={"3052#*"};

// EventInfo
Stream1.ItemList+={"2101#*"};

// SimpleTrackCollection
Stream1.ItemList+={"10003101#*"};

//---------------------------------------------------------------------
// switch off the writing
//ApplicationMgr.OutStream = { };
```

6.2.4.2 To read back data objects from POOL

To illustrate the job options for reading, we use RecExCommon_read_jobOptions.txt as an example:

Listing 6.15 RecExCommon_jobOptions.txt

```
//--------------------------------------------------------------
// Load POOL support
//--------------------------------------------------------------
#include "AthenaPoolCnvSvc/ReadAthenaPool_jobOptions.txt"
ApplicationMgr.DLLs   += {  "HbookCnv"};


// Define the input Db parameters (the default value are shown)
// PoolSvc.Output    = "SimplePoolFile.root";
// PoolSvc.DbServer  =  "db1.usatlas.bnl.gov";
// PoolSvc.DbAccount =  "athena";
// PoolSvc.DbPassword = "";
// PoolSvc.DbType    = "mysql";
// PoolSvc.ConnectionType = "MySQLCollection";
// PoolSvc.FullConnection =
"mysql://athena:insider@db1.usatlas.bnl.gov/pool_collection";
// PoolSvc.Collection = "NewPoolTry";
PoolSvc.DbType                       = "root";  // to define ROOT file
resident collection
EventSelector.InputCollection        = "NewPoolTry";


// Converters:
#include "$EVENTATHENAPOOLROOT/pool/EventAthenaPool_joboptions.txt"
#include "$RECATHENAPOOLROOT/pool/RecAthenaPool_joboptions.txt"

// all object on input files are read-in by default

//---------------------------------------------------------
```

6.2.5 caveats, problems and work-arounds

o Storing pointers to objects require polymorphic classes: For classes which contain pointers to other objects, the classes of these objects must be polymorphic, i.e. they must have a virtual table. The simplest way to enforce this is to add a virtual destructor. For example:

```
class SimpleTrackCollection : public DataVector  {
public:
  virtual ~SimpleTrackCollection() {};
};
```

and

```
class SimpleTrack {
```

```
public:
  // ...
  // destructor
  virtual ~SimpleTrack(){};
  // ...
};
```

Here SimpleTrackCollection itself is the "data object" to be saved and thus must be polymorphic. And a DataVector is actually a collection of pointers to T, so SimpleTrack must be polymorphic as well.

Another important implication of the polymorphic requirement is that one may NOT have a class with a pointer to an STL collection. For example, the following is not allowed:

```
class SimpleTrack {
public:
  // ...
private:
  // Pointer to vector of hits:
  std::vector<HitOnTrack*>*  m_hits;
};
```

Rather one must use collections "by value":

```
class SimpleTrack {
public:
  // ...
private:
  // vector of hits "by value":
  std::vector<HitOnTrack*>  m_hits;
};
```

The reason for this is simply that the STL collections do not have virtual tables and thus are not polymorphic.

¥ Default constructor must not be private: POOL creates objects using the default constructor and then "fills" them by doing a memory copy of the data being read in (i.e. streaming). One consequence of this is that you will get a runtime error for classes where the default constructor has been made private.

¥ Too many classes with "id" defined: only the "data object" should have an id provided in the selection file. Otherwise pool complains when trying to write out the object. For example:

```
   <class name="SimpleTrackCollection" id="9E3595D6-1362-429A-8BA2-3396C93D6BA0"
/>
    <class name="SimpleTrack" id="23EF4872-EBFB-45E7-A256-34FDF223C10E" />
```

Only SimpleTrackCollection should have an id.

¥Renaming POOL output files: What you should NOT do: Rename the first output file, SimplePoolFile.root, to something else, then try to recreate SimplePoolFile.root in a second job. This will create two physcial files with the same file ID, which will cause trouble when reading.

If you do not want to modify the jobOpt when you run the second job, and you want to rename the output file after the first job, you should do the following:

```
> mv SimplePoolFile.root AnotherPoolFile.root
> FCrenamePFN -p SimplePoolFile.root -n AnotherPoolFile.root
```

Then run the second job. After the second job, you can do the following:

```
> mv SimplePoolFile.root YetAnotherPoolFile.root
> FCrenamePFN -p SimplePoolFile.root -n YetAnotherPoolFile.root
```

You can read the files (using implicite collections) with the following line:

```
EventSelector.InputCollections = {
                                   "AnotherPoolFile.root",
                                   "YetAnotherPoolFile.root"
                                 };
```

in your jobOptions.

¥ Sharing POOL output files with other people: What you should NOT do: Copy only the data files, and forgot to copy the PoolFileCatalog.xml.

If you want to give your output to other people to read, all the POOL output files (*.root), and the PoolFileCatalog.xml should be copied over to the other person’s run directory. The same read jobOption can be run from the other person’s run directory.

If you use the absolute path in the write job, for example:

```
 PoolSvc.Output    =
"/afs/cern.ch/atlas/maxidisk/d73/7.5.0/SimplePoolFile.root";
```

and the files are accessible by the other user, then no copying of the root data files are needed. Just copy the PoolFileCatalog.xml. Note that in this case, the files specified for InputCollections should have the absolute path too

Chapter 7
Detector Description

7.1 Overview

The ATLAS detector description is based upon the GeoModel geometry modeller. A more detailed version of this chapter is available online at:

http://atlas.web.cern.ch/Atlas/GROUPS/DATABASE/detector_description/Geometry%20Kernel%20Classes.doc

In particular, that document contains the full reference manual describing the classes in the Geometry Kernel.

7.2 About the Geometry Kernel Classes.

The geometry kernel classes are provided by the package `GeoModelKernel`. These classes provide a set of geometrical primitives for describing detectors, and a scheme for accessing both the raw geometry of a detector and arbitrary subsystem-specific geometrical services. The scheme provides a means of keeping the geometrical services synched to the raw geometry, while incorporating time-dependent alignments. It also allows one to version the geometry of any subsystem.

The design of these classes reflects the belief that raw geometry is highly constrained by the simulation engines, while the readout geometry is highly subsystem specific and has practically no constraint at all. The description of both types of geometry is normally to be carried out by a subsystem specialist.

This specialist is asked to extend objects called `GeoVDetectorElement`, `GeoVDetectorManager`, *and* `GeoVDetectorFactory`, *by writing subclasses describing both the raw and readout geometry of his or her subsystem.*

Thus, the simulation engines available today (Geant3 and Geant4), which fortunately have a high degree of conceptual commonality, basically determine the format of the raw geometry. Every subsystem engineer who is responsible for describing a subdetector needs to provide one or more trees of raw geometry for the purpose of simulation. These tasks creating and accessing raw geometry-- are required methods of the three basic base classes. The geometry kernel classes provide a set of geometrical primitives to support these operations..

In addition, the subsystem engineer has to layer, upon this raw geometry, any detector specific readout services required in simulation, reconstruction, or analysis. This is a very broad task and relies heavily on the creativity and intelligence of the subsystems specialist. The specialist is asked only to provide access to this type of information through the same class (`GeoVDetectorManager`) that accesses the raw geometry.

The set of `GeoVDetectorFactories` are then all called upon during the initialization phase to build both raw and readout geometries. During normal execution, messages to move various pieces of material to new, aligned positions will be routed from the calibration database to the detector managers which must respond by applying new alignment transformations at specific points in the geometry tree designated as alignable. The position of one or more pieces of raw geometry moves about when the alignable transformations are tweaked.

Readout geometry synchronizes itself to raw geometry by holding a pointer to a volume in the raw geometry tree that holds its absolute transformation with respect to world coordinates in cache. The readout geometry should access this information when responding to any queries about, or relying upon, its absolute position.

Thus the detector managers have a dual function: they describe the geometry (potentially misaligned) to the simulation, and they serve as a central store of detector-specific geometrical information which is accessed throughout Athena-based applications in ATLAS.

The interface to this information is largely up to the subsystem engineer. The `GeoVDetectorFactories` for each subsystem are called upon during the initialization of a service (`GeoModelSvc`) to construct geometry through a method called `create()`. They must provide access to the volumes so created, for the purpose of simulation. The `GeoModelSvc` then makes the `GeoVDetectorManagers` available to a variety of different clients.

7.3 Examples

In this section we give a few simple examples of how to use the geometry kernel. First, we illustrate how to get the information into the transient representation this is the job of the subsystem engineer , for whom this section will be very important. Second, we illustrate how to get the information out of the transient representation this will be important mostly for the individual who passes the description along to a procedure such as Geant3 or Geant4.. or one of may reconstruction tasks.

7.3.1 Example 1: Getting the data into the transient represention.

In this section we provide and illustrate a simple `GeoVDetectorFactory` subclass called a `ToyDetectorFactory`. This code describes a geometry that has 100 rings contained within a square box. The `ToyDetectorManager` contains two different types of readout elements: `CentralScrutinizers`, and `ForwardScrutinizers`. The header file for `ToyDetectorFactory` is shown in Listing 7.1:

Listing 7.1 Header file for `ToyDetectorFactory`

```
#include "GeoModelKernel/GeoVDetectorFactory.h"
#include "GeoModelExamples/ToyDetectorManager.h"
class ToyDetectorFactory : public GeoVDetectorFactory  {

   public:

     // Constructor:
     ToyDetectorFactory();

    // Destructor:
    ~ToyDetectorFactory();

    // Creation of geometry:
    virtual void create(GeoPhysVol *world);

    // Access to the results:
    virtual const ToyDetectorManager * getDetectorManager() const;

   private:

    // Illegal operations:
    const ToyDetectorFactory & operator=(const ToyDetectorFactory &right);
    ToyDetectorFactory(const ToyDetectorFactory &right);

    // The manager:
    ToyDetectorManager       *detectorManager;

};
```

From the header file, one can see that the subsystem engineer has created a class called ToyDetectorFactory, that derives from the class GeoVDetectorFactory, which is the base class for all subsystem-specific detector geometry factories. The ToyDetectorFactory is required to provide the following methods (because the base class declares them to be abstract functions):

```
virtual void create(GeoPhysVol *world);
```

Which builds the geometry within a containing physical volume (world volume).

The detector manager is returned from the factory and holds the entire geometry description for the subdetector. The header file for ToyDetectorManager is shown in Listing 7.2:

Listing 7.2 Header file for `ToyDetectorManager`

```
#include "CLIDSvc/CLASS_DEF.h"
class ToyDetectorManager;
CLASS_DEF(ToyDetectorManager, 9876, 1)
#include "GeoModelKernel/GeoVPhysVol.h"
#include "GeoModelKernel/GeoVDetectorManager.h"
#include "GeoModelExamples/CentralScrutinizer.h"
#include "GeoModelExamples/ForwardScrutinizer.h"

class ToyDetectorManager : public GeoVDetectorManager

  public:
   enum Type { CENTRAL, FORWARD} ;

   // Constructor
   ToyDetectorManager();
   // Destructor
   ~ToyDetectorManager();

   // Access to raw geometry:
   virtual unsigned int getNumTreeTops() const;
   // Access to raw geometry:
   virtual PVConstLink getTreeTop(unsigned int i) const;
   // Access to readout geometry:
   const ForwardScrutinizer * getForwardScrutinizer(unsigned int i) const;
   // Access to readout geometry:
   const CentralScrutinizer * getCentralScrutinizer(unsigned int i) const;
   // Access to readout geometry:
   unsigned int getNumScrutinizers(Type type) const;
   // Add a Tree top:
   void addTreeTop(PVLink);
   // Add a Central Scrutinizer:
   void addCentralScrutinizer(const CentralScrutinizer *);
   // Add a Forward Scrutinizer:
   void addForwardScrutinizer(const ForwardScrutinizer *);

 private:
   [ ...]
} ;
```

One sees from the interface that the manager is essentially a class that permits one to add and retrieve bits of detector description. Two methods, `getNumTreeTops()` and `getTreeTop(unsigned int i)`, are required and are used to access the number of top-level physical volumes in the system and allow one to access sequentially each top-level physical volume. Physical volumes, essentially, are positioned pieces of material with specific shape and composition. They are explained below in more detail. The raw geometry is organized in a treelikle structure, and the detector managers must provide

the top-level branch in the tree. The third method in the toy detector node creates the tree of volumes. We shall see in detail how, shortly.

The last three methods are not required but are provided by the subsystem engineer to describe pieces of readout or other detector-related geometrical information.

```
unsigned int getNumScrutinizers(Type type) const
const ForwardScrutinizer *getForwardScrutinizer(unsigned int i) const
const CentralScrutinizer *getCentralScrutinizer(unsigned int i) const
```

The last three methods give access to readout geometry. The basic pieces of readout geometry in the ToyDetectorManager are called ForwardScrutinizer and CentralScrutinizer. They derive from a base class called GeoVDetectorElement, which stores and provides access to a pointer to a GeoFullVPhysVol (this is a physical volume with an absolute global-to-local coordinate transformation in cache).

What kind of geometrical object are the scrutinizers? They are meant to illustrate pieces of detector with both material and readout properties. For example, in the inner detector, instead of a Scrutinizer one would create perhaps a pixel detector, giving the pixel detector the properties of readout pitch along local x and y, number of channels in x and y, and perhaps a multiplexing scheme. The vectors normal to the each side of the pixel detector could be provided through the pixel detectors s interface if that is a useful geometrical service for the pixel detector to provide and could be computed from the

full physical volumes absolute global-to-local coordinate transformation information. In the case of a calorimeter, the Scrutinzers would be replaced with a class describing a calorimeter module that could describe the peculiar way in which signals were summed within the calorimeter slices. And so forth.

Looking again at the interface to `ToyDetectorManager` and `Factory`: we wish to disable copying and assignment so we make these methods private and leave them unimplemented. We also declare some private member data required to carry out the services described above: a vector to hold the top level physical volumes, and two more to hold the lists of forward and central scrutinizers. Next we shall see how to implement this detector factory.

The implementation of the `ToyDetectorFactory` is shown in Listing 7.3. Note how the factory creates both raw geometry and readout geometry and puts it in the manager. In principal, one can tailor the code so that the detector factory itself determines the shape of the whole detector geometry, so that alternate geometries can be constructed simply by creating different types of factories and using them at run time.

The `ToyDetectorFactory` shown in Listing 7.3 is a simplified version of actual code that can be found in the Atlas repository. This simplified version does not contain illustration of certain advanced features namely , access to the material manager, interface to Athena, insertion of the managers within

Storegate, and parametrization of volumes using `GeoSerialTransformer` —that are present in the full version.

Listing 7.3 Implementation of ToyDetectorFactory

```
#include "GeoModelExamples/ToyDetectorFactory.h"
#include "GeoModelExamples/CentralScrutinizer.h"
#include "GeoModelKernel/GeoMaterial.h"
#include "GeoModelKernel/GeoBox.h"
#include "GeoModelKernel/GeoTube.h"
#include "GeoModelKernel/GeoLogVol.h"
#include "GeoModelKernel/GeoNameTag.h"
#include "GeoModelKernel/GeoPhysVol.h"
#include "GeoModelKernel/GeoFullPhysVol.h"
#include "GeoModelKernel/GeoTransform.h"
#include "GeoModelKernel/GeoSerialDenominator.h"
#include "GeoModelKernel/GeoAlignableTransform.h"

ToyDetectorFactory::ToyDetectorFactory()
   :detectorManager(NULL){}

ToyDetectorFactory::~ToyDetectorFactory()
{}
const ToyDetectorManager * ToyDetectorFactory::getDetectorManager() const {
  return detectorManager;
}

//## Other Operations (implementation)
void ToyDetectorFactory::create(GeoPhysVol *world)
{
   detectorManager=new ToyDetectorManager();

//------------------------------------------------------------------------//
// Get the materials that we shall use (material manager from Storegate!) //
//------------------------------------------------------------------------//
   const GeoMaterial *air  = materialManager->getMaterial("std::Air");
   const GeoMaterial *poly =
                     materialManager->getMaterial("std::Polystyrene");
 //Next make the box that describes the shape of the toy volume:
  const GeoBox     *toyBox   = new GeoBox(800*cm,800*cm, 1000*cm);
```

7.3.2 Example 2: Getting the data out of the transient representation.

This example is missing.

7.4 An Overview of the Geometry Kernel

In this section we give a short overview of all of the pieces of the geometry kernel. These pieces are described in detail in the online manual. In this section our goal is to describe the big picture . The GeoModel class tree is shown in Figure 9.1.

Figure 7.1 The GeoModel Class Tree

Many of the classes in the library represent objects which are reference counted; these all inherit from `RCBase`. Others represent geometrical shapes; these inherit from `GeoShape`. Others represent objects that can be assembled into a geometry graph; these inherit from `GeoGraphNode`.

7.4.1 The Detector Store Service and Detector Managers

The detector store service is not part of GeoModel per se, but rather an interface from GeoModel to Athena and Storegate. It is a Storegate service running within Athena and providing access to all detector information. The service can be accessed in the following way, which is typical of all Storegate services:

```
StoreGateSvc *detStoreSvc;
```

```
StatusCode status = service("DetectorStore",detStoreSvc);
```

The service hold several important objects. The first is the world physical volume, the common ancestor of all physical volumes within the system. This object has type GeoModelExperiment, which is a Storegate-compatible physical volume. It can be accessed like this:

```
const DataHandle<GeoPhysVol>  world;
StatusCode status = detStoreSvc->retrieve(world,"ATLAS");
```

From there, one may navigate the physical volume tree. The other objects that one can access through the detector store are the detector nodes, which are the master copy of all readout information. For example, for the liquid argon calorimeter, this might look like this:

```
const DataHandle<AbsLARDetectorNode> *laRNode;
StatusCode status = detStoreSvc->retrieve(laRNode,"LAr");
```

The strings used to retrieve detector nodes are assigned subsystems engineers. No catalogue can be published at this time. The detector factories are created by an interface called a tool, which instantiates the detector, and causes it to build its geometry within the world physical volume, and then also records the readout geometry within the detector store. The class `ToyDetectorTool` provides an example. It is in the source tree, under `DetectorDescription/GeoModel/GeoModelExamples`.

7.4.2 Material Geometry

Material geometry consists of a set of classes that bears a large resemblance to the material geometry within some flavour of GEANT. These classes, however, are designed to take a minimal size in memory. This requirement determines the basic data structure used to hold the data for the geometry description. That structure is a graph of nodes consisting of both volumes and their properties. The tree is built directly and accessed in a way that provides users access to volumes and, simulataneously, to the properties accumulated during graph traversal that apply to the volumes. See the Actions section, below.

The requirement of minimizing the memory consumption has led us to foresee a system in which objects (as well as classes) in the detector description can be re-used. This is called shared instancing, and is described below. It essentially means that an element, compound, volume, or entire tree of volumes may be referenced by more than one object in the detector description. Shared instancing can make the deletion of objects difficult unless special measures are taken. We have used a technique called reference counting in order to facilitate clean-up and make it less error prone. Using that technique, objects can be created using operator new. The memory is then freed when some action is taken to clean up near the top of the tree. See the section How Objects are Created and Destroyed.

Before creating hierarchies of volumes representing positioned pieces of detectors, we need to create lower level primitives, such as elements, materials, and shapes. So, we will discuss these first.

7.4.3 Materials

Materials are represented within the geometry kernel class library by the class `GeoMaterial`, and are built up by combining different elements, specifying each element and its fraction-by-mass. Material constants such as the radiation length and the interaction length, as well as constants for ionization energy loss, are available through the interface but do not need to be provided to the constructor. Instead, they are computed from the material s element list.

The class `GeoElement` is used to represent elements. Their constructor requires a name, symbol, and effective Z and A. These properties can also be retrieved from the element.

`GeoMaterial` objects are created by specifying a name and a density. The material is empty until elements are added, one by one, using the `add()` method, which is overloaded so that one may provide either elements or prebuilt materials. After all materials are added, the `lock()` method must be called, which prevents further elements or materials from being added.

Material classes, as well as all other classes, use the CLHEP Units wherever applicable. One should normally give units when specifying densities, lengths, volumes, or other quantities in the methods of all of the classes in this library. Therefore, when specifying water, one should use a constructor call like this:

```
GeoMaterial *water = new GeoMaterial("H20", 1.0*gram/cm3);
```

The CLHEP Units are described on the CLHEP web page . To finish constructing this material, water, one needs to follow the constructor with the following lines:

```
GeoElement *hydrogen = new GeoElement("Hydrogen","H",1.0, 1.010);
GeoElement *oxygen   = new GeoElement("Oxygen",  "O", 8.0, 16.0);
water->add(hydrogen,0.11);
water->add(oxygen,0.89);
water->lock();
```

The materials are then used to together with shapes to form logical volumes, discussed below.

7.4.3.1 Shapes

Shapes are created using the new operator. Essentially, shapes within this system are required to store and provide access to the geometrical constants that describe their geometrical form. This data is, insofar as possible, to be specified on the constructor.

Shapes are extensible and we intend to service requests for extensions, by providing custom shapes to valued customers on request .

Listing 7.4 illustrates how one builds a box.

Listing 7.4 How to build a box

```
double        length=100*cm, width=200*cm, depth=33*cm;
GeoBox *box = new GeoBox(length, width, depth);
```

Most objects can be constructed along similar lines; exceptions are objects with multiple planes such as polycones and polygons; their interface allows one to add planes successively. For the polycone, for example, the shape is built as shown in Listing 7.5.

Listing 7.5 How to build a polycone

```
double dphi=10*degrees, sphi=40*degrees;
GeoPcon  *polycone=new GeoPcon(dphi,sphi);

double z0=0.0, rmin0=5, rmax0=10.0;
polycone->addPlane(z0,rmin0,rmax0);

double z1=10.0, rmin1=6, rmax1=12.0;
polycone->addPlane(z1,rmin1,rmax1);

double z2=15.0, rmin2=5, rmax2=10.0;
polycone->addPlane(z1,rmin1,rmax1);
```

This creates a polycone whose projection subtends an angle of 10 degrees between 40 degrees and 50 degrees, with planes at z=0, z=10, and z=15, with minimum and maximum radii there of (5,10), (6, 12), and (5,10).

The shapes can provide their data to a client through their accessors, and in addition support several other operations. Boolean operations on shapes are possible. They can be accomplished through Boolean operators in class GeoShape:

```
GeoShape       * donut  = new GeoTube();
GeoShape       * hole   = new GeoTube();
const GeoShape & result = (donut->subtract(*hole));
```

The result of a Boolean operation is a shape in a boolean expression tree that can, for example, be decoded when the geometry is declared to GEANT.

Another method that shapes can carry out is to compute their volume. This is useful in the context of mass inventory, in which the mass of the detector model is computed, usually for the purpose of comparing with an actual installed detector. One needs to call the `.volume()` method which is defined for all shape types.

Finally, we mention a type identification scheme for shapes. The scheme relies on two static and two virtual methods which together can be used as follows:

```
// Test if the shape is a box:
if (myShape->typeId()==GeoBox::classTypeId()) {
   .....
}
```

The methods `typeId()` and `classTypeId()` return unsigned integers, making the type identification very fast. Alternately one can use the methods `type()` and `classType()`, which work in the same way, except that these methods return std::strings: Box , T ubs, Cons, etc.

7.4.3.2 Logical Volumes

Logical volumes represent, conceptually, a specific manufactured piece that can be placed in one or more locations around the detector. A logical volume is created by specifying a name tag for the volume, a shape, and a material:

```
const GeoLogVol   *myLog  = new  GeoLogVol("MyLogVol",
                                           myShape,
                                           gNitrogen);
```

7.4.3.3 Physical Volumes and the Geometry Graph

Having created elements, materials, shapes, and logical volumes, you are now ready to create and locate placed volumes called physical volumes. Before you start, you will need to know that there are two kinds of these:

¥ Regular Physical Volumes, designed to be small.

¥ Full Physical Volumes, designed to hold in cache complete information about how the volume is located with respect to the world volume, its formatted name string and other important information.

There is a common abstract base class for all of these: `GeoVPhysVol`. In addition both the full physical volumes have another layer of abstraction, `GeoVFullPhysVol`, in order to allow us to introduce parametrized volumes in the near future. All physical volumes allow access to their children.

The concrete subclasses that you have at your disposition for detector description are called `GeoPhysVol` and `GeoFullPhysVol`. Both of these have a method to add either volumes or volume properties.

```
GeoPhysVol *myVol;
myVol->add(aTransformation);
myVol->add(anotherVolume);
```

When you add a transformation, you change the position of the subsequent volume with respect to the parent. If you add no transformation, you will not shift the daughter relative to the parent and commonly will create a daughter which is centered directly in the parent. If you add more than one transformation to the volume before adding a parent, they will be multiplied. The last transformation to be added is applied first to the child. Transformations are discussed next. Like logical volumes, they may be shared.

Like physical volumes, transformations come in two types:

¥ Regular transformations, designed to be small.

¥ Alignable transformations, which allow one to add a misalignment to the system. Misaligning a transformation changes the position of all volumes under the transformation and clears the absolute location caches of all full physical volumes.

When you create a transformation you must choose the type.

The model of the raw geometry is a tree of nodes, property nodes and volume nodes. The tree can be thought of as as tree of volumes, each one having a set of properties (inherited from property nodes throughout the tree). The subsystem engineer judiciously chooses which of the volumes are to contain full, cached, position information — usuallythese first-class volumes are to be associated with a detector. He or she also judiciously decides which of the transformations are to be alignable usually these are the transformations which position something that ultimately has a detector bolted, glued, riveted or otherwise clamped onto a sensitive piece. Then, a GeoVDetectorFactory which builds the geometry keeps track of these pointers so that it may connect the important volumes to detector elements and that it may connect the alignable transformations to the alignment database for periodic updating.

Finally, we provide three mechanisms for giving names to volumes:

- Do nothing. The volume will be called ANON .
- Add a `GeoNameTag` object to the graph before adding a volume. The next volume to be added will be given the `GeoNameTag` s name.
- Add a `GeoSerialDenominator` object to the graph before adding more volumes. The volumes will be named according to the base name of the `GeoSerialDenominator`, plus given a serial number 0, 1, 2, 3 ..

In effect this last method can be thought of as a way of parametrizing the name of the volume.

7.4.3.4 Actions

There are two ways of getting raw geometry information out of the model. Suppose that one has access to a particular physical volume (it could be the World physical volume).

One can access its children, there names, and their transformations with respect to the parent in the following way:

```
PVConstLink myVol;
for (int c=0; c< myVol->getNChildVols();c++) {
    PVConstLink child = myVol->getChildVol(c);
    HepTransform3D  xf = myVol->getXToChildVol(c);
}
```

One could then iterate in a similar way over the grand children, by using a double loop. Ultimately one would probably to visit all the volumes, whatever their depth in the tree, so probably this would call on some form of recursion. An easy way would be to embed the small sample of code shown above in a recursive subroutine or method. That would be fine, and is conceptually simple. However, within the geometry model s kernel, we have provided an alternate, probably better way to visit the entire tree.

That mechanism involves a `GeoVolumeAction`. A `GeoVolumeAction` is a way (for applications programmers) to obtain recursive behavior without writing any recursive routines. It s a class with a handler routine (`handleVPhysVol`) which is called for each node before (or after) it is called on its children. This can descend to an arbitrary depth in the tree. The `GeoVolumeAction` is an abstract base class and should be subclassed by programmers to suit their needs. Another class

`TemplateVolAction` is provided as a template that one can take and modify. To run it, one does this:

```
PVConstLink myVol;
TemplateVolAction tva;
myVol->apply(&tva);
```

The `handleVPhysVol` within the `TemplateVolAction` is where the work is supposed to get done. It will be invoked repeatedly, once for each node in the tree. Within that routine, one can access the physical volume as a subroutine parameter, and information about the transformation and the path to the node through the base class for actions, `GeoVolumeAction`. The action can be designed to run from the bottom up or from the top down.

Incidentally, there is another kind of action in the library called `GeoNodeAction`. `GeoNodeActions` visit all nodes (including naming nodes, transformation nodes, and perhaps other property nodes that may be added later to the model) Since usually an application programmer wants to see volumes and their properties, the `GeoVolumeAction` is more suited to casual users than the `GeoNodeAction`, which is considered mostly internal. However the usage is similar, except that node actions are exec d while volume actions are applied . Here for example is how we can rewrite the loop over children using volume actions:

```
PVConstLink myVol;
for (int c=0; c< myVol->getNChildVols();c++) {
    GeoAccessVolumeAction av(c);
    myVol->exec(&ac);
    PVConstLink  child = ac.getVolume();
    HepTransform3D  xf = ac.getTransform();
}
```

This, it turns out, will execute faster than the loop shown above, which (internally) will run the action, twice: once, in order to locate the daughter volume and then a second time, to locate its transform.

7.4.3.5 How Objects are Created and Destroyed

We now come to the important topic of how objects in this system are created and destroyed. The geometry kernel uses a technique called reference counting. Reference counting, shortly stated, is a way to perform an automatic garbage collection of nodes that are no longer in use. This is important when describing a large tree of information, much of which is ideally to be shared used again and again in many places.

You may have noticed, in the section Example 1: Getting the data into the transient represention. that many of the objects have been created using operator `new`. You may have also noticed, if you ve tried to play around with the kernel classes, that statements which allocate most kernel classes on the stack, such as:

```
GeoBox box(100, 100, 100);
```

are not allowed. Who is going to clean up the memory after all these new operations? And why does the compiler disallow allocation on the stack?

Let s look again at Example 1, especially at these lines shown in Listing 7.6.

Listing 7.6 Object creation

```
const GeoBox      *worldBox    = new GeoBox(1000,1000, 1000);
const GeoLogVol   *worldLog    = new GeoLogVol("WorldLog",
                                 worldBox, gNitrogen);
GeoPhysVol     *worldPhys  = new GeoPhysVol(worldLog);
```

Each of the three objects (worldBox, worldLog, and worldPhys) are created with a reference count. WorldBox s is initially zero, at the time it is created. WorldLog s is also zero when it is created. However, when worldVol is created, the reference count of worldBox increases to one, since now it is referenced somewhere namely by the logcal volume worldLog. We can diagram this sequence in the following way:

Now, when the physical volume worldPhys is created, the reference count of the logical volume will increase to one since it is used once by a single physical volume.

Each time a physical volume is positioned within another physical volume, its reference count increases. Suppose we look now at a sub-tree of physical volumes that is used five times. At a run boundary, it may happen that a piece of the tree is torn down. When the first node referencing the physical volume is destroyed, it decreases the volumes reference count, from five to four. When the next node referencing the physical volume is destroyed, the reference count goes from four to three. And so forth.

When the very last node referencing the physical volume is destroyed, this means that the physical volume itself has outlived its usefulness and should disappear. And that is what happens. The destruction of objects is carried out automatically when the reference count falls to zero. And in fact, the only way to delete an object is to arrange for all of its references to disappear. This is because the destructor of all reference counted objects is private.

This scheme applies to elements, materials, shapes, logical volumes, physical volumes, full physical volumes,

So far, we have described what happens to an object when it is no longer used by any other node in the tree. However, what about the top of the tree, which has no nodes that refer to it? Since the destructors of our physical volumes are private, how do you arrange to get it to go away?

Reference counts can also be manipulated manually, by using the methods `ref()` and `unref()`. The physical volume at the head of the tree, often known as the world physical volume, can be referenced manually using this call:

```
worldPhys->ref();  //reference count goes from 0 to 1.
```

Later, you can destroy the world volume and trigger a global collection of garbage by using this call:

```
worldPhys->unref();//reference count goes from 1 to 0.
```

When this happens the world physical volume deletes itself, decreasing the reference counts of it logical volumes and any children. These will then begin dereferencing and possibly deleting their own children, until all the memory has been freed.

Suppose now, that you want to arrange for a node to not be deleted automatically in this fashion even when nobody references it any more. In order to do this, simply call the ref() method on this object. That way, the reference counts starts at 1 and will not fall to zero until you call unref(), manually.

7.4.4 Detector Specific Geometrical Services

Detector specific geometrical services are known to some as readout geometry . This consists, first and foremost, of geometrical information that is not declared directly to the tracing engines, G3, for example, or G4. Examples would include: projective towers within a calorimeter, or implant regions within a piece of silicon. Information such as the position of the boundaries of these regions is not required in the simulation of basic physics processes, though it certainly is required in the digitization, and possibly hit-making phase of simulation.

Detector-specific geometrical services can and should include services that derive from the basic raw and readout geometry of the detector. Such services could include point-of-closest-approach calculations, global-to-local coordinate transformations, calculations that compute the total number of radiation lengths within a cell, et cetera. Additionally they could include nearest-neighbor calculations, hopefully in a highly detector specific way which is meaningful in the context of specific algorithms.

We have intended that this kind of service would be provided by the subsystem engineer, or somebody with an intimate knowledge of both the detector geometry and the requirements of hit simulation and/or reconstruction in the detector. This kind of service, ideally, would be spread across at least two classes.

The first place is in the detector element. The detector element (subclass of `GeoVDetectorElement`) has a required association with a piece of material geometry, and has access to that piece. The rest of the interface all of the geometrical services discussed above, such as the boundaries of implant layers, strip pitches, whatever, can be placed in the detector element.

The second place where detector specific geometrical services may be placed is in the interface to the the detector manager (subclass of `GeoVDetectorManager`), which constructs and manages all raw and readout geometry. This class should provide a fast mechanism for accessing the detector elements that it manages such as detector -specific, array-based random access. Other services, such as returning the maximum and minimum range of some array index (phi, eta, etc.) may also be appropriate.

So in general the subsystems people have a lot of flexibility, but need to devise an interface to both the detector manager and the detector element that satisfies their needs. The exact layout of these classes is hopefully the object of some design on the part of the engineer, can evolve with experience to involve a larger category of collaborating classes[1]. The basic framework requires only that 1) detector factories

1. In certain CDF subdetectors, for example, all questions involving numerical limits to array boundaries were ultimately handled separately by "numerology" classes, available through the detector node.

create a physical volume tree, 2) they associate readout elements to certain physical volumes, and 3) additional readout information appear in the interface to the detector manager and the detector element.

7.4.5 Alignment

There are two alignment issues we need to address: first, how does the GeoModel propagate alignment constants into the geometry description? Second, how is the subsystem engineer supposed to connect the alignment constants to the database so that the geometry changes when the run conditions are updated? The first issue concerns the way that GeoModel works, the second issue is mostly a policy question and outside the scope of GeoModel, per se.

GeoModel has a natural way of putting alignment constants into the geometry description and a natural way of getting them out. To put them it, one alters one or more `GeoAlignableTransform` objects by changing its Delta , or misalignment, which is a `HepTransform3D`. The misalignment is then composed with the default transformation.

To get the alignment out of GeoModel, simply query a physical volume for its transformation. All physical volumes have the notion of relative and absolute transformations, both default and (mis)aligned. Full physical volumes cache the absolute transformation, making it immediately available after the first request, while ordinary physical volumes compute it anew each time during tree traversal. In either case, GeoModel methods supply an answer that correctly incorporates the effects of misalignment.

In case of cached transformations, it s worthwhile to describe the mechanism by which the cache is updated. First, when an alignable transform is altered, all parent physical volumes receive a message to clear any caches. These messages are passed onto their daughter volumes, and any physical volume in the geometry tree that contains a cache of absolute transformations is cleared. Then, as soon as some client requests a transformation, it is recomputed recursively, starting from the first parent with valid cache information, and again cached.

A piece of readout geometry (class `GeoVDetectorElement`) cannot be constructed without a full physical volume. One constructed, it always has access to that volume s transformation. Readout geometry should respond to all queries relevant to absolute spatial positioning by referencing the absolute transformation of the physical volume. See section 2.9 for more details.

Finally, how should the subsystem engineer arrange for the geometry to be updated when run conditions change? The basic suggestion is to use the notification mechanisms of the calibration database. In this scheme, engineer should arrange for the detector manager to receive a message when some relevant database table has changed. The detector manager should then rescan the tables, construct new Delta s for each alignable transform under its jurisdiction, and alter those transforms.

When updates occur, readout elements may be required to update any local cache of information that derives, ultimately, from alignment constants. This can be arranged using the same notification mechanism.

For the moment no documentation on the calibration database can be cited. The need for this component is not considered urgent as of this writing.

7.4.6 On Memory Use

Some effort has been spent insuring that the memory used by the a geometry description can be made small, and indeed, it is our belief that using the techniques made possible by this class library a remarkably compact description of the geometry can be achieved. However a compact geometry will not occur automatically. Users need to know what tricks are available, and need to apply them as aggressively as possible.

If aggressive optimization of process size is done, across the board and from the beginning, we think that the GeoModel geometry description could contribute a negligible amount to the overall process size of a typical ATLAS executable.

This goal is worth working towards, for three reasons. First, if the process size is really negligible, then ATLAS executables can instantiate and use the whole geometry description, including even material geometry, at virtually no cost.

Second, it will mean that at GeoModel description could be kept alive even after the whole model has been declared to a simulation engine, such as GEANT3/4.

Third, experience shows that process size becomes unmanageable in large-scale projects unless the memory cost is carefully controlled from the beginning.

Here are some suggestions for how to minimize the size of the geometry description, in memory:

¥ Share instances of elements, materials, shapes, logical volumes and physical volumes, and even transformations.

¥ Use full physical volumes and alignable transforms only where necessary.

¥ Do not give names to physical volumes that represent uninteresting, nondescript pieces of material.

¥ In case you need to give names to physical volumes, use a serial denominator rather than multiple name tags.

¥ Parameterize volumes where possible.

The best way of sharing instances of elements and materials is to create them within a dedicated service and access that service, experiment-wide, for any materials that are required to construct the geometry. Logical volumes and shapes should be simple to share if adequate care is taken. Shared instancing and parameterization of physical volumes is limited mostly by the constraints that:

¥ Physical volumes representing active elements must be full and distinct, since they exist to cache an absolute position. This means that they must not be shared, or parameterized, nor live in any branch of a physical volume tree which is shared or parameterized.

Finally, transformations could be shared by creating a bank of common transformations such as common f rotations and reusing them instead of instantiating, say, a θ rotation hundreds of times. When shared instancing of transformations works, however, parameterization will also usually work and is generally a better solution. Note, parameterizing volumes in GeoModel does not mean that G4 parameterization must be used during simulation. We can and should make this optional.

Not all of the planned optimization tools are available in this release. Notably, parameterization of shapes (as opposed to transformations, only) has not been implemented, and a compressed representation for CLHEP transforms is not available. We foresee adding both of these features to the library at a later date. The first feature will give certain clients more powerful parameterization techniques, such as distortion fields which are needed ultimately by the liquid argon calorimeter software; while the second feature will allow a global reduction in memory cost in a way which is virtually transparent to the users.

Chapter 8
Histogram facilities

8.1 Overview

The histogram data store is one of the data stores discussed in Chapter 2. Its purpose is to store statistics based data and user created objects that have a lifetime of more than a single event (e.g. histograms).

As with the other data stores, all access to data is via a service interface. In this case it is via the `IHistogramSvc` interface, which is derived from the `IDataProviderSvc` interface discussed in Chapter 7. The user asks the Histogram Service to book a histogram and register it in the histogram data store. The service returns a pointer to the histogram, which can then be used to fill and manipulate the histogram, using the methods defined in the `IHistogram1D` and `IHistogram2D` interfaces and documented on the AIDA (Abstract Interfaces for Data Analysis) project web pages: *http://wwwinfo.cern.ch/asd/lhc++/AIDA/.*

Internally, Athena uses the transient part of HTL (Histogram Template Library, *http://wwwinfo.cern.ch/asd/lhc++/HTL/*) to implement histograms.

8.2 The Histogram service.

An instance of the histogram data service is created by the application manager. After the service has been initialised, the histogram data store will contain a root directory, always called `/stat`, in which users may book histograms and/or create sub-directories (for example, in the code fragment below, the histogram is stored in the subdirectory `/stat/simple`). A suggested naming convention for the sub-directories is given in Section 1. Note that the string "`/stat/`" can be omitted when referring to a histogram in the data store: "`/stat/simple`" is equivalent to "`simple`", **without** a leading "`/`".

As discussed in Section 3.2, the `Algorithm` base class defines a member function which returns a pointer to the `IHistogramSvc` interface of the standard histogram data service . Access to any other

```
IHistogramSvc* histoSvc()
```

non-standard histogram data service (if one exists) must be sought via the `ISvcLocator` interface of the application manager as discussed in section 10.2.

8.3 Using histograms and the histogram service

The code fragment below shows how to book a 1D histogram and place it in a directory within the histogram data store, followed by a simple statement which fills the histogram.

```
#include "AIDA/IHistogram1d.h"
...
// Book 1D histogram in the histogram data store
IHistogram1d* m_hTrackCount= histoSvc()->
           book( "simple", 1, “TrackCount“, 100, 0., 3000. );
SmartDataPtr<MyTrackVector> particles( eventSvc(),“/Event/MyTracks” )
if ( 0 != particles ) {
  // Filling the track count histogram
  m_hTrackCount->fill(particles->size(), 1.);
}
```

The parameters of the `book` function are the directory in which to store the histogram in the data store, the histogram identifier, the histogram title, the number of bins and the lower and upper limits of the X axis. 1D histograms with fixed and variable binning are available. In the case of 2D histograms, the `book` method requires in addition the number of bins and lower and upper limits of the Y axis.

If using HBOOK for persistency, the histogram identifier should be a valid HBOOK histogram identifier (number) and must be unique within the RZ directory the histogram is assigned to. The name of the RZ directory is given by the directory and parent directories in the transient histogram store. Please note that HBOOK accepts only directory names, which are shorter than 16 characters and that HBOOK internally converts any directory name into upper case. Even if using another persistency solution (e.g. ROOT) it is recommended to comply with the HBOOK constraints in order to make the code independent of the persistency choice.

The call to `histoSvc()->book(...)` returns a pointer to an object of type `IHistogram1D` (or `IHistogram2D` in the case of a 2D histogram). All the methods of this interface can be used to further manipulate the histogram, and in particular to fill it, as shown in the example. Note that this pointer is guaranteed to be non-null, the algorithm would have failed the initialisation step if the histogram data service could not be found. On the contrary the user variable `particles` may be null (in case of absence of tracks in the transient data store and in the persistent storage), and the fill statement would fail - so the value of `particles` must be checked before using it.

Algorithms that create histograms will in general keep pointers to those histograms, which they may use for filling operations. However it may be that you wish to share histograms between different algorithms. Maybe one algorithm is responsible for filling the histogram and another algorithm is responsible for fitting it at the end of the job. In this case it may be necessary to look for histograms within the store. The mechanism for doing this is identical to the method for locating event data objects within the event data store, namely via the use of smart pointers, as discussed in section 7.8.

```
SmartDataPtr<IHistogram1D> hist1D( histoSvc(), "simple/1" );
if( 0 != hist1D ) {
  // Print the found histogram
  histoSvc()->print( hist1D );
}
```

8.4 Persistent storage of histograms

By default, Athena does not produce a persistent histogram output. The options exist to write out histograms either in HBOOK or in ROOT format. The choice is made by giving the job option `ApplicationMgr.HistogramPersistency`, which can take the values "`NONE`" (no histograms saved, default), "`HBOOK`" or "`ROOT`". Depending on the choice, additional job options are needed, as described below.

8.4.1 HBOOK persistency

The HBOOK conversion service converts objects of types `IHistogram1D` and `IHistogram2D` into a form suitable for storage in a standard HBOOK file. In order to use it you first need to tell Athena where to find the `HbookCnv` shared library. If you are using CMT, this is done by adding the following line to the CMT `requirements` file:

```
use HbookCnv v*
```

You then have to tell the application manager to load this shared library and to create the HBOOK conversion service, by adding the following lines to your job options file:

```
ApplicationMgr.DLLs += {"HbookCnv"};
ApplicationMgr.HistogramPersistency = "HBOOK";
```

Finally, you have to tell the histogram persistency service the name of the output file:

```
HistogramPersistencySvc.OuputFile = "histo.hbook";
```

Note that it is also possible to print the histograms to the standard output destination (HISTDO) by setting the following job option (default is `false`).

```
HistogramPersistencySvc.PrintHistos = true;
```

8.4.2 ROOT persistency

The ROOT conversion service converts objects of types `IHistogram1D` and `IHistogram2D` into a form suitable for storage in a standard ROOT file. In order to use it you first need to tell Athena where to find the `RootHistCnv` shared library. If you are using CMT, this is done by adding the following line to the CMT `requirements` file:

```
use RootHistCnv v*
```

You then have to tell the application manager to load this shared library and to create the ROOT histograms conversion service, by adding the following lines to your job options file:

```
ApplicationMgr.DLLs += {"RootHistCnv"};
ApplicationMgr.HistogramPersistency = "ROOT";
```

Finally, you have to tell the histogram persistency service the name of the output file:

```
HistogramPersistencySvc.OuputFile = "histo.rt";
```

Chapter 9
N-tuple and Event Collection facilities

9.1 Overview

In this chapter we describe facilities available in Athena to create and retrieve N-tuples. We discuss how Event Collections, which can be considered an extension of N-tuples, can be used to make preselections of event data. Finally, we explore some possible tools for the interactive analysis of N-tuples.

9.2 N-tuples and the N-tuple Service

User data - so called N-tuples - are very similar to event data. Of course, the scope may be different: a row of an N-tuple may correspond to a track, an event or complete runs. Nevertheless, user data must be accessible by interactive tools such as PAW or ROOT.

Athena N-tuples allow to freely format structures. Later, during the running phase of the program, data are accumulated and written to disk.

The transient image of an N-tuple is stored in a Athena data store which is connected to the N-tuple service. Its purpose is to store user created objects that have a lifetime of more than a single event.

As with the other data stores, all access to data is via a service interface. In this case it is via the `INTupleSvc` interface which extends the `IDataProviderSvc` interface. In addition the interface to the N-tuple service provides methods for creating N-tuples, saving the current row of an N-tuple or retrieving N-tuples from a file. The N-tuples are derived from `DataObject` in order to be storable, and are stored in the same type of tree structure as the event data. This inheritance allows to load and locate N-tuples on the store with the same smart pointer mechanism as is available for event data items (c.f. Chapter 7).

9.2.1 Access to the N-tuple Service from an Algorithm.

The `Algorithm` base class defines a member function which returns a pointer to the `INTupleSvc` interface .

```
INTupleSvc* ntupleSvc()
```

The N-tuple service provides methods for the creation and manipulation of N-tuples and the location of N-tuples within the persistent store.

The top level directory of the N-tuple transient data store is always called /NTUPLES . The next directory layer is connected to the different output streams: e.g. /NTUPLES/FILE1 , where FILE1 is the logical name of the requested output file for a given stream. There can be several output streams connected to the service. In case of persistency using HBOOK, FILE1 corresponds to the top level RZ directory of the file (...the name given to HROPEN). From then on the tree structure is reflected with normal RZ directories (caveat: HBOOK only accepts directory names with less than 8 characters! It is recommended to keep directory names to less than 8 characters even when using another technology (e.g. ROOT) for persistency, to make the code independent of the persistency choice.). Note that the top level directory name "/NTUPLES/" can be omitted when referring to an N-tuple in the transient data store - in the example above the name could start with "FILE1" (without a leading "/").

9.2.2 Using the N-tuple Service.

This section explains the steps to be performed when defining an N-tuple:

¥ The N-tuple tags must be defined.

¥ The N-tuple must be booked and the tags must be declared to the N-tuple.

¥ The N-tuple entries have to be filled.

¥ The filled row of the N-tuple must be committed.

¥ Persistent aspects are steered by the job options.

9.2.2.1 Defining N-tuple tags

When creating an N-tuple it is necessary to first define the tags to be filled in the N-tuple, as shown for example in Listing 9.1:

Listing 9.1 Definition of N-tuple tags.

```
1: NTuple::Item<long>            m_ntrk; // A scalar item (number)
2: NTuple::Array<bool>           m_flag; // Vector items
3: NTuple::Array<long>           m_index;
4: NTuple::Array<float>          m_px, m_py, m_pz;
5: NTuple::Matrix<long>          m_hits; // Two dimensional tag
```

Typically the tags belong to the filling algorithm and hence should be provided in the Algorithm s header file. Currently the supported data types are: `bool`, `long`, `float` and `double`. `double` types (Fortran REAL*8) are not recommened if using HBOOK for persistency: HBOOK will complain if the N-tuple structure is not defined in a way that aligns `double` types to 8 byte boundaries. In addition PAW cannot understand `double` types.

9.2.2.2 Booking and Declaring Tags to the N-tuple

Listing 9.2 shows how to book a column-wise N-Tuple. The first directory specifier (`FILE1` in the example) must correspond to an open output stream (see Section 9.2.3.2); lower directory levels are created automatically. After booking, the previously defined tags must be declared to the N-tuple; if not, they are invalid and will cause an access violation at run-time.

Listing 9.2 Creation of a column-wise N-tuple in a specified directory and file.

```
 1: #include "GaudiKernel/NTuple.h"
 2: ..
 3: NTuplePtr nt1(ntupleSvc(), "FILE1/MC/1");
 4: if ( !nt1 )    {     // Check if already booked
 5:  nt1=ntupleSvc()->book("FILE1/MC/1",CLID_ColumnWiseTuple,"Hello World");
 6:  if ( 0 != nt1 )    {
 7:  // Add an index column
 8:    status = nt1->addItem ("Ntrack", m_ntrk, 0, 5000 );
 9:  // Add a variable size column, type float (length=length of index col)
10:    status = nt1->addIndexedItem ("px", m_ntrk, m_px);
11:    status = nt1->addIndexedItem ("py", m_ntrk, m_py);
12:    status = nt1->addIndexedItem ("pz", m_ntrk, m_pz);
13:  // Another one, but this time of type bool
14:    status = nt1->addIndexedItem ("flg",m_ntrk, m_flag);
15:  // Another one, type integer, numbers must be within [ 0, 5000]
16:    status = nt1->addIndexedItem ("idx",m_ntrk, m_index, 0, 5000 );
17:  // Add 2-dim column: [ 0:m_ntrk][ 0:2] ; numerical numbers within [ 0, 8]
18:    status = nt1->addIndexedItem ("hit",m_ntrk, 2, m_hits, 0, 8 );
19:  }
20:  else    {    // did not manage to book the N tuple....
21:    return StatusCode::FAILURE;
22:  }
23: }
```

In previous versions of Gaudi (up to v8), indexed items were added with the `addItem` function, causing confusion for users. For this reason the calls to add indexed arrays and matrices were changed, these should now be added using the member function `addIndexedItem`. Please consult the doxygen code documentation for further details. The old calls still exist, however they are deprecated.

Row wise N-tuples are booked in the same way, but giving the type `CLID_RowWiseTuple`. However, only individual items (class `NTuple::Item`) can be filled, no arrays and no matrices. Clearly this excludes the usage of indexed items. For row-wise N-tuples to be saved in HBOOK format, it is recommended to use only `float` type, for the reasons explained in Section 9.2.3.3.

When using HBOOK for persistency, the N-tuple identifier ("1" in this example) must be a number and must be unique in a given directory. This is a limitation imposed by HBOOK RZ directories. It is recommended to keep this number unique even when using another technology (e.g. ROOT) for persistency, to make the code independent of the persistency choice.

9.2.2.3 Filling the N-tuple

Tags are usable just like normal data items, where

¥ `Items<TYPE>` are the equivalent of numbers: `bool`, `long`, `float`.

¥ `Array<TYPE>` are equivalent to 1 dimensional arrays: `bool[size]`, `long[size]`, `float[size]`

¥ `Matrix<TYPE>` are equivalent to an array of arrays or matrix: `bool[dim1][dim2]`.

Implicit bounds checking is not possible without a rather big overhead at run-time. Hence it is up to the user to ensure the arrays do not overflow.

When all entries are filled, the row must be committed, i.e. the record of the 7N-tuple must be written.

Listing 9.3 Filling an N-tuple.

```
 1: m_ntrk = 0;
 2: for( MyTrackVector::iterator i = mytracks->begin(); i !=
    mytracks->end(); i++ )    {
 3:   const HepLorentzVector& mom4 = (*i)->fourMomentum();
 4:   m_px[ m_ntrk]  = mom4.px();
 5:   m_py[ m_ntrk]  = mom4.py();
 6:   m_pz[ m_ntrk]  = mom4.pz();
 7:   m_index[ m_ntrk]  = cnt;
 8:   m_flag[ m_ntrk]  = (m_ntrk%2 == 0) ? true : false;
 9:   m_hits[ m_ntrk][ 0]  = 0;
10:   m_hits[ m_ntrk][ 1]  = 1;
11:   m_ntrk++;
12: // Make sure the array(s) do not overflow.
13:   if ( m_ntrk > m_ntrk->range().distance() ) break;
14: }
15: // Commit N tuple row. See Listing 9.2 for initialisation of m_ntuple
16: status = m_ntuple->write();
17: if ( !status.isSuccess() ) {
18:   log << MSG::ERROR << "Cannot fill id 1" << endreq;
19: }
```

9.2.2.4 Reading N-tuples

Although N-tuples intended for interactive analysis, they can also be read by a regular program. An example of reading back such an N-tuple is given in Listing 9.4.

Listing 9.4 Reading an N-tuple.

```
 1:  NTuplePtr nt(ntupleSvc(), "FILE1/ROW_WISE/2");
 2:  if ( nt )    {
 3:    long count = 0;
 4:    NTuple::Item<float> px, py, pz;
 5:    status = nt->item("px", px);
 6:    status = nt->item("py", py);
 7:    status = nt->item("pz", pz);
 8:    // Access the N tuple row by row and print the first 10 tracks
 9:    while ( nt->read().isSuccess() )    {
10:      log << MSG::INFO  << " Entry [ " << count++ << "]:";
11:      log << " Px=" << px << " Py=" << py << " Pz=" << pz << endreq;
12:    }
13:  }
```

9.2.3 N-tuple Persistency

9.2.3.1 Choice of persistency technology

N-tuples are of special interest to the end-user, because they can be accessed using commonly known tools such as PAW, ROOT or Java Analysis Studio (JAS). In the past it was not a particular strength of the software used in HEP to plug into many possible persistent data representations. Except for JAS, only proprietary data formats are understood. For this reason the choice of the output format of the data depends on the preferred analysis tool/viewer.

HBOOK This data format is used by PAW. PAW can understand this and only this data format. Files of this type can be converted to the ROOT format using the `h2root` data conversion program. The use of PAW in the long term is deprecated.

ROOT This data format is used by the interactive ROOT program.

In the current implementation, N-tuples must use the same persistency technology as histograms. The choice of technology is therefore made in the same way as for histograms, as described in Section 8.4. Obviously the options have to be given only once and are valid for both histograms and N-tuples. The only difference is that histograms are saved to a different output file (defined by the job option `HistogramPersistencySvc.OuputFile`), a different output file (or set of output files) must be defined for the N-tuples.

9.2.3.2 Input and Output File Specification

Conversion services exist to convert N-tuple objects into a form suitable for persistent storage in a number of storage technologies. In order to use this facility it is necessary to add the following line in the job options file:

```
NTupleSvc.Output = { "FILE1 DATAFILE='tuples.hbook' OPT='NEW' ",
                     "FILE2 ...",
                      ...
                     "FILEN ..."};
```

where `<tuples.hbook>` should be replaced by the name of the file to which you wish to write the N-tuple. FILE1 is the logical name of the output file, which must be the same as the data store directory name described in Section 9.2.1. Several files are possible, corresponding to different data store directories whose name can be chosen at will.

The detailed syntax of the options is as follows. In each case only the three leading characters are significant: DATAFILE=<...>, DATABASE=<...> or simply DATA=<...> would lead to the same result.

- **`DATAFILE='<file-specs>'`**
  Specifies the datafile (file name) of the output stream.
- **OPT= <opt-spec>**
  - NEW, CREATE, WRITE: Create a new data file. Not all implementations allow to over-write existing files.
  - OLD, READ: Access an existing file for read purposes
  - UPDATE: Open an existing file and add records. It is not possible to update already existing records.

A similar option `NTupleSvc.Input` exists for N-tuple input.

9.2.3.3 Saving row wise N-tuples in HBOOK

Since the persistent representation of row wise N-tuples in HBOOK is done by floats only, a convention is needed to access the proper data type. By default the `float` type is assumed, i.e. all data members are of `float` type. This is the recommended format.

It is possible to define row wise N-tuples in Athena containing data types other than `float`. This was the default in Gaudi versions previous to v8, where the first row of the N-tuple contained the type information. This possibility can be switched on by using the option

HistogramPersistencySvc.RowWiseNTuplePolicy = "USE_DATA_TYPES";

which also provides backwards compatibility for reading back old N-tuples produced with old Gaudi versions. Remember however that when using PAW to read N-tuples produced using this option, you must skip the first row and start with the second event.

9.3 Event Collections

Event collections or, to be more precise, event tag collections, are used to minimize data access by performing preselections based on small amounts of data. Event tag data contain flexible event classification information according to the physics needs. This information could either be stored as flags indicating that the particular event has passed some preselection criteria, or as a small set of parameters which describe basic attributes of the event. Fast access is required for this type of event data.

Event tag collections can exist in several versions:

- Collections recorded during event processing stages from the online, reconstruction, reprocessing etc.
- Event collections defined by analysis groups with pre-computed items of special interest to a given group.
- Private user defined event collections.

Starting from this definition an event tag collection can be interpreted as an N-tuple which allows to access the data used to create the N-tuple. Using this approach any N-tuple which allows access to the data is an event collection.

Event collections allow pre-selections of event data. These pre-selections depend on the underlying storage technology.

First stage pre-selections based on scalar components of the event collection. First stage preselection is not necessarily executed on your computer but on a database server e.g. when using ORACLE. Only the accessed columns are read from the event collection. If the criteria are fulfilled, the N-tuple data are returned to the user process. Preselection criteria are set through a job options, as described in section 9.3.2.3.

The **second stage pre-selection** is triggered for all items which passed the first stage pre-selection criteria. For this pre-selection, which is performed on the client computer, all data in the N-tuple can be used. The further preselection is implemented in a user defined function object (functor) as described in section 9.3.2.3. Athena algorithms are called only when this pre-selector also accepts the event, and normal event processing can start.

9.3.1 Writing Event Collections

Event collections are written to the data file using a Athena sequencer. A sequencer calls a series of algorithms, as discussed in section 3.2. The execution of these algorithms may terminate at any point of the series (and the event not selected for the collection) if one of the algorithms in the sequence fails to pass a filter.

9.3.1.1 Defining the Address Tag

The event data is accessed using a special N-tuple tag of the type

```
NTuple::Item<IOpaqueAddress*> m_evtAddress
```

It is defined in the algorithm s header file in addition to any other ordinary N-tuple tags, as described in section 9.2.2.1. When booking the N-tuple, the address tag must be declared like any other tag, as shown in Listing 9.5. It is recommended to use the name "`Address`" for this tag.

Listing 9.5 Connecting an address tag to an event collection N-tuple.

```
1:    StatusCode status = service("EvtTupleSvc", m_evtTupleSvc);
2:    if ( status.isSuccess() )   {
3:      NTuplePtr nt(m_evtTupleSvc, "/NTUPLES/EvtColl/Collection");
4:     ... Book N-tuple
5:     // Add an event address column
6:     status = nt->addItem ("Address", m_evtAddress);
```

The usage of this tag is identical to any other tag except that it only accepts variables of type `IOpaqueAddress` - the information necessary to retrieve the event data.

Please note that the event tuple service necessary for writing event collections is not instantiated by default and hence must be specified in the job options file:

Listing 9.6 Adding the event tag collection service to the job options.

```
1:  ApplicationMgr.ExtSvc += {  "TagCollectionSvc/EvtTupleSvc" };
```

It is up to the user to locally remember within the algorithm writing the event collection tuple the reference to the corresponding service. Although the `TagCollectionSvc` looks like an N-tuple service, the implementation is different.

9.3.1.2 Filling the Event Collection

At fill time the address of the event must be supplied to the `Address` item. Otherwise the N-tuple may be written, but the information to retrieve the corresponding event data later will be lost. Listing 9.7 also demonstrates the setting of a filter to steer whether the event is written out to the event collection.

Listing 9.7 Fill the address tag of an N-tuple at execution time:

```
1:    SmartDataPtr<Event> evt(eventSvc(),"/Event");
2:    if ( evt )    {
3:      ... Some data analysis deciding whether to keep the event or not
4:      // keep_event=true if event should be written to event collection
5:      setFilterPassed( keep_event );
6:      m_evtAddrColl = evt->address();
7:    }
```

9.3.2 Event Collection Persistency

9.3.2.1 Output File Specification

Conversion services exist to convert event collection objects into a form suitable for persistent storage in a number of storage technologies. In order to use this facility it is necessary to add the following line in the job options file:

```
EvtTupleSvc.Output = {"FILE1 DATAFILE='coll.root' TYP='ROOT' OPT='NEW'",
                      "FILE2 ...",
                       ...
                      "FILEN ..."};
```

where `<coll.root>` should be replaced by the name of the file to which you wish to write the event collection. FILE1 is the logical name of the output file - it could be any other string.

The options are the same as for N-tuples (see Section 9.2.3.2) with the following additions:

- **TYP= <typ-spec>**
  Specifies the type of the output stream. Currently supported types are:
  - **ROOT: Write as a ROOT tree.**
  - **MS Access: Write as a Microsoft Access database.**
  - **There is also weak support for the following database types1:**
    - **SQL Server**
    - **MySQL**
    - **Oracle ODBC**

      These database technologies are supported through their ODBC interface. They were tested privately on local installations. However all these types need special setup to grant access to the database.

    You need to specify the use of the technology specific persistency package (i.e. GaudiRootDb) in your CMT requirements file and to load explicitly in the job options the DLLs containing the generic (GaudiDb) and technology specific (e.g. GaudiRootDb) implementations of the database access drivers:

    ApplicationMgr.DLLs += { "GaudiDb", "GaudiRootDb" };
- **SVC= <service-spec> (optional)**

  Connect this file directly to an existing conversion service. This option however needs special care. It should only be used to replace default services.

- **AUTHENTICATION= <authentication-specs> (optional)**

  For protected datafiles (e.g. Microsoft Access) it can happen that the file is password protected. In this case the authentication string allows to connect to these databases. The connection string in this case is the string that must be passed to ODBC, for example: AUTH= SERVER=server_host;UID=user_name;PWD=my_password;

- All other options are passed without any interpretation directly to the conversion service responsible to handle the specified output file.

For all options at most three leading characters are significant: DATAFILE=<...>, DATABASE=<...> or simply DATA=<...> would lead to the same result. Additional options are availible when accessing events using an event tag collection.

9.3.2.2 Writing out the Event Collection

The event collection is written out by an `EvtCollectionStream`, which is the last member of the event collection `Sequencer`. Listing 9.8 (which is taken from the job options of `EvtCollection` example), shows how to set up such a sequence consisting of a user written `Selector` algorithm (which could for example contain the code in Listing 9.7), and of the `EvtCollectionStream`.

Listing 9.8 Job options for writing out an event collection

```
1: ApplicationMgr.OutStream   = { "Sequencer/EvtCollection" };
2: ApplicationMgr.ExtSvc += { "TagCollectionSvc/EvtTupleSvc" };
3: EvtCollection.Members = { "EvtCollectionWrite/Selector",
   "EvtCollectionStream/Writer"};
4: Writer.ItemList   = { "/NTUPLES/EvtColl/Collection" };
5: Writer.EvtDataSvc = "EvtTupleSvc";
6: EvtTupleSvc.Output = { "EvtColl DATAFILE='MyEvtCollection.root'
                         OPT='NEW' TYP='ROOT'" };
```

9.3.2.3 Reading Events using Event Collections

Reading event collections as the input for further event processing in Gaudi is transparent. The main change is the specification of the input data to the event selector:

Listing 9.9 Connecting an address tag to an N-tuple.

```
1: EventSelector.Input = {
2:   "COLLECTION='Collection' ADDRESS='Address'
   DATAFILE='MyEvtCollection.root' TYP='ROOT' SEL='(Ntrack>80)'
   FUN='EvtCollectionSelector'"
3: };
```

The tags that were not already introduced earlier are:

¥ **COL**LECTION
Specifies the sub-path of the N-tuple used to write the collection. If the N-tuple which was written was called e.g. "/NTUPLES/FILE1/Collection", the value of this tag must be "Collection".

¥ **ADD**RESS **(optional)**
Specifies the name of the N-tuple tag which was used to store the opaque address to be used to retrieve the event data later. This is an optional tag, the default value is "`Address`". Please use this default value when writing, conventions are useful!

¥ **SEL**ECTION **(optional)**:
Specifies the selection string used for the first stage pre-selection. The syntax depends on the database implementation; it can be:

  ¥ SQL like, if the event collection was written using ODBC.
  Example: `(NTrack>200 AND Energy>200)`

  ¥ C++ like, if the event collection was written using ROOT.
  Example: `(NTrack>200 && Energy>200).`
  Note that event collections written with ROOT also accept the SQL operators `AND` instead of `&&` as well as `OR` instead of `||`. Other SQL operators are not supported.

¥ **FUN**CTION **(optional)**
Specifies the name of a function object used for the second-stage preselection. An example of a such a function object is shown in Listing 9.10. Note that the factory declaration on line 16 is mandatory in order to allow Gaudi to instantiate the function object.

¥ The **DATAFILE** and **TYP** tags, as well as additional optional tags, have the same meaning and syntax as for N-tuples, as described in section 9.2.3.2.

Listing 9.10 Example of a function object for second stage pre-selections.

```
 1: class EvtCollectionSelector : public NTuple::Selector  {
 2:   NTuple::Item<long> m_ntrack;
 3: public:
 4:   EvtCollectionSelector(IInterface* svc) : NTuple::Selector(svc)  {  }
 5:   virtual ~EvtCollectionSelector()   {  }
 6:   /// Initialization
 7:   virtual StatusCode initialize(NTuple::Tuple* nt)    {
 8:     return nt->item("Ntrack", m_ntrack);
 9:   }
10:   /// Specialized callback for NTuples
11:   virtual bool operator()(NTuple::Tuple* nt)    {
12:     return m_ntrack>cut;
13:   }
14: };
15:
16: ObjectFactory<EvtCollectionSelector> EvtCollectionSelectorFactory
```

9.3.3 Interactive Analysis using Event Tag Collections

Event tag collections are very similar to N-tuples and hence allow within limits some interactive analysis.

9.3.3.1 ROOT

This data format is used by the interactive ROOT program. Using the helper library `TBlob` located in the package `GaudiRootDb` it is possible to interactively analyse the N-tuples written in ROOT format. However, access is only possible to scalar items (`int`, `float`, ...) not to arrays.

Analysis is possible through directly plotting variables:

```
root [1] gSystem->Load("D:/mycmt/GaudiRootDb/v3/Win32Debug/TBlob");
root [2] TFile* f = new TFile("tuple.root");
root [3] TTree* t = (TTree*)f->Get("<local>_MC_ROW_WISE_2");
root [4] t->Draw("pz");
```

or using a ROOT macro interpreted by ROOT s C/C++ interpreter (see for example the code fragment `interactive.C` shown in Listing 9.11):

```
root [0] gSystem->Load("D:/mycmt/GaudiRootDb/v3/Win32Debug/TBlob");
root [1] .L ./v8/NTuples/interactive.C
root [2] interactive("./v8/NTuples/tuple.root");
```

More detailed explanations can be found in the ROOT tutorials (*http://root.cern.ch*).

Listing 9.11 Interactive analysis of ROOT N-tuples: interactive.C

```
 1: void interactive(const char* fname)  {
 2:    TFile *input = new TFile(fname);
 3:    TTree *tree = (TTree*)input->Get("<local>_MC_ROW_WISE_2");
 4:    if ( 0 == tree )    {
 5:      printf("Cannot find the requested tree in the root file!\n");
 6:      return;
 7:    }
 8:    Int_t           ID, OBJSIZE, NUMLINK, NUMSYMB;
 9:    TBlob           *BUFFER = 0;
10:    Float_t         px, py, pz;
11:    tree->SetBranchAddress("ID",&ID);
12:    tree->SetBranchAddress("OBJSIZE",&OBJSIZE);
13:    tree->SetBranchAddress("NUMLINK",&NUMLINK);
14:    tree->SetBranchAddress("NUMSYMB",&NUMSYMB);
15:    tree->SetBranchAddress("BUFFER", &BUFFER);
16:    tree->SetBranchAddress("px",&px);
17:    tree->SetBranchAddress("py",&py);
18:    tree->SetBranchAddress("pz",&pz);
19:    Int_t nbytes = 0;
20:    for (Int_t i = 0, nentries = tree->GetEntries(); i<nentries;i++) {
21:       nbytes += tree->GetEntry(i);
22:       printf("Trk#=%d PX=%f PY=%f PZ=%f\n",i,px,py,pz);
23:    }
24:    printf("I have read a total of %d Bytes.\n", nbytes);
25:    delete input;
26: }
```

¥

Chapter 10
Framework services

10.1 Overview

`Services` are generally sizeable components that are setup and initialized once at the beginning of the job by the framework and used by many algorithms as often as they are needed. It is not desirable in general to require more than one instance of each service. Services cannot have a state because there are many potential users of them so it would not be possible to guarantee that the state is preserved in between calls.

In this chapter we describe how services are created and accessed, and then give an overview of the various services, other than the data access services, which are available for use within the Athena framework. The *Job Options* service, the *Message* service, the *Particle Properties* service, the *Chrono & Stat* service, the *Auditor* service, the *Random Numbers* service, the *Incident* service and the *Introspection* service are available in this release. The *Tools* service is described in <Tools Chapter>.

We also describe how to implement new services for use within the Athena environment. We look at how to code a service, what facilities the `Service` base class provides and how a service is managed by the application manager.

10.2 Requesting and accessing services

The Application manager only creates by default the `JobOptionsSvc` and `MessageSvc`. Other services are created on demand the first time they are accessed, provided the corresponding DLL has been loaded. The services in the GaudiSvc package are accessible in this way by default - these are the default data store services (`EventDataSvc`, `DetectorDataSvc`, `HistogramDataSvc`, `NTupleSvc`) and many of the framework services described in this chapter and in <Tools Chapter>

(ToolSvc, ParticlePropertySvc, ChronoStatSvc, AuditorSvc, RndmGenSvc, IncidentSvc).

Additional services can be made accessible by loading the appropriate DLL, using the property ApplicationMgr.DLLs in the job options file, as shown for example in Listing 7.6 on page 12.

Sometimes it may be necessary to force the Application Manager to create a service at initialisation (for example if the order of creation is important). This can be done using the property ApplicationMgr.ExtSvc. In the example below this option is used to create a specific type of persistency service.:

Listing 10.1 Job Option to create additional services

```
ApplicationMgr.ExtSvc += { "DbEventCnvSvc/RootEvtCnvSvc" };
```

Once created, services must be accessed via their interface. The Algorithm base class provides a number of accessor methods for the standard framework services, listed on lines <line24> to <line35> of <Listing 5.1> on <page??>. Other services can be located using the templated service function. In the example below we use this function to return the IParticlePropertySvc interface of the Particle Properties Service: The third argument is optional: when set to true, the service will be

Listing 10.2 Code to access the IParticlePropertySvc interface from an Algorithm

```
#include "GaudiKernel/IParticlePropertySvc.h"
...
IParticlePropertySvc* m_ppSvc;
StatusCode sc = service( "ParticlePropertySvc", m_ppSvc, true );
if ( sc.isFailure) {
...
```

created if it does not already exist; if it is missing, or set to false, the service will not be created if it is not found, and an error is returned.

In components other than Algorithms and Services (e.g. Tools, Converters), which do not provide the service function, you can locate a service using the serviceLocator function:

```
#include "GaudiKernel/IParticlePropertySvc.h"
...
IParticlePropertySvc* m_ppSvc;
IService* theSvc;

StatusCode sc=serviceLocator()->getService("ParticlePropertySvc",theSvc,true);
if ( sc.isSuccess() ) {
  sc = theSvc->queryInterface(IID_IParticlePropertySvc, (void**)&m_ppSvc);
}
if ( sc.isFailure) {
...
```

10.3 The Job Options Service

The Job Options Service is a mechanism which allows to configure an application at run time, without the need to recompile or relink. The options, or properties, are set via a job options file, which is read in when the Job Options Service is initialised by the Application Manager. In what follows we describe the format of the job options file, including some examples.

10.3.1 Algorithm, Tool and Service Properties

In general a concrete Algorithm, Service or Tool will have several data members which are used to control execution. These data members (*properties*) can be of a basic data type (`int`, `float`, etc.) or class (`Property`) encapsulating some common behaviour and higher level of functionality. Each concrete Algorithm, Service, Tool declares its properties to the framework using the `declareProperty` templated method as shown for example on line 12 of Listing 10.4 (see also <Section 5.2>). The method `setProperties()` is called by the framework in the initialization phase; this causes the job options service to make repeated calls to the `setProperty()` method of the Algorithm, Service or Tool (once for each property in the job options file), which actually assigns values to the data members.

10.3.1.1 SimpleProperties

Simple properties are a set of classes that act as properties directly in their associated Algorithm, Tool or Service, replacing the corresponding basic data type instance. The primary motivation for this is to allow optional bounds checking to be applied, and to ensure that the Algorithm, Tool or Service itself doesn’t violate those bounds.Available SimpleProperties are:

¥ `int` ==> `IntegerProperty` or `SimpleProperty<int>`

¥ `double` ==> `DoubleProperty` or `SimpleProperty<double>`

¥ `bool` ==> `BooleanProperty` or `SimpleProperty<bool>`)

¥ `std::string` ==> `StringProperty` or `SimpleProperty<std::string>`

and the equivalent vector classes

¥ `std::vector<int>` ==> `IntegerArrayProperty` or `SimpleProperty<std::vector<int>>`

¥ etc.

The use of these classes is illustrated by the `EventCounter` class (Listings 10.3 and 10.4).

Listing 10.3 EventCounter.h

```
 1: #include "GaudiKernel/Algorithm.h"
 2: #include "GaudiKernel/Property.h"
 3: class EventCounter : public Algorithm {
 4: public:
 5:     EventCounter( const std::string& name, ISvcLocator* pSvcLocator );
 6:     ~EventCounter( );
 7:     StatusCode initialize();
 8:     StatusCode execute();
 9:     StatusCode finalize();
10: private:
11:     IntegerProperty m_frequency;
12:     int m_skip;
13:     int m_total;
14: };
```

Listing 10.4 EventCounter.cpp

```
 1: #include "GaudiAlg/EventCounter.h"
 2: #include "GaudiKernel/MsgStream.h"
 3: #include "GaudiKernel/AlgFactory.h"
 4:
 5: static const AlgFactory<EventCounter>    Factory;
 6: const IAlgFactory& EventCounterFactory = Factory;
 7:
 8: EventCounter::EventCounter(const std::string& name, ISvcLocator*
 9:                            pSvcLocator) :
10:                            Algorithm(name, pSvcLocator),
11:                            m_skip ( 0 ), m_total( 0 ) {
12:   declareProperty( "Frequency", m_frequency=1 );  // [ 1]
13:   m_frequency.setBounds( 0, 1000 );               // [ 2]
14: }
15:
16: StatusCode EventCounter::initialize() {
17:   MsgStream log(msgSvc(), name());
18:   log << MSG::INFO << "Frequency: " << m_frequency << endreq;    // [ 3]
19:   return StatusCode::SUCCESS;
20: }
```

Notes:
1. A default value may be specified when the property is declared.
2. Optional upper and lower bounds may be set (see later).
3. The value of the property is accessible directly using the property itself.

In the Algorithm constructor, when calling `declareProperty`, you can optionally set the bounds using any of:

```
setBounds( const T& lower, const T& upper );
setLower ( const T& lower );
```

```
setUpper ( const T& upper );
```

There are similar selectors and modifiers to determine whether a bound has been set etc., or to clear a bound.

```
bool hasLower( )
bool hasUpper( )
T lower( )
T upper( )
void clearBounds( )
void clearLower( )
void clearUpper( )
```

You can set the value using the "=" operator or the set functions

```
bool set( const T& value )
bool setValue( const T& value )
```

The function value indicates whether the new value was within any bounds and was therefore successfully updated. In order to access the value of the property, use:

```
m_property.value( );
```

In addition there's a cast operator, so you can also use `m_property` directly instead of `m_property.value()`.

10.3.1.2 CommandProperty

`CommandProperty` is a subclass of `StringProperty` that has a handler that is called whenever the value of the property is changed. Currently that can happen only during the job initialization so it is not terribly useful. Alternatively, an Algorithm could set the property of one of its sub-algorithms. However, it is envisaged that Athena will be extended with a scripting language such that properties can be modified during the course of execution.

The relevant portion of the interface to `CommandProperty` is:

```
class CommandProperty : public StringProperty {
public:
  [ ...]
  virtual void handler( const std::string& value ) = 0;
  [ ...]
};
```

Thus subclasses should override the `handler()` member function, which will be called whenever the property value changes. A future development is expected to be a ParsableProperty (or something similar) that would offer support for parsing the string.

10.3.2 Accessing and modifiying properties

Properties are private data which are initialised by the framework using the default values given when they are declared in constructors, or the values read from the job options file. On occasions it may be necessary for components to access (or even modify) the values of properties of other components. This can be done by using the `getProperty()` and `setProperty()` methods of the `IProperty` interface. In the example below,,an algorithm stores the default value of a cut of its sub-algorithm, then

```
Algorithm* myAlg;
...
std:string dfltCut;
StatusCode sc = myAlg->getProperty( "TheCut", dfltCut );
if ( sc.isSuccess() ) {
  msgAlg->setProperty( "TheCut", "0.8" );
  StatusCode sc1 = myAlg->execute();
  ...
}
if( scl.isSuccess() ) msgProp->setProperty( "The Cut", dfltCut );
```

executes the sub-algorithm with a different cut, before resetting the cut back to its default value. Note that in the example we begin with a pointer to an `Algorithm` object, not an `IAlgorithm` interface. This means that we have access to the methods of both the `IAlgorithm` and `IProperty` interfaces and can therefore call the methods of the `IProperty` interface. In the general one may need to navigate to the `IProperty` interface first, as explaned in <Section 1.6>.

10.3.3 Job options file format

An example of a job options file was shown in <Section 4.2> on <Page 29>. The job options file has a well-defined syntax (similar to a simplified C++-Syntax) without data types. The data types are recognised by the Job Options Compiler , which interprets the job options file according to the syntax (described in <Appendix C> together with possible compiler error codes).

The job options file is an ASCII-File, composed logically of a series of statements. The end of a statement is signalled by a semicolon ; - as in C++.

Comments are the same as in C++, with // until the end of the line, or between /* and */ .

There are four constructs which can be used in a job options file:

¥ Assignment statement

¥ Append statement

¥ Include directive

¥ Platform dependent execution directive

10.3.3.1 Assignment statement

An assignment statement assigns a certain value (or a vector of values) to a property of an object or identifier. An assignment statement has the following structure:

```
<Object / Identifier> . < Propertyname > = < value >;
```

The first token (`Object / Identifier`) specifies the name of the object whose property is to be set. This must be followed by a dot (.)

The next token (`Propertyname`) is the name of the option to be set, as declared in the `declareProperty()` method of the `IProperty` interface. This must be followed by an assign symbol (=).

The final token (`value`) is the value to be assigned to the property. It can be a vector of values, in which case the values are enclosed in array brackets ({ , }), and separated by commas (,). The token must be terminated by a semicolon (;).

The type of the value(s) must match that of the variable whose value is to be set, as declared in `declareProperty()`. The following types are recognised:

Boolean-type, written as true or false.
e.g. `true; false;`

Integer-type, written as an integer value (containing one or more of the digits 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9)
e.g.: `123; -923;` or in scientific notation, e.g.: `12e2;`

Real-type (similar to `double` in C++), written as a real value (containing one or more of the digits 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 followed by a dot `.` and optionally one or more of digits again)
e.g.: `123.; -123.45;` or in scientific notation, e.g. `12.5e7;`

String type, written within a pair of double quotes ( )
e.g.: `“I am a string”;` (Note: strings without double quotes are not allowed!)

Vector of the types above, within array-brackets ({ , }), separated by a comma (,)
e.g.: `{true, false, true};`
e.g.: `{124, -124, 135e2};`
e.g.: `{123.53, -23.53, 123., 12.5e2};`
e.g.: `{“String 1”, “String 2”, “String 3”};`

A single element which should be stored in a vector must be within array-brackets without a comma

e.g. `{true};`
e.g. `{“String”};`

A vector which has already been defined earlier in the file (or in included files) can be reset to an empty vector
e.g. `{} ;`

10.3.3.2 Append Statement

Because of the possibility of including other job option files (see below), it is sometimes necessary to extend a vector of values already defined in the other job option file. This functionality is provided be the append statement.

An append statement has the following syntax:

```
<Object / Identifier> . < Propertyname > += < value >;
```

The only difference from the assignment statement is that the append statement requires the `+=` symbol instead of the `=` symbol to separate the `Propertyname` and `value` tokens.

The value must be an array of one or more values

e.g. `{true};`
e.g. `{“String”};`
e.g.: `{true, false, true};`
e.g.: `{124, -124, 135e2};`
e.g.: `{123.53, -23.53, 123., 12.5e2};`
e.g.: `{“String 1”, “String 2”, “String 3”};`

The job options compiler itself tests if the object or identifier already exists (i.e. has already been defined in an included file) and the type of the existing property. If the type is compatible and the object exists the compiler appends the value to the existing property. If the property does not exist then the append operation "+=" behaves as assignment operation `=`.

10.3.3.3 Including other Job Option Files

It is possible to include other job option files in order to use pre-defined options for certain objects. This is done using the `#include` directive:

```
#include “filename.opts”
```

The `filename` can also contain the path where this file is located. By convention we use "`.opts`" as the file extension for job options. The include directive can be placed anywhere in the job option file, usually at the top (as in C++). Note that the value of a property defined earlier in the file may be over-ridden by assigning a new value to the same property: the last value assigned is the valid value! This makes it possible to over-ride the value of a property defined in a previously included file without changing the include file.

It is possible to use environment variables in the `#include` statement, either standalone or as part of a string. Both Unix style (`$environmentvariable`) and Windows style (`%environmentvariable%`) are understood (on both platforms!). For example, in line <2>: of <Listing 4.2> the logical name `$STDOPTS`, which is defined in the `GaudiExamples` package, points to a directory containing a number of standard job options include files that can be used by applications.

As mentioned above, you can append values to vectors defined in an included job option file. The interpreter creates these vectors at the moment he interprets the included file, so you can only append elements defined in a file included before the append-statement!

As in C/C++, an included job option file can include other job option files. The compiler checks itself whether the include file has already been included, so there is no need for `#ifndef` statements as in C or C++ to check for multiple inclusion.

10.3.3.4 Platform dependent execution

The possibility exists to execute statements only according to the used platform. Statements within platform dependent clauses are only executed if they are asserted to the current used platform.:

Table 1

#ifdef WIN32 (Platform-Dependent Statement) #else *(optional)* (Platform-Dependent Statement) #endif

Only the variable `WIN32` is defined! An `#ifdef WIN32` will check if the used platform is a Windows platform. If so, it will execute the statements until an `#endif` or an optional `#else`. On non-Windows platforms it will execute the code within `#else` and `#endif`. Alternatively one directly can check for a non-Windows platform by using the `#ifndef WIN32` clause.

10.3.3.5 Switching on/off printing

By default, the Job Options Service prints out the contents of the Job Options files to the standard output destination. The possibility exists to switch off this printing, and to toggle between the two states, as shown below:

```
1: // Switch off printing
2: #pragma print off
3: ..(some job options)
4: //Switch printing back on
5: #pragma print on
```

In the example above, all lines between line 2 and line 5 will not be printed.

10.4 The Standard Message Service

One of the components directly visible to an algorithm object is the message service. The purpose of this service is to provide facilities for the logging of information, warnings, errors etc. The advantage of introducing such a component, as opposed to using the standard `std::cout` and `std::cerr` streams available in C++ is that we have more control over what is printed and where it is printed. These considerations are particularly important in an online environment.

The Message Service is configurable via the job options file to only output messages if their activation level is equal to or above a given output level . The output level can be configured with a global default for the whole application:

Table 2

```
// Set output level threshold
//(1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL, 7=ALWAYS)
MessageSvc.OutputLevel = 4;
```

and/or locally for a given client object (e.g. `myAlgorithm`):

Table 3

```
myAlgorithm.OutputLevel = 2;
```

Any object wishing to print some output should (must) use the message service. A pointer to the `IMessageSvc` interface of the message service is available to an algorithm via the accessor method `msgSvc()`, see section <5.2>. It is of course possible to use this interface directly, but a utility class called `MsgStream` is provided which should be used instead.

10.4.1 The MsgStream utility

The `MsgStream` class is responsible for constructing a `Message` object which it then passes onto the message service. Where the message is ultimately sent to is decided by the message service.

In order to avoid formatting messages which will not be sent because the verboseness level is too high, a `MsgStream` object first checks to see that a message will be printed before actually constructing it. However the threshold for a `MsgStream` object is not dynamic, i.e. it is set at creation time and remains the same. Thus in order to keep synchronized with the message service, which in principle could change its printout level at any time, `MsgStream` objects should be made locally on the stack when needed. For example, if you look at the listing of the `HelloWorld` class (see also Listing 10.5 below) you will note that `MsgStream` objects are instantiated locally (i.e. not using `new`) in all three of the `IAlgorithm` methods and thus are destructed when the methods return. If this is not done messages may be lost, or too many messages may be printed.

The MsgStream class has been designed to resemble closely a normal stream class such as std::cout, and in fact internally uses an ostrstream object. All of the MsgStream member functions write unformatted data; formatted output is handled by the insertion operators.

An example use of the MsgStream class is shown below.

Listing 10.5 Use of a MsgStream object.

```
 1: #include “GaudiKernel/MgsStream.h”
 2:
 3: StatusCode myAlgo::finalize() {
 4:   StatusCode status = Algorithm::finalise();
 5:   MsgStream log(msgSvc(), name());
 6:   if ( status.isFailure() ) {
 7:     // Print a two line message in case of failure.
 8:     log << MSG::ERROR << “ Finalize failed” << endl
 9:         << “Error initializing Base class.” << endreq;
10:   }
11:   else {
12:     log << MSG::DEBUG << “Finalize completed successfully” << endreq;
13:   }
14:   return status;
15: }
```

When using the MsgStream class just think of it as a configurable output stream whose activation is actually controlled by the first word (message level) and which actually prints only when endreq is supplied. For all other functionality simply refer to the C++ ostream class.

The activation level of the MsgStream object is controlled by the first expression, e.g. MSG::ERROR or MSG::DEBUG in the example above. Possible values are given by the enumeration below:

```
enum MSG::Level { VERBOSE, DEBUG, INFO, WARNING, ERROR, FATAL, ALWAYS };
```

Thus the code in Listing 10.5 will produce NO output if the print level of the message service is set higher than MSG::ERROR. In addition if the service s print level is lower than or equal to MSG::DEBUG the Finalize completed successfully message will be printed (assuming of course it was successful).

10.4.1.1 User interface

What follows is a technical description of the part of the MsgStream user interface most often seen by application developers. Please refer to the header file for the complete interface.

Insertion Operator

The MsgStream class overloads the << operator as described below.

MsgStream& operator <<(TYPE arg);

Insertion operator for various types. The argument is only formatted by the stream object if the print level is sufficiently high and the stream is active. Otherwise the insertion operators simply return. Through this mechanism extensive debug printout does not cause large run-time overheads. All common base types such as `char`, `unsigned char`, `int`, `float`, etc. are supported

MsgStream& operator <<(MSG::Level level);

This insertion operator does not format any output, but rather (de)activates the stream s formatting and forwarding engine depending on the value of `level`.

Accepted Stream Manipulators

The `MsgStream` specific manipulators are presented below, e.g. `endreq`: `MsgStream& endreq(MsgStream& stream)`. Besides these, the common `ostream` and ios manipulators such as `std::ends`, `std::endl`,... are also accepted.

endl Inserts a newline sequence. Opposite to the ostream behaviour this manipulator does not flush the buffer. Full name: `MsgStream& endl(MsgStream& s)`

ends Inserts a null character to terminate a string. Full name: `MsgStream& ends(MsgStream& s)`

flush Flushes the stream's buffer but does not produce any output! Full name: `MsgStream& flush(MsgStream& s)`

endreq Terminates the current message formatting and forwards the message to the message service. If no message service is assigned the output is sent to `std::cout`. Full name: `MsgStream& endreq(MsgStream& s)`

endmsg Same as `endreq`

10.5 The Particle Properties Service

The Particle Property service is a utility to find information about a named particle s Geant3 ID, Jetset/Pythia ID, Geant3 tracking type, charge, mass or lifetime. The database used by the service can be changed, but by default is the same as that used by the LHCb SICB program. Note that the units conform to the CLHEP convention, in particular MeV for masses and ns for lifetimes. Any comment to the contrary in the code is just a leftover which has been overlooked!

10.5.1 Initialising and Accessing the Service

This service is created by adding the following line in the Job Options file::

Table 4

```
// Create the particle properties service
ApplicationMgr.ExtSvc += { "ParticlePropertySvc" };
```

Listing 10.2 on page 114 shows how to access this service from within an algorithm.

10.5.2 Service Properties

The Particle Property Service currently only has one property: `ParticlePropertiesFile`. This string property is the name of the database file that should be used by the service to build up its list of particle properties. The default value of this property, on all platforms, is `$LHCBDBASE/cdf/particle.cdf`[1]

10.5.3 Service Interface

The service implements the `IParticlePropertySvc` interface. In order to use it, clients must include the file `GaudiKernel/IParticlePropertySvc.h`.

The service itself consists of one STL vector to access all of the existing particle properties, and three STL maps, one to map particles by name, one to map particles by Geant3 ID and one to map particles by stdHep ID.

Although there are three maps, there is only one copy of each particle property and thus each property must have a unique particle name and a unique Geant3 ID. Particles that are known to Geant but not to stdHep, such as Deuteron, have an artificial stdHep ID using unreserved (>7) digits. Although retrieving particles by name should be sufficient, the second and third maps are there because most often generated data stores a particle s Geant3 ID or stdHep ID, and not the particle s name. These maps speed up searches using the IDs.

1. This is an LHCb specific file. A generic implementation will be available in a future release of Gaudi

The `IParticlePropertySvc` interface provides the following functions:

Listing 10.6 The IParticlePropertySvc interface.

```
// IParticlePropertySvc interface:
// Create a new particle property.
// Input: particle, String name of the particle.
// Input: geantId, Geant ID of the particle.
// Input: jetsetId, Jetset ID of the particle.
// Input: type, Particle type.
// Input: charge, Particle charge (/e).
// Input: mass, Particle mass (MeV).
// Input: tlife, Particle lifetime (ns).
// Return: StatusCode - SUCCESS if the particle property was added.
virtual StatusCode push_back( const std::string& particle, int geantId, int
jetsetId, int type, double charge, double mass, double tlife );

// Create a new particle property.
// Input: pp, a particle property class.
// Return: StatusCode - SUCCESS if the particle property was added.
virtual StatusCode push_back( ParticleProperty* pp );

// Get a const reference to the begining of the map.
virtual const_iterator begin() const;

// Get a const reference to the end of the map.
virtual const_iterator end() const;

// Get the number of properties in the map.
virtual int size() const;

// Retrieve a property by geant id.
// Pointer is 0 if no property found.
virtual ParticleProperty* find( int geantId );

// Retrieve a property by particle name.
// Pointer is 0 if no property found.
virtual ParticleProperty* find( const std::string& name );

// Retrieve a property by StdHep id
// Pointer is 0 if no property found.
virtual ParticleProperty* findByStdHepID( int stdHepId );

// Erase a property by geant id.
virtual StatusCode erase( int geantId );

// Erase a property by particle name.
virtual StatusCode erase( const std::string& name );

// Erase a property by StdHep id
virtual StatusCode eraseByStdHepID( int stdHepId );
```

The IParticlePropertySvc interface also provides some typedefs for easier coding:

Listing 1

```
typedef ParticleProperty* mapped_type;
typedef std::map< int, mapped_type, std::less<int> > MapID;
typedef std::map< std::string, mapped_type, std::less<std::string> >
MapName;
typedef std::map< int, mapped_type, std::less<int> > MapStdHepID;
typedef IParticlePropertySvc::VectPP VectPP;
typedef IParticlePropertySvc::const_iterator const_iterator;
typedef IParticlePropertySvc::iterator iterator;
```

10.5.4 Examples

Below are some extracts of code from the LHCb ParticleProperties example to show how one might use the service:

Listing 10.7 Code fragment to find particle properties by particle name.

```
 // Try finding particles by the different methods
  log << MSG::INFO << "Trying to find properties by Geant3 ID..." << endreq;
  ParticleProperty* pp1 = m_ppSvc->find( 1 );
  if ( pp1 ) log << MSG::INFO << *pp1 << endreq;
  log << MSG::INFO << "Trying to find properties by name..." << endreq;
  ParticleProperty* pp2 = m_ppSvc->find( "e+" );
  if ( pp2 ) log << MSG::INFO << *pp2 << endreq;
  log << MSG::INFO << "Trying to find properties by StdHep ID..." << endreq;
  ParticleProperty* pp3 = m_ppSvc->findByStdHepID( 521 );
  if ( pp3 ) log << MSG::INFO << *pp3 << endreq;
```

Listing 10.8 Code fragment showing how to use the map iterators to access particle properties.

```
// List all properties
log << MSG::DEBUG << "Listing all properties..." << endreq;
for( IParticlePropertySvc::const_iterator i = m_ppSvc->begin();
i != m_ppSvc->end(); i++ ) {
if ( *i ) log << *(*i) << endreq;
}
```

10.6 The Chrono & Stat service

The Chrono & Stat service provides a facility to do time profiling of code (*Chrono* part) and to do some statistical monitoring of simple quantities (*Stat* part). The service is created by default by the Application Manager, with the name ChronoStatSvc and service ID extern const CLID&

`IID_IChronoStatSvc` To access the service from inside an algorithm, the member function `chronoSvc()` is provided. The job options to configure this service are described in <Appendix B>, <Table B-2>.

10.6.1 Code profiling

Profiling is performed by using the `chronoStart()` and `chronoStop()` methods inside the codes to be profiled, e.g:

Listing 2

```
/// ...
IChronoStatSvc* svc = chronoSvc();
///  start
svc->chronoStart( "Some Tag" );
/// here some user code are placed:
...
/// stop
svc->chronoStop( "SomeTag" );
```

The profiling information accumulates under the tag name given as argument to these methods. The service measures the time elapsed between subsequent calls of `chronoStart()` and `chronoStop()` with the same tag. The latter is important, since in the sequence of calls below, only the elapsed time between lines 3 and 5 lines and between lines 7 and 9 lines would be accumulated.:

Listing 3

```
 1:  svc->chronoStop("Tag");
 2:  svc->chronoStop("Tag");
 3:  svc->chronoStart("Tag");
 4:  svc->chronoStart("Tag");
 5:  svc->chronoStop("Tag");
 6:  svc->chronoStop("Tag");
 7:  svc->chronoStart("Tag");
 8:  svc->chronoStart("Tag");
 9:  svc->chronoStop("Tag");
```

The profiling information could be printed either directly using the `chronoPrint()` method of the service, or in the summary table of profiling information at the end of the job.

Note that this method of code profiling should be used only for fine grained monitoring inside algorithms. To profile a complete algorithm you should use the Auditor service, as described in section 10.7.

10.6.2 Statistical monitoring

Statistical monitoring is performed by using the `stat()` method inside user code:

Listing 4

```
1: /// ... Flag and Weight to be accumulated:
2: svc->stat( " Number of Tracks " , Flag , Weight );
```

The statistical information contains the "accumulated" *flag*, which is the sum of all *Flag*s for the given tag, and the "accumulated" *weight*, which is the product of all *Weight*s for the given tag. The information is printed in the final table of statistics.

In some sense the profiling could be considered as statistical monitoring, where the variable *Flag* equals the elapsed time of the process.

10.6.3 `Chrono` and `Stat` helper classes

To simplify the usage of the Chrono & Stat Service, two helper classes were developed: `class Chrono` and `class Stat`. Using these utilities, one hides the communications with Chrono & Stat Service and provides a more friendly environment.

10.6.3.1 Chrono

`Chrono` is a small helper class which invokes the `chronoStart()` method in the constructor and the `chronoStop()` method in the destructor. It must be used as an *automatic local object*.

It performs the profiling of the code between its own creation and the end of the current scope, e.g:

Listing 5

```
1: #include GaudiKernel/Chrono.h
2: /// ...
3: { // begin of the scope
4:    Chrono chrono( chronoSvc() , "ChronoTag" ) ;
5:    /// some codes:
6:     ...
7:    ///
8: } // end of the scope
9: /// ...
```

If the Chrono & Stat Service is not accessible, the *Chrono* object does nothing

10.6.3.2 Stat

`Stat` is a small helper class, which invokes the `stat()` method in the constructor.

Listing 6

```
 1: GaudiKernel/Stat.h
 2: /// ...
 3: Stat stat( chronoSvc() , "StatTag" , Flag , Weight ) ;
 4: /// ...
```

If the Chrono & Stat Service is not accessible, the `Stat` object does nothing.

10.6.4 Performance considerations

The implementation of the Chrono & Stat Service uses two `std::map` containers and could generate a performance penalty for very frequent calls. Usually the penalty is small relative to the elapsed time of algorithms, but it is worth avoiding both the direct usage of the Chrono & Stat Service as well as the usage of it through the `Chrono` or `Stat` utilities inside internal loops:

Listing 7

```
 1: /// ...
 2: {   /// begin of the scope
 3: Chrono chrono( chronoSvc() , "Good Chrono");  /// OK
 4: long double a = 0 ;
 5: for(  long i = 0 ; i < 1000000 ; ++i )
 6:  {
 7:  Chrono chrono( svc , "Bad Chrono");    /// not OK
 8:  /// some codes :
 9:    a += sin( cos( sin( cos( (long double) i ) ) ) );
10:  ///  end of codes
11: Stat   stat  ( svc , "Bad Stat", a ); /// not OK
12:  }
13: Stat   stat  ( svc , "Good Stat", a); /// OK
14: } ///  end of the scope!
15: /// ...
```

10.7 The Auditor Service

The Auditor Service provides a set of auditors that can be used to provide monitoring of various characteristics of the execution of Algorithms. Each auditor is called immediately before and after each call to each Algorithm instance, and can track some resource usage of the Algorithm. Calls that are thus monitored are `initialize()`, `execute()` and `finalize()`, although monitoring can be disabled for any of these for particular Algorithm instances. Only the `execute()` function monitoring is enabled by default.

Several examples of auditors are provided. These are:

¥ *NameAuditor*. This just emits the name of the Algorithm to the Standard Message Service immediately before and after each call. It therefore acts as a diagnostic tool to trace program execution.

¥ *ChronoAuditor*. This monitors the cpu usage of each algorithm and reports both the total and per event average at the end of job.

¥ *MemoryAuditor*. This monitors the state of memory usage during execution of each Algorithm, and will warn when memory is allocated within a call without being released on exit. Unfortunately this will in fact be the general case for Algorithms that are creating new data and registering them with the various transient stores. Such Algorithms will therefore cause warning messages to be emitted. However, for Algorithms that are just reading data from the transient stores, these warnings will provide an indication of a possible memory leak. Note that currently the MemoryAuditor is only available for Linux.

¥ *MemStatAuditor*. The same as MemoryAuditor, but prints a table of memory usage statistics at the end.

10.7.1 Enabling the Auditor Service and specifying the enabled Auditors

The Auditor Service is enabled by the following line in the Job Options file:

Table 5

```
// Enable the Auditor Service
ApplicationMgr.DLLs += { "GaudiAud" };
```

Specifying which auditors are enabled is illustrated by the following example:

Table 6

```
// Enable the NameAuditor and ChronoAuditor
AuditorSvc.Auditors = { "NameAuditor", "ChronoAuditor" };
```

10.7.2 Overriding the default Algorithm monitoring

By default, only monitoring of the Algorithm `execute()` function is enabled by default. This default can be overridden for individual Algorithms by use of the following Algorithm properties:

Table 7

```
// Enable initialize and finalize auditing & disable execute auditing
// for the myAlgorithm Algorithm
myAlgorithm.AuditInitialize = true;
myAlgorithm.AuditExecute    = false;
myAlgorithm.AuditFinalize   = true;
```

10.7.3 Implementing new Auditors

The relevant portion of the `IAuditor` abstract interface is shown below:

Table 8

```
virtual StatusCode beforeInitialize( IAlgorithm* theAlg ) = 0;
virtual StatusCode afterInitialize ( IAlgorithm* theAlg ) = 0;

virtual StatusCode beforeExecute   ( IAlgorithm* theAlg ) = 0;
virtual StatusCode afterExecute    ( IAlgorithm* theAlg ) = 0;

virtual StatusCode beforeFinalize  ( IAlgorithm* theAlg ) = 0;
virtual StatusCode afterFinalize   ( IAlgorithm* theAlg ) = 0;
```

A new Auditor should inherit from the Auditor base class and override the appropriate functions from the IAuditor abstract interface. The following code fragment is taken from the ChronoAuditor:

Table 9

```
#include "GaudiKernel/Auditor.h"

class ChronoAuditor : virtual public Auditor {
public:
   ChronoAuditor(const std::string& name, ISvcLocator* pSvcLocator);
   virtual ~ChronoAuditor();
   virtual StatusCode beforeInitialize(IAlgorithm* alg);
   virtual StatusCode afterInitialize(IAlgorithm* alg);
   virtual StatusCode beforeExecute(IAlgorithm* alg);
   virtual StatusCode afterExecute(IAlgorithm* alg);
   virtual StatusCode beforeFinalize(IAlgorithm* alg);
   virtual StatusCode afterFinalize(IAlgorithm* alg);
};
```

10.8 The Random Numbers Service

When generating random numbers two issues must be considered:

- reproducibility and
- randomness of the generated numbers.

In order to ensure both, Athena implements a single service ensuring that these criteria are met. The encapsulation of the actual random generator into a service has several advantages:

¥ Random seeds are set by the framework. When debugging the detector simulation, the program could start at any event independent of the events simulated before. Unlike the random number generators that were known from CERNLIB, the state of modern generators is no longer defined by one or two numbers, but rather by a fairly large set of numbers. To ensure reproducibility the random number generator must be initialized for every event.

¥ The distribution of the random numbers generated is independent of the random number engine behind. Any distribution can be generated starting from a flat distribution.

¥ The actual number generator can easily be replaced if at some time in the future better generators become available, without affecting any user code.

The implementation of both generators and random number engines are taken from CLHEP. The default random number engine used by Athena is the RanLux engine of CLHEP with a luxury level of 3, which is also the default for Geant4, so as to use the same mechanism to generate random numbers as the detector simulation.

Figure 10.1 shows the general architecture of the Athena random number service. The client interacts with the service in the following way:

¥ The client requests a generator from the service, which is able to produce a generator according to a requested distribution. The client then retrieves the requested generator.

¥ Behind the scenes, the generator service creates the requested generator and initializes the object according to the parameters. The service also supplies the shared random number engine to the generator.

¥ After the client has finished using the generator, the object must be released in order to inhibit resource leaks

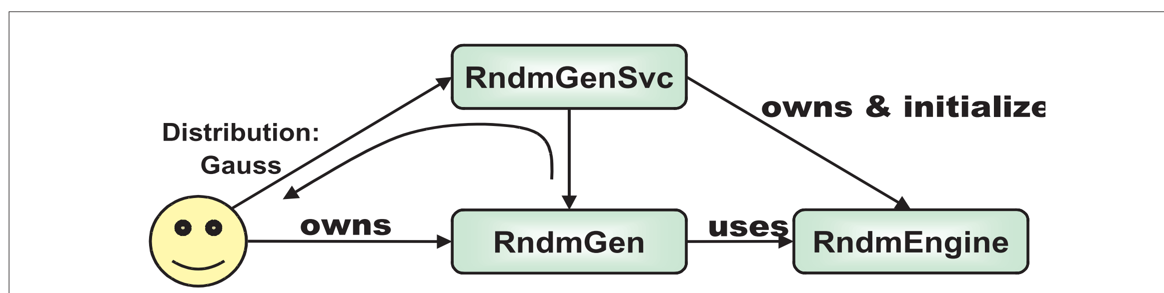

Figure 10.1 The architecture of the random number service. The client requests from the service a random number generator satisfying certain criteria

There are many different distributions available. The shape of the distribution must be supplied as a parameter when the generator is requested by the user.

Currently implemented distributions include the following. See also the header file `GaudiKernel/RndmGenerators.h` for a description of the parameters to be supplied.

¥ Generate random bit patterns with parameters `Rndm::Bit()`

¥ Generate a flat distribution with boundaries [`min, max`] with parameters: `Rndm::Flat(double min, double max)`

- Generate a gaussian distribution with parameters: Rndm::Gauss(double mean, double sigma)
- Generate a poissonian distribution with parameters: Rndm::Poisson(double mean)
- Generate a binomial distribution according to n tests with a probability p with parameters: Rndm::Binomial(long n, double p)
- Generate an exponential distribution with parameters: Rndm::Exponential(double mean)
- Generate a Chi**2 distribution with n_dof degrees of freedom with parameters: Rndm::Chi2(long n_dof)
- Generate a Breit-Wigner distribution with parameters: Rndm::BreitWigner(double mean, double gamma)
- Generate a Breit-Wigner distribution with a cut-off with parameters: Rndm::BreitWignerCutOff (mean, gamma, cut-off)
- Generate a Landau distribution with parameters: Rndm::Landau(double mean, double sigma)
- Generate a user defined distribution. The probability density function is given by a set of descrete points passed as a vector of doubles: Rndm::DefinedPdf(const std::vector<double>& pdf, long intpol)

Clearly the supplied list of possible parameters is not exhaustive, but probably represents most needs. The list only represents the present content of generators available in CLHEP and can be updated in case other distributions will be implemented.

Since there is a danger that the interfaces are not released, a wrapper is provided that automatically releases all resources once the object goes out of scope. This wrapper allows the use of the random number service in a simple way. Typically there are two different usages of this wrapper:

- Within the user code a series of numbers is required only once, i.e. not every event. In this case the object is used locally and resources are released immediately after use. This example is shown in Listing 10.9 .

Listing 10.9 Example of the use of the random number generator to fill a histogram with a Gaussian distribution within a standard Athena algorithm

```
1:  Rndm::Numbers gauss(randSvc(), Rndm::Gauss(0.5,0.2));
2:  if ( gauss )    {
3:    IHistogram1D* his = histoSvc()->book("/stat/2","Gaussian",40,0.,3.);
4:    for ( long i = 0; i < 5000; i++ )
5:      his->fill(gauss(), 1.0);
6:  }
```

¥ One or several random numbers are required for the processing of every event. An example is shown in Listing 10.10.

Listing 10.10 Example of the use of the random number generator within a standard Athena algorithm, for use at every event. The wrapper to the generator is part of the Algorithm itself and must be initialized before being used. Afterwards the usage is identical to the example described in Listing 10.9

```
 1: #include "GaudiKernel/RndmGenerators.h"
 2:
 3: // Constructor
 4: class myAlgorithm : public Algorithm {
 5:   Rndm::Numbers m_gaussDist;
 6: ...
 7: };
 8:
 9: // Initialisation
10: StatusCode myAlgorithm::initialize() {
11: ...
 1:   StatusCode sc=m_gaussDist.initialize(randSvc(), Rndm::Gauss(0.5,0.2));
 2:   if ( !status.isSuccess() ) {
 3:     // put error handling code here...
 4:   }
 5: ...
 6: }
```

There are a few points to be mentioned in order to ensure the reproducibility:

¥ Do not keep numbers across events. If you need a random number ask for it. Usually caching does more harm than good. If there is a performance penalty, it is better to find a more generic solution.

¥ Do not access the RndmEngine directly.

¥ Do not manipulate the engine. The random seeds should only be set by the framework on an event by event basis.

10.9 The Incident Service

The Incident service provides synchronization facilities to components in a Athena application. Incidents are named *software events* that are generated by software components and that are delivered to other components that have requested to be informed when that incident happens. The Athena components that want to use this service need to implement the `IIncidentListener` interface,

which has only one method: `handle(Incident&)`, and they need to add themselves as Listeners to the IncidentSvc. The following code fragment works inside *Algorithms*.

Table 10

```
#include "GaudiKernel/IIncidentListener.h"
#include "GaudiKernel/IIncidentSvc.h"

class MyAlgorithm : public Algorithm, virtual public IIncidentListener {
  ...
};

MyAlgorithm::Initialize() {
  IIncidentSvc* incsvc;
  StatusCode sc = service("IncidentSvc", incsvc);
  int priority = 100;
  if( sc.isSuccess() ) {
     incsvc->addListener( this, "BeginEvent", priority);
     incsvc->addListener( this, "EndEvent");
  }
}

MyAlgorithm::handle(Incident& inc) {
  log << "Got informed of incident: " << inc.type()
      << " generated by: " << inc.source() << endreq;
}
```

The third argument in method `addListener()` is for specifying the priority by which the component will be informed of the incident in case several components are listeners of the same named incident. This parameter is used by the `IncidentSvc` to sort the listeners in order of priority.

10.9.1 Known Incidents

Table 10.1 Table of known named incidents

Incident Type	Source	Description
BeginEvent	ApplicationMgr	The ApplicationMgr is starting processing of a new physics event. This incident can be use to clear caches of the previous event in Services and Tools.
EndEvent	ApplicationMgr	The ApplicationMgr has finished processing the physics event. The Event data store is not yet purged at this moment.

10.10 The Gaudi Introspection Service

Introspection is the ability of a programming language to interact with objects from a meta-level. The Gaudi Introspection package defines a meta-model which gives the layout of this meta-information.

The data to fill this meta-information (i.e. the *dictionary*) can be generated by the Gaudi Object Description package (described in Section 7.7 on page 7) by adding a few lines to the CMT requirements file, as shown for example in Listing 10.11.

Listing 10.11 CMT requirements for generation of data dictionary of the LHCbEvent package

```
#---- dictionary
document obj2dict LHCbEventObj2Dict -group=dict ../xml/LHCbEvent.xml
library LHCbEventDict -group=dict ../dict/*.cpp
macro LHCbEventDict_shlibflags  "$(use_linkopts) $(libraryshr_linkopts)"
```

The C++-code generated in this way is compiled into a dll and loaded into the Gaudi Introspection Model at runtime.

To get a reference to information about a real object, clients have to use the Gaudi Introspection Service (`IntrospectionSvc`). The service can also be used to load the meta-information into the model. The Gaudi Introspection Service is already used in several places in the framework (e.g. Interface to Python, Data Store Browser).

Further information about this service is available at *http://cern.ch/lhcb-comp/Frameworks/DataDictionary/default.htm.*

10.11 Developing new services

10.11.1 The Service base class

Within Athena we use the term "Service" to refer to a class whose job is to provide a set of facilities or utilities to be used by other components. In fact we mean more than this because a concrete service must derive from the `Service` base class and thus has a certain amount of predefined behaviour; for example it has `initialize()` and `finalize()` methods which are invoked by the application manager at well defined times.

Figure 10.1 shows the inheritance structure for an example service called `SpecificService`. The key idea is that a service should derive from the `Service` base class and additionally implement one or more pure abstract classes (interfaces) such as `IConcreteSvcType1` and `IConcreteSvcType2` in the figure.

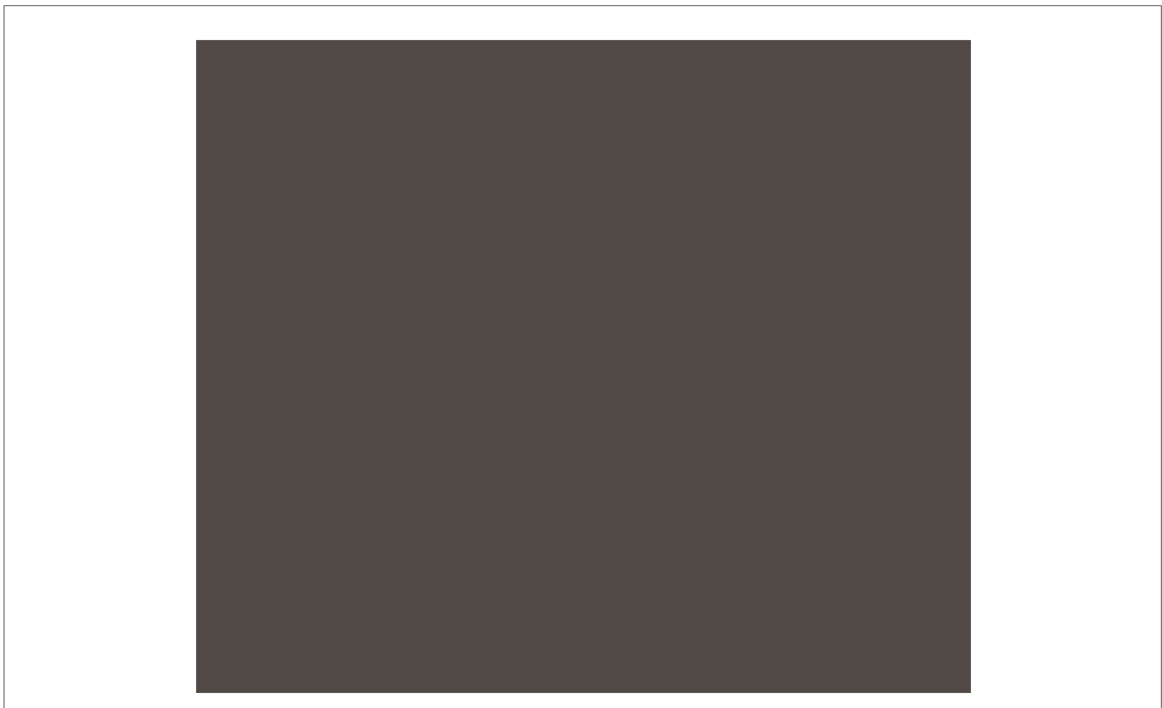

Figure 10.1 Implementation of a concrete service class. Though not shown in the figure, both of the IConcreteSvcType interfaces are derived from IInterface.

As discussed above, it is necessary to derive from the Service base class so that the concrete service may be made accessible to other Athena components. The actual facilities provided by the service are available via the interfaces that it provides. For example the `ParticleProperties` service implements an interface which provides methods for retrieving, for example, the mass of a given particle. In figure 10.1 the service implements two interfaces each of two methods.

A component which wishes to make use of a service makes a request to the application manager. Services are requested by a combination of name, and interface type, i.e. an algorithm would request specifically either `IConcreteSvcType1` or `IConcreteSvcType2`.

The identification of what interface types are implemented by a particular class is done via the `queryInterface` method of the `IInterface` interface. This method must be implemented in the concrete service class. In addition the `initialize()` and `finalize()` methods should be implemented. After initialization the service should be in a state where it may be used by other components.

The service base class offers a number of facilities itself which may be used by derived concrete service classes:

¥ Properties are provided for services just as for algorithms. Thus concrete services may be fine tuned by setting options in the job options file.

¥ A `serviceLocator` method is provided which allows a component to request the use of other services which it may need.

¥ A message service.

10.11.2 Implementation details

The following is essentially a checklist of the minimal code required for a service.

1. Define the interfaces
2. Derive the concrete service class from the `Service` base class.
3. Implement the `queryInterface()` method.
4. Implement the `initialize()` method. Within this method you should make a call to `Service::initialize()` as the first statement in the method and also make an explicit call to `setProperties()` in order to read the service s properties from the job options (note that this is different from Algorithms, where the call to `setProperties()` is done in the base class).

:

Listing 10.12 An interface class

```
#include "GaudiKernel/IInterface.h"

class IConcreteSvcType1 : virtual public IInterface {
public:
  void method1() = 0;
  int method2() = 0;
}
```

Listing 10.12 An interface class

```
#include "IConcreteSvcType1.h"

const IID& IID_IConcreteSvcType1 = 143; // UNIQUE within LHCb !!
```

Listing 10.13 A minimal service implementation

```
#include "GaudiKernel/Service.h"
#include "IConcreteSvcType1.h"
#include "IConcreteSvcType2.h"

class SpecificService : public Service,
                        virtual public IConcreteSvcType1,
                        virtual public IConcreteSvcType2 {
public:
  // Constructor of this form required:
  SpecificService(const std::string& name, ISvcLocator* sl);

  queryInterface(constIID& riid, void** ppvIF);
};
```

Listing 10.13 A minimal service implementation

```
// Factory for instantiation of service objects
static SvcFactory<SpecificService> s_factory;
const ISvcFactory& SpecificServiceFactory = s_factory;

// UNIQUE Interface identifiers defined elsewhere
extern const IID& IID_IConcreteSvcType1;
extern const IID& IID_IConcreteSvcType2;

// queryInterface
StatusCode SpecificService::queryInterface(const IID& riid, void** ppvIF) {
  if(IID_IConcreteSvcType1 == riid) {
    *ppvIF = dynamic_cast<IConcreteSvcType1*> (this);
    return StatusCode::SUCCESS;
  } else if(IID_IConcreteSvcType2 == riid) {
    *ppvIF = dynamic_cast<IConcreteSvcType2*> (this);
    return StatusCode::SUCCESS;
  } else {
    return Service::queryInterface(riid, ppvIF);
  }
}

StatusCode SpecificService::initialize() { ... }
StatusCode SpecificService::finalize() { ... }

// Implement the specifics ...
SpecificService::method1() {...}
SpecificService::method2() {...}
SpecificService::method3() {...}
SpecificService::method4() {...}
```

Chapter 11
Tools and ToolSvc

11.1 Overview

Tools are light weight objects whose purpose is to help other components perform their work. A framework service, the `ToolSvc`, is responsible for creating and managing Tools. An `Algorithm` requests the tools it needs to the `ToolSvc`, specifying if requesting a private instance by declaring itself as the parent. Since Tools are managed by the `ToolSvc`, any component[1] can request a tool. Algorithms, Services and other Tools can declare themselves as Tools parents.

In this chapter we first describe these objects and the difference between private and shared tools. We then look at the `AlgTool` base class and how to write concrete Tools.

In section 11.3 we describe the `ToolSvc` and show how a component can retrieve `Tools` via the service.

Finally we describe Associators, common utility `GaudiTools` for which we provide the interface and base class.

11.2 Tools and Services

As mentioned elsewhere ***Algorithms*** make use of framework services to perform their work. In general the same instance of a service is used by many algorithms and ***Services*** are setup and initialized once at the beginning of the job by the framework. Algorithms also delegate some of their work to sub-algorithms. Creation and execution of sub-algorithms are the responsibilities of the parent

1. In this chapter we will use an Algorithm as example component requesting tools.

algorithm whereas the `initialize()` and `finalize()` methods are invoked automatically by the framework while initializing the parent algorithm. The properties of a sub-algorithm are automatically set by the framework but the parent algorithm can change them during execution. Sharing of data between nested algorithms is done via the Transient Event Store.

Both Services and Algorithms are created during the initialization stage of a job and live until the jobs ends.

Sometimes an encapsulated piece of code needs to be executed only for specific events, in which case it is desirable to create it only when necessary. On other occasions the same piece of code needs to be executed many times per event. Moreover it can be necessary to execute a sub-algorithm on specific contained objects that are selected by the parent algorithm or have the sub-algorithm produce new contained objects that may or may not be put in the Transient Store. Finally different algorithms may wish to configure the same piece of code slightly differently or share it *as-is* with other algorithms.

To provide this kind of functionality we have introduced a category of processing objects that encapsulate these light algorithms. We have called this category ***Tools***.

Some examples of possible tools are single track fitters, association to Monte Carlo truth information, vertexing between particles, smearing of Monte Carlo quantities.

11.2.1 Private and Shared Tools

`Algorithms` can share instances of `Tools with other Algorithms` if the configuration of the tool is suitable. In some cases however an `Algorithm` will need to customize a tool in a specific way in order to use it. This is possible by requesting the `ToolSvc` to provide a *private* instance of a tool.

If an `Algorithm` passes a pointer to itself when it asks the `ToolSvc` to provide it with a `tool`, it is declaring itself as the parent and a *private* instance is supplied. Private instances can be configured according to the needs of each particular `Algorithm`.

As mentioned above many `Algorithms` can use a tool *as-is*, in which case only one instance of a `Tool` is created, configured and passed by the `ToolSvc` to the different algorithms. This is called a *shar ed* instance. The parent of shared tools is the `ToolSvc`.

11.2.2 The Tool classes

11.2.2.1 The AlgTool base class

The main responsibilities of the `AlgTool` base class (see Listing 11.1) are the identification of the tools instances, the initialisation of certain internal pointers when the tool is created and the

management of the tools properties. The `AlgTool` base class also offers some facilities to help in the implementation of derived tools and management of the additional tools interfaces..

Listing 11.1 The definition of the AlgTool Base class. Highlighted in bold are methods relevant for the implementation of concrete tools.

```
 1: class AlgTool : public virtual IAlgTool,
 2:                 public virtual IProperty {
 3:
 4: public:
 5: // Standard Constructor.
 6: AlgTool( const std::string& type, const std::string& name,
             const IInterface* parent);
 7:
 8: ISvcLocator* serviceLocator() const;
 9: IMessageSvc* msgSvc() const;
10:
11: virtual StatusCode setProperty( const Property& p );
12: virtual StatusCode setProperty( std::istream& s );
13: virtual StatusCode setProperty( const std::string& n,
                                    const std::string& v );
14: virtual StatusCode getProperty(Property* p) const;
15: virtual const Property& getProperty( const std::string& name ) const;
16: virtual StatusCode getProperty( const std::string& n,std::string& v )
                      const;
17: virtual const std::vector<Property*>& getProperties( ) const;
18:
19: StatusCode setProperties();
20:
21: template <class T>
22:   StatusCode declareProperty(const std::string& name, T& property) const
23:
24: virtual const std::string& name() const;
25: virtual const std::string& type() const;
26: virtual const IInterface*  parent() const;
27:
28: virtual StatusCode initialize();
29: virtual StatusCode finalize();
30:
31: virtual StatusCode queryInterface(const IID& riid, void** ppvUnknown);
32: void declInterface( const IID&, void*);
33: template <class I> class declareInterface {
      public:
        template <class T> declareInterface(T* tool)
    }
34:
35: protected:
36:   // Standard destructor.
37:   virtual ~AlgTool();
```

Constructor - The base class has a single constructor which takes three arguments. The first is the type (i.e. the class) of the Tool object being instantiated, the second is the full name of the object and the third is a pointer to the `IInterface` of the parent component. The name is used for the identification of the tool instance as described below.The parent interface is used by the tool to access for example the `outputLevel` of the parent.

Access to Services - A `serviceLocator()` method is provided to enable the derived tools to locate the services necessary to perform their jobs. Since concrete `Tools` are instantiated by the `ToolSvc` upon request, all Services created by the framework prior to the creation of a tool are available. In addition access to the message service is provided via the `msgSvc()` method. Both pointers are retrieved from the parent of the tool.

Properties - A template method for declaring properties similarly to `Algorithms` is provided. This allows tuning of data members used by the Tools via JobOptions files. The `ToolSvc` takes care of calling the `setProperties()` method of the `AlgTool` base class after having instantiated a tool. Properties need to be declared in the constructor of a `Tool`. The property `outputLevel` is declared in the base class and is identically set to that of the parent component, unless specified otherwise in the JobOptions. For details on Properties see section 10.3.1.

`IAlgTool` Interface - It consists of three accessor methods for the identification and managment of the tools: `type()`, `name()` and `parent()`. These methods are all implemented by the base class and should not be overridden. Two additional methods, `initialize()` and `finalize()`, allow concrete tools to be configured after creation and orderly terminated before deletion. An empty implementation is provided by the `AlgTool` base class and concrete tools need to implement these methods only when relevant for their purpose. The `ToolSvc` is responsible for calling these methods at the appropriate time.

Tools Interfaces - Concrete tools must implement additional interfaces that will inherit from **`IAlgTool`**. When a component implements more that one interface it is necessary to "recognize" the various interfaces. This is taken care of by the **`AlgTool`** base class once the additional interface is declared by a concrete tool (or tools' base class). The declaration of the additional interface must be done in the constructor of a concrete tool and is done via the template method **`declareInterface.`**

11.2.2.2 Tools identification

A tool instance is identified by its full name. The name consist of the concatenation of the parent name, a dot, and a tool dependent part. The tool dependent part can be specified by the user, when not specified the tool type (i.e. the class) is automatically taken as the tool dependent part of the name. Examples of tool names are `RecPrimaryVertex.VertexSmearer` (a private tool) and `ToolSvc.AddFourMom` (a shared tool). The full name of the tool has to be used in the jobOptions file to set its properties.

11.2.2.3 Concrete tools classes

Operational functionalities of tools must be provided in the derived tool classes. A concrete tool class must inherit directly or indirectly from the `AlgTool` base class to ensure that it has the predefined behaviour needed for management by the `ToolSvc`.

Concrete tools must implement additional interfaces, specific to the task a tool is designed to perform. Specialised tools intended to perform similar tasks can be derived from a common base class that will provide the common functionality and implement the common interface. Consider as example the vertexing of particles, where separate tools can implement different algorithms but the arguments passed are the same. The `ToolSvc` interacts with specialized tools only through the additional tools interface, therefore the interface itself must inherit from the `IAlgTool` interface in order for the tool to be correctly managed by the `ToolSvc`.

The inheritance structure of derived tools is shown in Figure 11.1. `ConcreteTool1` implements one additional abstract interface while `ConcreteTool2` and `ConcreteTool3` derive from a base class `SubTool` that provides them with additional common functionality.

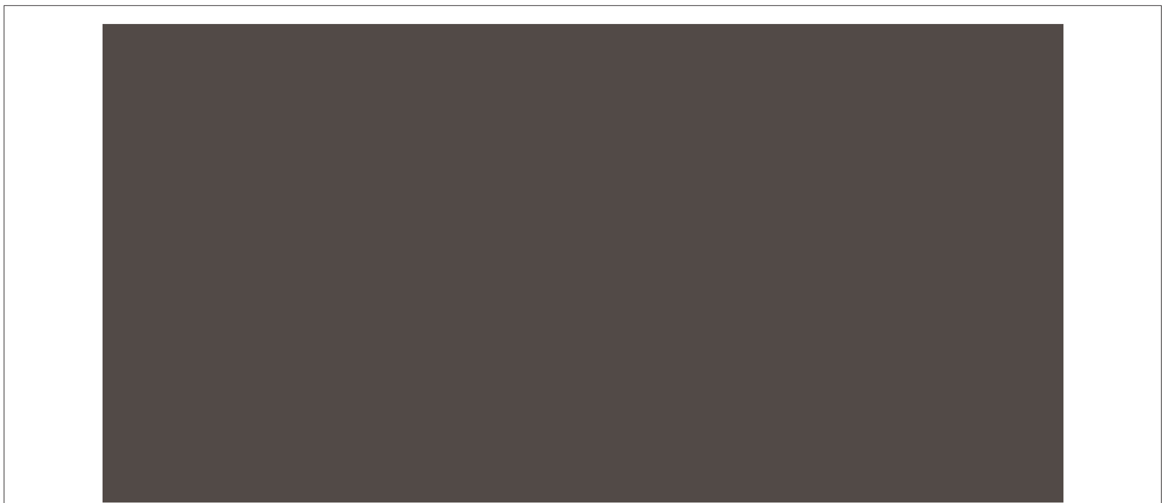

Figure 11.1 Tools class hierarchy

11.2.2.4 Implementation of concrete tools

An example minimal implementation of a concrete tool is shown in Listings 11.2, 11.3 and 11.4, taken from the LHCb `ToolsAnalysis` example application ..

Listing 11.2 Example of a concrete tool additional interface

```
1: static const InterfaceID IID_IVertexSmearer("IVertexSmearer", 1 , 0);
2:
3: class IVertexSmearer : virtual public IAlgTool {
4: public:
5:   /// Retrieve interface ID
6:   static const InterfaceID& interfaceID() { return IID_IVertexSmearer; }
7:   // Actual operator function
8:   virtual StatusCode smear( MyAxVertex* ) = 0;
9: };
```

Listing 11.3 Example of a concrete tool minimal implementation header file

```
 1:  #include "GaudiKernel/AlgTool.h"
 2:  class VertexSmearer : public AlgTool, virtual public IVertexSmearer {
 3:  public:
 4:    // Constructor
 5:    VertexSmearer( const std::string& type, const std::string& name,
                      const IInterface* parent);
 6:    // Standard Destructor
 7:    virtual ~VertexSmearer() { }
 8:    // specific method of this tool
 9:    StatusCode smear( MyAxVertex* pvertex );
```

Listing 11.4 Example of a concrete tool minimal implementation file

```
 1:  #include "GaudiKernel/ToolFactory.h"
 2:  // Static factory for instantiation of algtool objects
 3:  static ToolFactory<VertexSmearer> s_factory;
 4:  const IToolFactory& VertexSmearerFactory = s_factory;
 5:
 6:  // Standard Constructor
 7:  VertexSmearer::VertexSmearer(const std::string& type,
                                  const std::string& name,
                                  const IInterface* parent)
                            : AlgTool( type, name, parent ) {
 8:
 9:    // Locate service needed by the specific tool
10:    m_randSvc = 0;
11:    if( serviceLocator() ) {
12:      StatusCode sc=StatusCode::FAILURE;
13:      sc = serviceLocator()->service( "RndmGenSvc", m_randSvc, true );
14:    }
15:    // Declare additional interface
16:    declareInterface<IVertexSmearer>(this);
17:
18:    // Declare properties of the specific tool
19:    declareProperty("dxVtx", m_dxVtx = 9 * micrometer);
20:    declareProperty("dyVtx", m_dyVtx = 9 * micrometer);
21:    declareProperty("dzVtx", m_dzVtx = 38 * micrometer);
22:  }
23:  // Implement the specific method ....
24:  StatusCode VertexSmearer::smear( MyAxVertex* pvertex ) {...}
```

The creation of concrete tools is similar to that of Algorithms, making use of a Factory Method. As for Algorithms, Tool factories enable their creator to instantiate new tools without having to include any of the concrete tools header files. A template factory is provided and a tool developer will only need to add the concrete factory in the implementation file as shown in lines 1 to 4 of Listing 11.4

In addition a concrete tool class must specify a single constructor with the same parameter signatures as the constructor of the `AlgTool` base class as shown in line 5 of Listing 11.3.

Below is the minimal checklist of the code necessary when developing a Tool:

1. Define the specific interface (inheriting from the IAlgTool interface).
2. Derive the tool class from the AlgTool base class
3. Provide the constructor
4. Declare the additional interface in the constructor.
5. Implement the factory adding the lines of code shown in Listing 11.4
6. Implement the specific interface methods.

In addition if a tool requires special initialization and termination you can implement the initialize and finalize methods.

11.3 The ToolSvc

The ToolSvc manages Tools. It is its responsibility to create tools, configure them, make them available to Algorithms or Services and terminate them in an orderly fashion before deleting them.

The ToolSvc verifies if a tool type is available and creates the necessary instance after having verified if it doesn t already exist. If a tool instance exists the ToolSvc will not create a new identical one but pass to the algorithm the existing instance. Tools are created on a first request basis: the first Algorithm requesting a tool will prompt its creation. The relationship between an algorithm, the ToolSvc and Tools is shown in Figure 11.1.

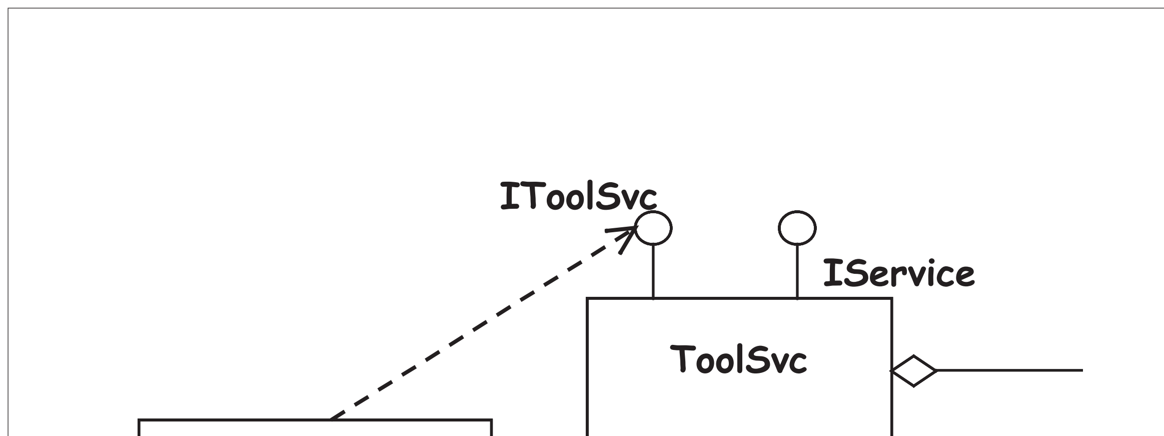

Figure 11.1 ToolSvc design diagram

Immediately after having created a tool, the ToolSvc will configure it by setting its properties and calling the tool initialize() method.

The ToolSvc will hold a tool until it is no longer used by any component or until the finalize() method of the tool service is called. Algorithms can inform the ToolSvc they are not going to use a

tool previously requested via the releaseTool method of the IToolSvc interface. Before deleting the tools the ToolSvc will cleanly terminate them by calling their finalize() method.

The ToolSvc is created by default by the ApplicationMgr and algorithms wishing to use the service can do so via the algorithm toolSvc() accessor method. Services and AlgTools need to retrieve it using the serviceLocator() method of their respective base classes.

11.3.1 Retrieval of tools via the IToolSvc interface

The IToolSvc interface is the ToolSvc specific interface providing methods to retrieve tools. The interface has two retrieve methods that differ in their parameters signature, as shown in Listing 11.5

Listing 11.5 The IToolSvc interface methods

```
1:  virtual StatusCode retrieve(const std::string& type,
                                const IID&,
                                IAlgTool*& tool,
                                const IInterface* parent=0,
                                bool createIf=true ) = 0;
2:  virtual StatusCode retrieve(const std::string& type,
                                const IID&,
                                const std::string& name,
                                IAlgTool*& tool,
                                const IInterface* parent=0,
                                bool createIf=true ) = 0;
```

The arguments of the method shown in Listing 11.5, line 1, are the tool type (i.e. the class), the tool additional interface ID and the IAlgTool interface of the returned tool. In addition there are two arguments with default values: one is the IInterface of the component requesting the tool, the other a boolean creation flag. If the component requesting a tool passes a pointer to itself as the third argument, it declares to the ToolSvc that it is asking for a private instance of the tool. By default a shared instance is provided. In general if the requested instance of a Tool does not exist the ToolSvc will create it. This behaviour can be changed by setting to false the last argument of the method.

The method shown in Listing 11.5, line 2 differs from the one shown in line 1 by an extra argument, a string specifying the tool dependent part of the full tool name. This enables a component to request two separately configurable instances of the same tool.

When retriving concrete tools, it is recommended to use the two templated functions provided in the `IToolSvc` interface file which are shown in Listing 11.6.

Listing 11.6 The IToolSvc template methods

```
 1: template <class T>
 2:   StatusCode retrieveTool( const std::string& type,
                                 T*& tool,
                                 const IInterface* parent=0,
                                 bool createIf=true ) {...}
 3: template <class T>
 4:   StatusCode retrieveTool( const std::string& type,
                                 const std::string& name,
                                 T*& tool,
                                 const IInterface* parent=0,
                                 bool createIf=true ) {...}
```

The two template methods correspond to the `IToolSvc` retrieve methods but have the tool returned as a template parameter. Using these methods the component retrieving a tool avoids explicit dynamic-casting to specific additional interfaces or to derived classes.

Listing 11.7 shows an example of retrieval of a shared and of a common tool.

Listing 11.7 Example of retrieval by an algortihm of a shared tool in line 4: and of a private tool in line 10:

```
 1: // Example of tool belonging to the ToolSvc and shared between
 2: // algorithms
 3: StatusCode sc;
 4: sc = toolsvc()->retrieveTool("AddFourMom", m_sum4p );
 5: if( sc.isFailure() ) {
 6:   log << MSG::FATAL << "    Unable to create AddFourMom tool" << endreq;
 7:   return sc;
 8: }
 9: // Example of private tool
10: sc = toolsvc()->retrieveTool("ImpactPar", m_ip, this );
11: if( sc.isFailure() ) {
12:   log << MSG::FATAL << "    Unable to create ImpactPar tool" << endreq;
13:   return sc;
14: }
```

11.4 GaudiTools

In general concrete tools are specific to applications or detectors code but there are some tools of common utility for which interfaces and base classes can be provided. The *Associators* described below and contained in the `GaudiTools` package are one of such tools.

11.4.1 Associators

When working with Monte Carlo data it is often necessary to compare the results of reconstruction or physics analysis with the original corresponding Monte Carlo quantities on an event-by-event basis as well as on a statistical level.

Various approaches are possible to implement navigation from reconstructed simulated data back to the Monte Carlo truth information. Each of the approaches has its advantages and could be more suited for a given type of event data or data-sets. In addition the reconstruction and physics analysis code should treat simulated data in an identical way to real data.

In order to shield the code from the details of the navigation procedure, and to provide a uniform interface to the user code, a set of Gaudi Tools, called *Associators*, has been introduced. The user can navigate between any two arbitrary classes in the Event Model using the same interface as long as a corresponding associator has been implemented. Since an Associator retrieves existing navigational information, its actual implementation depends on the Event Model and how the navigational information is stored. For some specific Associators, in addition, it can depend on some algorithmic choices: consider as an example a physics analysis particle and a possible originating Monte Carlo particle where the associating discriminant could be the fractional number of hits used in the reconstruction of the tracks. An advantage of this approach is that the implementation of the navigation can be modified without affecting the reconstruction and analysis algorithms because it would affect only the associators. In addition short-cuts or complete navigational information can be provided to the user in a transparent way. By limiting the use of such associators to dedicated *monitoring* algorithms where the comparison between raw/reconstructed data and MC truth is done, one could ensure that the reconstruction and analysis code treat simulated and real data in an identical way.

Associators must implement a common interface called `IAssociator`. An `Associator` base class providing at the same time common functionality and some facilities to help in the implementation of concrete Associators is provided. A prototype version of these classes is provided in the current release of Athena.

11.4.1.1 The `IAssociator` Interface

As already mentioned Associators must implement the `IAssociator` interface.

In order for Associators to be retrieved from the `ToolSvc` only via the `IAssociator` interface, the interface itself inherits from the `IAlgTool` interface. While the implementation of the `IAlgTool` interface is done in the `AlgTool` base class, the implementation of the `IAssociator` interface is the full responsibility of concrete associators.

The four methods of the `IAssociator` interface that a concrete Associator must implement are show in Listing 11.8

Listing 11.8 Methods of the IAssociator Interface that must be implemented by concrete associators

```
1: virtual StatusCode i_retrieveDirect( ContaineData Objectect* objFrom,
                               ContaineData Objectect*& objTo,
                               const CLID idFrom,
                               const CLID idTo ) = 0;
2: virtual StatusCode i_retrieveDirect( ContaineData Objectect* objFrom,
                                  std::vector<ContaineData Objectect*>&
   vObjTo,                               const CLID idFrom,
                               const CLID idTo ) = 0;
3: virtual StatusCode i_retrieveInverse( ContaineData Objectect* objFrom,
                               ContaineData Objectect*& objTo,
                               const CLID idFrom,
                               const CLID idTo) = 0;
4: virtual StatusCode i_retrieveInverse( ContaineData Objectect* objFrom,
                               std::vector<ContaineData Objectect*>&
   vObjTo,                               const CLID idFrom,
                               const CLID idTo) = 0;
```

Two `i_retrieveDirect` methods must be implemented for retrieving associated classes following the same direction as the links in the data: for example from reconstructed particles to Monte Carlo particles. The first parameter is a pointer to the object for which the associated Monte Carlo quantity(ies) is requested. The second parameter, the discriminating signature between the two methods, is one or a vector of pointers to the associated Monte Carlo objects of the type requested. Some reconstructed quantities will have only one possible Monte Carlo associated object of a certain type, some will have many, others will have many out of which a best associated object can be extracted. If one of the two methods is not valid for a concrete associator, such method must return a failure. The third and fourth parameters are the class IDs of the objects for which the association is requested. This allows to verify at run time if the objects types are those the concrete associator has been implemented for.

The two `i_retrieveInverse` methods are complementary and are for retrieving the association between the same two classes but in the opposite direction to that of the links in the data: for example from Monte Carlo particles to reconstructed particles. The different name is intended to alert the user that navigation in this direction may be a costly operation

Four corresponding template methods are implemented in `IAssociator` to facilitate the use of Associators by Algorithms (see Listing 11.9). Using these methods the component retrieving a tool

avoids some explicit dynamic-casting as well as the setting of class IDs. An example of how to use such methods is described in section 11.4.1.3.

Listing 11.9 Template methods of the IAssociator interface

```
1: template <class T1, class T2>
     StatusCode retrieveDirect( T1* from, T2*& to ) {...}
2: template <class T1>
     StatusCode retrieveDirect( T1* from,
                                std::vector<ContaineData
   Objectect*>& objVTo,                          const CLID idTo )
   {...}
3: template <class T1, class T2>
     StatusCode retrieveInverse( T1* from, T2*& to ) {...}
4: template <class T1>
     StatusCode retrieveInverse( T1* from,
                                std::vector<ContaineData
   Objectect*>& objVTo,                          const CLID idTo )
   {...}
```

11.4.1.2 The Associator base class

An associator is a type of `AlgTool`, so the `Associator` base class inherits from the `AlgTool` base class. Thus, Associators can be created and managed as `AlgTools` by the `ToolSvc`. Since all the methods of the `AlgTool` base class (as described in section 11.2.2.1) are available in the `Associator` base class, only the additional functionality is described here.

Access to Event Data Service - An `eventSvc()` method is provided to access the Event Data Service since most concrete associators will need to access data, in particular if accessing navigational short-cuts.

Associator Properties - Two properties are declared in the constructor and can be set in the jobOptions: `FollowLinks` and `DataLocation`. They are respectively a `bool` with initial value `true` and a `std::string` with initial value set to . The first is foreseen to be used by an associator when it is possible to either follow links between classes or retrieve navigational short cuts from the data. A user can choose to set either behaviour at run time. The second property contains the location in the data where the stored navigational information is located. Currently it must be set via the jobOptions when necessary, as shown in Listing 11.10 for a particular implementation provided in the Associator example. Two corresponding methods are provided for using the information from these properties: `followLinks()` and `whichTable()`.

Inverse Association - Retrieving information in the direction opposite to that of the links in the data is in general a time consuming operation, that implies checking all the direct associations to access the inverse relation for a specified object. For this reason Associators should keep a local copy of the inverse associations after receiving the first request for an event. A few methods are provided to facilitate the work of Associators in this case. The methods `inverseExist`() and `setInverseFlag(bool)` help in keeping track of the status of the locally kept inverse information.The method `buildInverse`() has to be overridden by concrete associators since they choose in which form to keep the information and should be called by the associator when receiving the first request during the processing of an event.

Locally kept information - When a new event is processed, the associator needs to reset its status to the same conditions as those after having been created . In order to be notified of such an incident happening the Associator base class implements the IListener interface and, in the constructor, registers itself with the Incident Service (see section 10.9 for details of the Incident Service). The associator' s flushCache() method is called in the implementation of the IListener interface in the Associator base class. This method must be overridden by concrete associators wanting to do a meaningful reset of their initial status.

11.4.1.3 A concrete example

In this section we look at an example implementation of a specific associator. The code is taken from the LHCb Associator example, but the points illustrated should be clear even without a knowledge of the LHCb data model.

The AxPart2MCParticleAsct provides association between physics analysis particles (AxPartCandidate) and the corresponding Monte Carlo particles (MCParticle). The direct navigational information is stored in the persistent data as short-cuts, and is retrieved in the form of a SmartRefTable in the Transient Event Store. This choice is specific to AxPart2MCParticleAsct, any associator can use internally a different navigational mechanism. The location in the Event Store where the navigational information can be found is set in the job options via the DataLocation property, as shown in Listing 11.10.

Listing 11.10 Example of setting properties for an associator via jobOptions

```
ToolSvc.AxPart2MCParticleAsct.DataLocation =
"/Event/Anal/AxPart2MCParticle";
```

In the current LHCb data model only a single MCParticle can be associated to one AxPartCandidate and vice-versa only one or no AxPartCandidate can be associated to one MCParticle. For this reason only the i_retrieveDirect and i_retrieveInverse methods providing one-to-one association are meaningful. Both methods verify that the objects passed are of the *correct* type before attempting to retrieve the information, as shown in Listing 11.11. When no association is found, a StatusCode::FAILURE is returned.

Listing 11.11 Checking if objects to be associated are of the correct type

```
1: if ( idFrom != AxPartCandidate::classID() ){
2:   objTo = 0;
3:   return StatusCode::FAILURE;
4: }
5: if ( idTo != MCParticle::classID() ) {
6:   objTo = 0;
7:   return StatusCode::FAILURE;
8: }
```

The i_retrieveInverse method providing the one-to-many association returns a failure, while a fake implementation of the one-to-many i_retrieveDirect method is implemented in the example, to show how an Algorithm can use such a method. In the AxPart2MCParticleAsct

example the inverse table is kept locally and both the `buildInverse`() and `flushCache`() methods are overridden. In the example the choice has been made to implement an additional method `buildDirect`() to retrieve the direct navigational information on a first request per event basis.

Listing 11.12 shows how a *monitoring* Algorithm can get an associator from the `ToolSvc` and use it to retrieve associated objects through the template interfaces.

Listing 11.12 Extracted code from the AsctExampleAlgorithm

```
 1: #include "GaudiTools/IAssociator.h"

 2: // Example of retrieving an associator
 3: IAssociator
 4: StatusCode sc = toolsvc()->retrieveTool("AxPart2MCParticleAsct",
                                            m_pAsct);
 5: if( sc.isFailure() ) {
 6:   log << MSG::FATAL << "Unable to create Associator tool" << endreq;
 7:   return sc;
 8: }
 9: // Example of retrieving inverse one-to-one information from an
10: // associator
11: SmartDataPtr<MCParticleVector> vmcparts (evt,"/MC/MCParticles");
12: for( MCParticleVector::iterator itm = vmcparts->begin();
         vmcparts->end() != itm; itm++) {
13:      AxPartCandidate* mptry = 0;
14:      StatusCode sc = m_pAsct->retrieveInverse( *itm, mptry );
15:      if( sc.isSuccess() ) {...}
16:      else {...}
17: }
18: // Example of retrieving direct one-to-many information from an
19: // associator
20: SmartDataPtr<AxPartCandidateVector> candidates(evt,
                                     "/Anal/AxPartCandidates");
21: std::vector<ContaineData Objectect*> pptry;
22: AxPartCandidate* itP = *(candidates->begin());
23: StatusCode sa =
      m_pAsct->retrieveDirect(itP, pptry, MCParticle::classID());
24: if( sa.isFailure() ) {...}
25: else {
26:   for (std::vector<ContaineData Objectect*>::iterator it =
    pptry.begin();        pptry.end() != it; it++ ) {
27:        MCParticle* imc = dynamic_cast<MCParticle*>( *it );
28:   }
29: }
```

Chapter 12
Converters

12.1 Overview

Consider a small piece of detector; a silicon wafer for example. This object will appear in many contexts: it may be drawn in an event display, it may be traversed by particles in a Geant4 simulation, its position and orientation may be stored in a database, the layout of its strips may be queried in an analysis program, etc. All of these uses or views of the silicon wafer will require code.

One of the key issues in the design of the framework was how to encompass the need for these different views within Athena. In this chapter we outline the design adopted for the framework and look at how the conversion process works. This is followed by sections which deal with the technicalities of writing converters for reading from and writing to ROOT files.

12.2 Persistency converters

Athena gives the possibility to read event data from, and to write data back to, ROOT files. The use of ODBC compliant databases is also possible, though this is not yet part of the Athena release. Other persistency technologies have been implemented for LHCb, in particular the reading of data from LHCb DSTs based on ZEBRA.

Figure 12.1 is a schematic illustrating how converters fit into the transient-persistent translation of event data. We will not discuss in detail how the transient data store (e.g. the event data service) or the

persistency service work, but simply look at the flow of data in order to understand how converters are used. An introduction to the persistency mechanism of Gaudi can be found in reference [13].

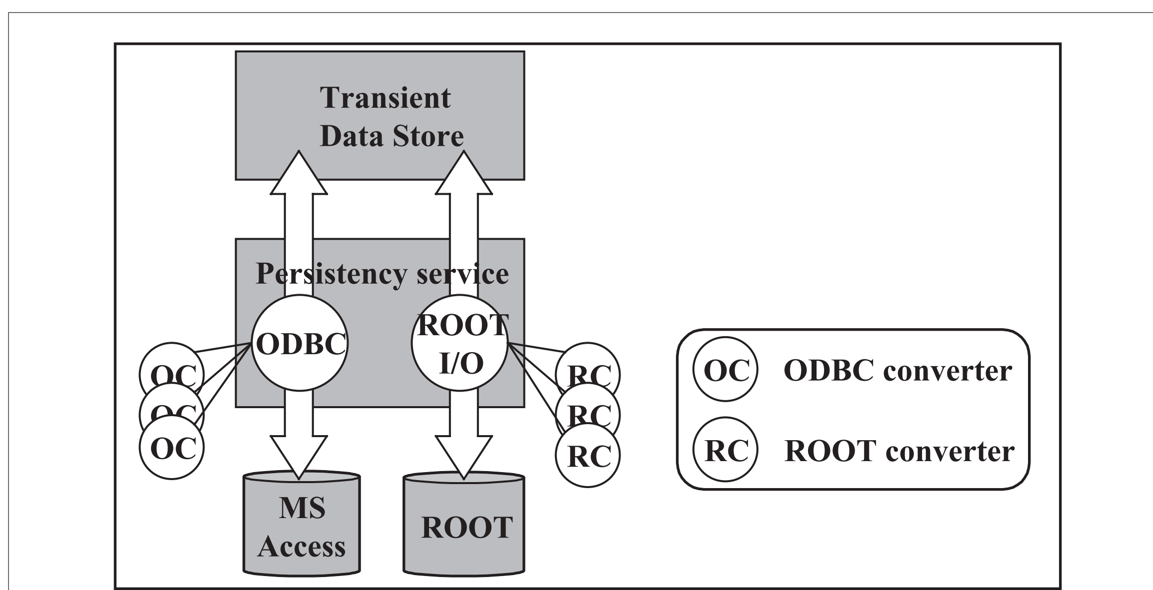

Figure 12.1 Persistency conversion services in Gaudi

One of the issues considered when designing the Gaudi framework was the capability for users to create their own data types and save objects of those types along with references to already existing objects . A related issue was the possibility of having links between objects which reside in different stores (i.e. files and databases) and even between objects in different types of store.

Figure 12.1 shows that data may be read from an ODBC database and/or ROOT files into the transient event data store and that data may be written out again to the same media. It is the job of the persistency service to orchestrate this transfer of data between memory and disk.

The figure shows two slave services: the ODBC conversion service and the ROOT I/O service. These services are responsible for managing the conversion of objects between their transient and persistent representations. Each one has a number of converter objects which are actually responsible for the conversion itself. As illustrated by the figure a particular converter object converts between the transient representation and one other form, here either MS Access or ROOT.

12.3 Collaborators in the conversion process

In general the conversion process occurs between the transient representation of an object and some other representation. In this chapter we will be using persistent forms, but it should be borne in mind that this could be any other transient form such as those required for visualisation or those which serve as input into other packages (e.g. Geant4).

Figure 12.1 shows the interfaces (classes whose name begins with "I") which must be implemented in order for the conversion process to function.

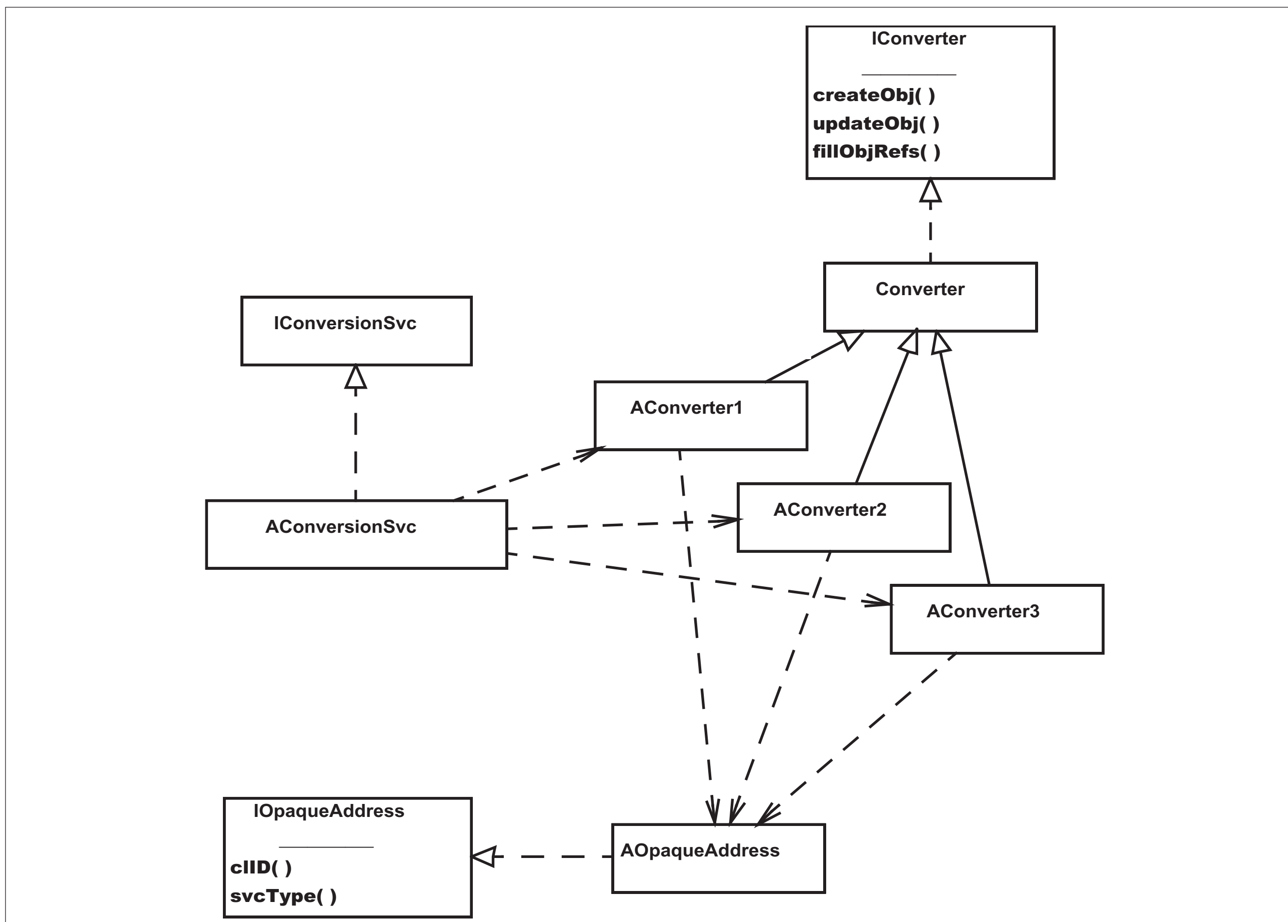

Figure 12.1 The classes (and interfaces) collaborating in the conversion process.

The conversion process is essentially a collaboration between the following types:

¥ IConversionSvc

¥ IConverter

¥ IOpaqueAddress

For each persistent technology, or non-transient representation, a specific conversion service is required. This is illustrated in the figure by the class `AConversionSvc` which implements the `IConversionSvc` interface.

A given conversion service will have at its disposal a set of converters. These converters are both type and technology specific. In other words a converter knows how to convert a single transient type (e.g. `MuonHit`) into a single persistent type (e.g. `RootMuonHit`) and vice versa. Specific converters implement the `IConverter` interface, possibly by extending an existing converter base class.

A third collaborator in this process are the opaque address objects. A concrete opaque address class must implement the `IOpaqueAddress` interface. This interface allows the address to be passed around between the transient data service, the persistency service, and the conversion services without any of them being able to actually decode the address. Opaque address objects are also technology specific. The internals of an `OdbcAddress` object are different from those of a `RootAddress` object.

Only the converters themselves know how to decode an opaque address. In other words only converters are permitted to invoke those methods of an opaque address object which do not form a part of the `IOpaqueAddress` interface.

Converter objects must be registered with the conversion service in order to be usable. For the standard converters this will be done automatically . For user defined converters (for user defined types) this registration must be done at initialisation time (see Section 7.10).

12.4 The conversion process

As an example (see Figure 12.1) we consider a request from the event data service to the persistency service for an object to be loaded from a data file.

As we saw previously, the persistency service has one conversion service slave for each persistent technology in use. The persistency service receives the request in the form of an opaque address object. The `svcType()` method of the `IOpaqueAddress` interface is invoked to decide which conversion service the request should be passed onto. This returns a technology identifier which allows the persistency service to choose a conversion service.

The request to load an object (or objects) is then passed onto a specific conversion service. This service then invokes another method of the `IOpaqueAddress` interface, `clID()`, in order to decide which converter will actually perform the conversion. The opaque address is then passed onto the concrete converter who knows how to decode it and create the appropriate transient object.

The converter is specific to a specific type, thus it may immediately create an object of that type with the new operator. The converter must now unpack the opaque address, i.e. make use of accessor methods specific to the address type in order to get the necessary information from the persistent store.

For example, a ZEBRA converter might get the name of a bank from the address and use that to locate the required information in the ZEBRA common block. On the other hand a ROOT converter may extract a file name, the names of a ROOT `TTree` and an index from the address and use these to load an object from a ROOT file. The converter would then use the accessor methods of this persistent object in order to extract the information necessary to build the transient object.

We can see that the detailed steps performed within a converter depend very much on the nature of the non-transient data and (to a lesser extent) on the type of the object being built.

AConversionSvc
AOpaqueAddress
AConverter
DB/File
createObj(OA)
clID( )
Id
createObj(OA)
"unpack"
pointers into
persistent file/DB
new
DataObject
"access(es)"
data to build
transient object
setX( )
setY( )
return reference to DataObject
DataObject

Figure 12.1 A trace of the creation of a new transient object.

If all transient objects were independent, i.e. if there were no references between objects then the job would be finished. However in general objects in the transient store do contain references to other objects.

These references can be of two kinds:

i. Macroscopic references appear as separate leaves in the data store. They have to be registered with a separate opaque address structure in the data directory of the object being converted. This must be done after the object was registered in the data store in the method **`fillObjRefs()`.**

ii. Internal references must be handled differently. There are two possibilities for resolving internal references:

1. Load on demand. If the object the reference points to should only be loaded when accessed, the pointer must no longer be a raw C++ pointer, but rather a smart pointer object containing itself the information for later resolution of the reference. This is the preferred solution for references to objects within the same data store (e.g. references from Monte-Carlo tracks to Monte-Carlo vertices) and is generated by the Object Description Tools when a relation tag is found in the XML class description (see Section 7.9).

2. Filling of raw C++ pointers. This is only necessary if the object points to an object in another store, e.g. the detector data store, and should be avoided in classes foreseen to be made persistent. To resolve the reference a converter has to retrieve the other object and set the raw pointer. These references should be set in the `fillObjRefs()` method. This of course is more complicated, because it must be ensured that both objects are present at the time the reference is accessed (i.e. when the pointer is actually used).

12.5 Converter implementation - general considerations

After covering the ground work in the preceding sections, let us look exactly what needs to be implemented in a specific converter class. The starting point is the Converter base class from which a user converter should be derived.

Listing 12.1 An example converter class

```
// Converter for class UDO.
extern const CLID& CLID_UDO;
extern unsigned char OBJY_StorageType;

static CnvFactory<UDOCnv> s_factory;
const ICnvFactory& UDOCnvFactory = s_factory;

class UDOCnv : public Converter {
public:
  UDOCnv(ISvcLocator* svcLoc) :
      Converter(Objectivity_StorageType, CLID_UDO, svcLoc) { }

  createRep(DataObject* pO, IOpaqueAddress*& a);  // transient->persistent
  createObj(IOpaqueAddress* pa, DataObject*& pO); // persistent->transient

  fillObjRefs( ... ); // transient->persistent
  fillRepRefs( ... ); // persistent->transient
}
```

The converter shown in Listing 12.1 is responsible for the conversion of UDO type objects into objects that may be stored into an Objectivity database and vice-versa. The UDOCnv constructor calls the Converter base class constructor with arguments which contain this information. These are the values CLID_UDO, defined in the UDO class, and Objectivity_StorageType which is also defined elsewhere. The first two extern statements simply state that these two identifiers are defined elsewhere.

All of the book-keeping can now be done by the Converter base class. It only remains to fill in the guts of the converter. If objects of type UDO have no links to other objects, then it suffices to implement the methods createRep() for conversion from the transient form (to Objectivity in this case) and createObj() for the conversion to the transient form.

If the object contains links to other objects then it is also necessary to implement the methods fillRepRefs() and fillObjRefs().

12.6 Storing Data using the ROOT I/O Engine

One possibility for storing data is to use the ROOT I/O engine to write ROOT files. Although ROOT by itself is not an object oriented database, with modest effort a structure can be built on top to allow the

Converters to emulate this behaviour. In particular, the issue of object linking had to be solved in order to resolve pointers in the transient world.

The concept of ROOT supporting paged tuples called trees and branches is adequate for storing bulk event data. Trees split into one or several branches containing individual leaves with data.

The data structure within the Gaudi data store is also tree like. In the transient world Gaudi objects are sub-class instances of the DataObject . The DataObject offers some basic functionality like the implicit data directory which allows e.g. to browse a data store. This tree structure will be mapped to a flat structure in the ROOT file resulting in a separate tree representing each leaf of the data store. Each data tree contains a single branch containing objects of the same type. The Gaudi tree is split up into individual ROOT trees in order to give easy access to individual items represented in the transient model without the need of loading complete events from the root file i.e. to allow for selective data retrieval. The feature of ROOT supporting selective data reading using split trees did not seem too attractive since, generally, complete nodes in the transient store should be made available in one go.

However, ROOT expects ROOT objects, they must inherit from `TObject`. Therefore the objects from the transient store have to be converted to objects understandable by ROOT.

The following sections are an introduction to the machinery provided by the Gaudi framework to achieve the migration of transient objects to persistent objects. The ROOT specific aspects are not discussed here; the ROOT I/O engine is documetned on the ROOT web site *http://root.cern.ch*). Note that Gaudi only uses the I/O engine, not all ROOT classes are available. Within Gaudi the ROOT I/O engine is implemented in the `GaudiRootDb` package.

12.7 The Conversion from Transient Objects to ROOT Objects

As for any conversion of data from one representation to another within the Gaudi framework, conversion to/from ROOT objects is based on Converters. The support of a generic Converter accesses pre-defined entry points in each object. The transient object converts itself to an abstract byte stream. However, for specialized objects specific converters must be built.

Whenever objects must change their representation within Gaudi, data converters are involved. For the ROOT case, the converters must have some knowledge of ROOT internals and of the service finally used to migrate ROOT objects (->`TObject`) to a file. They must be able to translate the functionality of the `DataObject` component to/from the ROOT storage. Within ROOT itself the object is stored as a Binary Large Object (BLOB).

The generic data conversion mechanism relies on two functionalities, which must be present:

¥ When writing or reading objects, the object s data must be "serializable". The corresponding persistent type is of a generic type, the data are stored as a machine independent byte stream. This method is implemented automatically if the class is described using the Gaudi Object Description tools (described in Section 7.7 on page 7). When reading objects, an empty object must be created before any de-serialization can take place. The constructor must be called. This functionality does not imply any knowledge of the conversion mechanism itself and

hence can be encapsulated into an object factory simply returning a DataObject. These data object factories are distinguished within Gaudi through the persistent data type information, the class ID. For this reason the class ID of objects, which are written must only depend on the object type, i.e. every class needs it's own class ID.The instantiation of the appropriate factory is done by a macro. Please see the RootIO example for details how to instantiate the factory.

12.8 Storing Data using other I/O Engines

Once objects are stored as BLOBs, it is possible to adopt any storage technology supporting this datatype. This is the case not only for ROOT, but also for

¥ Objectivity/DB
¥ most relational databases, which support an ODBC interface like
    ¥ Microsoft Access,
    ¥ Microsoft SQL Server,
    ¥ MySQL,
    ¥ ORACLE and others.

Note that although storing objects using these technologies is possible, there is currently no implementation available in the Gaudi release. If you desperately want to use Objectivity or one of the ODBC databases, please contact Markus Frank (*Markus.Frank@cern.ch*).

Chapter 13
Visualization

13.1 Overview

This Chapter is in preparation.

Chapter 14
Framework packages, interfaces and libraries

14.1 Overview

It is clearly important to decompose large software systems into hierarchies of smaller and more manageable entities. This decomposition can have important consequences for implementation related issues, such as compile-time and link dependencies, configuration management, etc. A *package* is the grouping of related components into a cohesive physical entity. A package is also the minimal unit of software release.

In this chapter we describe the Athena package structure, and how these packages are implemented in libraries. We also discuss abstract inerfaces, which are one of the main design features of Athena

14.2 Athena Package Structure

14.2.1 Packaging Guidelines

Packaging is an important architectural issue for the Gaudi framework, but also for the experiment specific software packages based on Gaudi. Typically, experiment packages consist of:

¥ Specific event model
¥ Specific detector description
¥ Sets of algorithms (digitisation, reconstruction, etc.)

The packaging should be such as to minimise the dependencies between packages, and must absolutely avoid cyclic dependencies. The granularity should not be too small or too big. Care should be taken to identify the external interfaces of packages: if the same interfaces are shared by many packages, they should be promoted to a more basic package that the others would then depend on. It is a good idea to discuss your packaging with the librarian and/or architect.

14.3 Interfaces in Gaudi

One of the main design choices at the architecture level in Gaudi was to favour abstract interfaces when building collaborations of various classes. This is the way we best decouple the client of a class from its real implementation.

An abstract interface in C++ is a class where all the methods are pure virtual. We have defined some practical guidelines for defining interfaces. An example is shown in Listing 14.1:

Listing 14.1 Example of an abstract interface (IService)

```
 1: // $Header:  $
 2: #ifndef GAUDIKERNEL_ISERVICE_H
 3: #define GAUDIKERNEL_ISERVICE_H
 4:
 5: // Include files
 6: #include "GaudiKernel/IInterface.h"
 7: #include <string>
 8:
 9: // Declaration of the interface ID. (id, major, minor)
10: static const InterfaceID IID_IService(2, 1, 0);
11:
12: /** @class IService IService.h GaudiKernel/IService.h
13:
14:     General service interface definition
15:
16:     @author Pere Mato
17: */
18: class IService : virtual public IInterface  {
19: public:
20:   /// Retrieve name of the service
21:   virtual const std::string&  name() const = 0;
22:   /// Retrieve ID of the Service. Not really used.
23:   virtual const IID&          type() const = 0;
24:   /// Initilize Service
25:   virtual StatusCode initialize() = 0;
26:   /// Finalize Service
27:   virtual StatusCode finalize() = 0;
28:   /// Retrieve interface ID
29:   static const InterfaceID& interfaceID() {  return IID_IService; }
30: };
31:
32: #endif  // GAUDIKERNEL_ISERVICE_H
```

From this example we can make the following observations:

¥ **Interface Naming**. The name of the class has to start with capital I to denote that it is an interface.

¥ Derived from IInterface. We follow the convention that all interfaces should be derived from a basic interface `IInterface`. This interface defined 3 methods: `addRef()`, `release()` and `queryInterface()`. This methods allow the framework to manage the reference counting of the framework components and the possibility to obtain a different interface of a component using any interface (see Section 14.3.2).

¥ **Pure Abstract Methods**. All the methods should be pure abstract (`virtual ReturnType method(...) = 0;`) With the exception of the static method `interfaceID()` (see later) and some inline templated methods to facilitate the use of the interface by the end-user.

¥ **Interface ID**. Each interface should have a unique identification (see Section 14.3.1) used by the query interface mechanism.

14.3.1 Interface ID

We needed to introduce an interface ID for identifying interfaces for the queryInterface functionality. The interface ID is made of a numerical identifier (generated from the interface name by a hash function) and major and minor version numbers. The version number is used to decide if the interface the service provider is returning is compatible with the interface the client is expecting. The rules for deciding if the interface request is compatible are:

¥ The interface identifier is the same

¥ The major version is the same

¥ The minor version of the client is less than or equal to the one of the service provider. This allows the service provider to add functionality (incrementing minor version number) keeping old clients still compatible.

The interface ID is defined in the same header file as the rest of the interface. Care should be taken of globally allocating the interface identifier (by giving a unique name to the constructor), and of modifying the version whenever a change of the interface is required, according to the rules. Of course changes to interfaces should be minimized.

```
static const InterfaceID IID_Ixxx("Ixxx" /*id*/, 1 /*major*/, 0 /*minor*/);

class Ixxx : public IInterface {
  . . .
  static const InterfaceID& interfaceID() { return IID_Ixxx; }
};
```

The static method `Ixxx::interfaceID()` is useful for the implementation of templated methods and classes using an interface as template parameter. The construct `T::interfaceID()` returns the interface ID of interface T.

14.3.2 Query Interface

The method `queryInterface()` is used to request a reference to an interface implemented by a component within the Gaudi framework. This method is implemented by each component class of the framework and allows us to navigate from one interface of a component to another, as shown for example in Listing 14.2, where we navigate from the `IMessageSvc` interface of the message service to its `IProperty` interface, in order to discover the value of its "`OutputLevel`" property.

Listing 14.2 Example usage of `queryInterface` to navigate between interfaces

```
 1:  IMessageSvc* msgSvc();
 2:  ...
 3:  IProperty* msgProp;
 4:  msgSvc()->queryInterface( IID_IProperty, (void**)&msgProp );
 5:  std::string dfltLevel;
 6:  StatusCode scl = msgProp->getProperty( "OutputLevel", dfltLevel );
```

The implementation of `queryInterface()` is usually not very visible since it is done in the base class from which you inherit. A typical implementation is shown in Listing 14.3:

Listing 14.3 Example implementation of queryInterface()

```
 1:  StatusCode DataSvc::queryInterface(const InterfaceID& riid,
 2:                                     void** ppvInterface) {
 3:    if ( IID_IDataProviderSvc.versionMatch(riid) )  {
 4:      *ppvInterface = (IDataProviderSvc*)this;
 5:    }
 6:    else if ( IID_IDataManagerSvc.versionMatch(riid) )   {
 7:      *ppvInterface = (IDataManagerSvc*)this;
 8:    }
 9:    else  {
10:      return Service::queryInterface(riid, ppvInterface);
11:    }
12:    addRef();
13:    return SUCCESS;
14:  }
```

The implementation returns the corresponding interface pointer if there is a match between the received `InterfaceID` and the implemented one. The method `versionMatch()` takes into account the rules mentioned in Section 14.3.1.

If the requested interface is not recognized at this level (line 9), the call can be forwarded to the inherited base class or possible sub-components of this component.

14.4 Libraries in Athena

Three different sorts of library can be identified that are relevant to the framework. These are *component* libraries, *linker* (or *installed*) libraries and *dual-use* libraries. These libraries are used for different purposes and are built in different ways.

14.4.1 Component libraries

Component libraries are shared libraries that contain standard framework components which implement abstract interfaces. Such components are *Algorithms*, *Auditors*, *Services*, *Tools* or *Converters*. These libraries do not export their symbols apart from one which is used by the framework to discover what components are contained by the library. Thus component libraries should not be linked against; they are used purely at run-time, being loaded dynamically upon request, the configuration being specified by the job options file. Changes in the implementation of a component library do not require the application to be relinked.

Component libraries contain factories for their components, and it is important that the factory entries are declared and loaded correctly. The following sections describe how this is done.

When a component library is loaded, the framework attempts to locate a single entrypoint, called `getFactoryEntries()`. This is expected to declare and load the component factories from the library. Several macros are available to simplify the declaration and loading of the components via this function.

Consider a simple package `MyComponents`, that declares and defines the `MyAlgorithm` class, being a subclass of `Algorithm`, and the `MyService` class, being a subclass of `Service`. Thus the package will contain the header and implementation files for these classes (`MyAlgorithm.h`, `MyAlgorithm.cpp`, `MyService.h` and `MyService.cpp`) in addition to whatever other files are necessary for the correct functioning of these components.

In order to satisfy the requirements of a component library, two additional files must also be present in the package. One is used to declare the components, the other to load them. Because of the technical limitations inherent in the use of shared libraries, it is important that these two files remain separate, and that no attempt is made to combine their contents into a single file.

The names of these files and their contents are described in the following sections.

14.4.1.1 Declaring Components

Components within the component library are declared in a file MyComponents_entries.cpp. By convention, the name of this file is the package name concatenated with _entries. The contents of this file are shown below:

Listing 14.4 The MyComponents_entries.cpp file

```
#include "GaudiKernel/DeclareFactoryEntries.h"

DECLARE_FACTORY_ENTRIES( MyComponents ) {        [ 1]
    DECLARE_ALGORITHM( MyAlgorithm );            [ 2]
    DECLARE_SERVICE   ( MyService );
}
```

Notes:

1. The argument to the DECLARE_FACTORY_ENTRIES statement is the name of the component library.
2. Each component within the library should be declared using one of the DECLARE_XXX statements discussed in detail in the next Section.

14.4.1.2 Component declaration statements

The complete set of statements that are available for declaring components is given below. They include those that support C++ classes in different namespaces, as well as for DataObjects or ContaineData Objectects using the generic converters.

Listing 14.5 The available component declaration statements

```
DECLARE_ALGORITHM(X)
DECLARE_AUDITOR(X)
DECLARE_CONVERTER(X)
DECLARE_GENERIC_CONVERTER(X)                     [ 1]
DECLARE_OBJECT(X)
DECLARE_SERVICE(X)

DECLARE_NAMESPACE_ALGORITHM(N,X)                 [ 2]
DECLARE_NAMESPACE_AUDITOR(N,X)
DECLARE_NAMESPACE_CONVERTER(N,X)
DECLARE_NAMESPACE_GENERIC_CONVERTER(N,X)
DECLARE_NAMESPACE_OBJECT(N,X)
DECLARE_NAMESPACE_SERVICE(N,X)
```

Listing 14.5 The available component declaration statements

Notes:

1. Declarations of the form `DECLARE_GENERIC_CONVERTER(X)` are used to declare the generic converters for `DataObject` and `ContaineData Objectect` classes. For `DataObject` classes, the argument should be the class name itself (*e.g.* `EventHeader`), whereas for `ContaineData Objectect` classes, the argument should be the class name concatenated with either `List` or `Vector` (*e.g.* `CellVector`) depending on whether the objects are associated with an `ObjectList` or `ObjectVector`.
2. Declarations of this form are used to declare components from explicit C++ namespaces. The first argument is the namespace (*e.g.* `Atlfast`), the second is the class name (*e.g.* `CellMaker`).

14.4.1.3 Loading Components

Components within the component library are loaded in a file `MyComponents_load.cpp`. By convention, the name of this file is the package name concatenated with `_load`. The contents of this file are shown below:

Listing 14.6 The `MyComponents_load.cpp` file

```
#include "GaudiKernel/LoadFactoryEntries.h"

LOAD_FACTORY_ENTRIES( MyComponents )          [ 1]
```

Notes:

1. The argument of `LOAD_FACTORY_ENTRIES` is the name of the component library.

14.4.1.4 CMT requirements file fragment to create a component library

The fragment of the package requirements file that creates a component library is shown below:.

Listing 14.7 Creating a component library

```
library MyPackage <list of files>      [ 1][ 2][ 3]

apply_pattern component_library        [ 3]
```

Listing 14.7 Creating a component library

Notes:

1. The normal convention is for the library name to be the same as the package name.
2. The `<list of files>` can either be an explicit list of files as shown, or wildcards may be used:

```
library MyPackage *.cxx
```

3. Source files not located in the package `src/` directory can be specified using the `-s=<directory>` option, which specifies a directory path relative to the `src/` directory:

```
library MyPackage *.cxx -s=components *.cxx
```

4. The `component_library` pattern operates on the library specified in the previous `library` statement.

14.4.1.5 Specifying component libraries at run-time

The fragment of the job options file that specifies the component library at run-time is shown below.

Listing 14.8 Specifying a component library at run-time

```
ApplicationMgr.DLLs  += {  "MyComponents" } ;   [ 1]
```

Notes:

1. This is a list property, allowing multiple such libraries to be specified in a single line.
2. It is important to use the += syntax to append the new component library or libraries to any that might already have been configured.

The convention in Gaudi is that component libraries have the same name as the package they belong to (prefixed by "`lib`" on Linux). When trying to load a component library, the framework will look for it in various places following this sequence:

Look for an environment variable with the name of the package, suffixed by "`Shr`" (e.g. `${ MyComponentsShr}`). If it exists, it should translate to the full name of the library, without the file type suffix (e.g. `${ MyComponentsShr} ="$MYSOFT/MyComponents/v1/i386_linux22/libMyComponents"`).

Try to locate the file `libMyComponents.so` using the `LD_LIBRARY_PATH` (on Linux), or `MyComponents.dll` using the `PATH` (on Windows).

14.4.2 Linker (or installed) libraries

These are libraries containing implementation classes. For example, libraries containing code of a number of base classes or specific classes without abstract interfaces, etc. These libraries, contrary to the component libraries, export all the symbols and are needed during the linking phase in the application building. These libraries can be linked to the application "*statically*" or "*dynamically*", requiring a different file format. In the first case the code is added physically to the executable file. In this case, changes in these libraries require the application to be re-linked, even if these changes do not affect the interfaces. In the second case, the linker only adds into the executable minimal information required for loading the library and resolving the symbols at run time. Locating and loading the proper shareable library at run time is done exclusively using the `LD_LIBRARY_PATH` for Linux and `PATH` for Windows. The convention in Gaudi is that linker libraries have the same name as the package, suffixed by "`Lib`" (and prefixed by "`lib`" on Linux, e.g. `libMyPackageLib.so`).

14.4.2.1 CMT requirements file fragment to create a linker or installed library

The fragment of the package requirements file that creates a linker (or installed) library is shown below:..

Listing 14.9 Creating a linker/installled library

```
library MyPackage <list of files>     [ 1][ 2][ 3]

apply_pattern installed_library       [ 3]
```

Notes:

1. The normal convention is for the library name to be the same as the package name.
2. The `<list of files>` can either be an explicit list of files as shown, or wildcards may be used:

```
library MyPackage *.cxx
```

3. Source files not located in the package `src/` directory can be specified using the `-s=<directory>` option, which specifies a directory path relative to the `src/` directory:

```
library MyPackage *.cxx -s=components *.cxx
```

4. The `installed_library` pattern operates on the library specified in the previous `library` statement.

14.4.3 Dual use libraries

Because component libraries are not designed to be linked against, it is important to separate the functionalities of these libraries from linker libraries. For example, consider the case of a DataProvider service that provides DataObjects for clients. It is important that the declarations and definitions of the DataObjects be handled by a different shared library than that handling the service itself. This implies the presence of two different packages - one for the component library, the other for the DataObjects. Clients should only depend on the second of these packages. Obviously the package handling the component library will in general also depend on the second package.

It *is* possible to have dual purpose libraries - ones which are simultaneously component and linker libraries. In general such libraries will contain DataObjects and ContaineData Objectects, together with their converters and associated factories. It is recommended that such dual purpose libraries be separated from single purpose component or linker libraries. Consider the case where several Algorithms share the use of several DataObjects (*e.g.* where one Algorithm creates them and registers them with the transient event store, and another Algorithm locates them), and also share the use of some helper classes in order to decode and manipulate the contents of the DataObjects. It is recommended that three different packages be used for this - one pure component package for the Algorithms, one dual-purpose for the DataObjects, and one pure linker package for the helper classes.

14.4.3.1 CMT requirements file fragment to create a dual use library

The fragment of the package requirements file that creates a dual use library is shown below:.

Listing 14.10 Creating a dual use library

```
apply_pattern dual_use_library files="MyFile1.cxx MyFile2.cxx"
```

Notes:

1. The normal convention is for the library name to be the same as the package name.
2. The list of files can either be an explicit list of files as shown, or wildcards may be used:

   ```
   library MyPackage *.cxx
   ```

3. Two component declaration files must exist in the `src/components` directory. They are:

   ```
   Pkg_entries.cxx
   Pkg_load.cxx
   ```

   These have the same content as described in Section 14.4.1 for component libraries.
4. Factory code as described in Section 3.3.1 should be removed from the Algorithm, Service, Tool or Converter header file.

14.4.4 Linking FORTRAN code

Any library containing FORTRAN code (more specifically, code that references COMMON blocks) must be linked statically. This is because COMMON blocks are, by definition, static entities. When mixing C++ code with FORTRAN, it is recommended to build separate libraries for the C++ and FORTRAN, and to write the code in such a way that communication between the C++ and FORTRAN worlds is done exclusively via wrappers. This makes it possible to build shareable libraries for the C++ code, even if it calls FORTRAN code internally.

Chapter 15
Analysis utilities

15.1 Overview

In this chapter we give pointers to some of the third party software libraries that we use within Athena or recommend for use by algorithms implemented in Athena.

15.2 CLHEP

CLHEP (Class Library for High Ener gy Physics) is a set of HEP-specific foundation and utility classes such as random generators, physics vectors, geometry and linear algebra. It is structured in a set of packages independent of any external package. The documentation for CLHEP can be found on WWW at *http://wwwinfo.cern.ch/asd/lhc++/clhep/index.html*

CLHEP is used extensively inside Athena, in the `GaudiSvc` and `GaudiDb` packages.

15.3 HTL

HTL ("Histogram Template Library") is used internally in Athena (GaudiSvc package) to provide histogramming functionality. It is accessed through its abstract AIDA[16] compliant interfaces. Athena uses only the transient part of HTL. Histogram persistency is available with ROOT or HBOOK.

The documentation on HTL is available at *http:cern.ch/anaphe/documentation.html*.

15.4 NAG C

The NAG C library is a commercial mathematical library providing a similar functionality to the FORTRAN mathlib (part of CERNLIB). It is organised into chapters, each chapter devoted to a branch of numerical or statistical computation. A full list of the functions is available at *http://cern.ch/anaphe/documentation/Nag_C/NAGdoc/cl/html/mark6.html*

NAG C is not explicitly used in the Athena framework, but developers are encouraged to use it for mathematical computations. Instructions for linking NAG C with Gaudi can be found at *http://cern.ch/lhcb-comp/Support/NagC/nagC.html*

Some NAG C functions print error messages to `stdout` by default, without any information about the calling algorithm and without filtering on severity level. A facility is provided by Gaudi to redirect these messages to the Gaudi MessageSvc. This is documented at *http://cern.ch/lhcb-comp/Support/NagC/GaudiNagC.html*

15.5 ROOT

ROOT is used by Athena for I/O and as a persistency solution for event data, histograms and n-tuples. In addition, it can be used for interactive analysis, as discussed in Chapter 9. Information about ROOT can be found at *http://root.cern.ch/*

Appendix A
Options for standard components

The following is a list of options that may be set for the standard components: e.g. data files for input, print-out level for the message service, etc. The options are listed in tabular form for each component along with the default value and a short explanation. The component name is given in the table caption thus: [ComponentName].

Table A.1 Standard Options for the Application manager [ApplicationMgr]

Option name	Default value	Meaning
EvtSel	""	If "NONE", no event input[a]
EvtMax	-1	Maximum number of events to process. The default is -1 (infinite) unless EvtSel = "NONE"; in which case it is 10.
TopAlg	{}	List of top level algorithms. Format: {<Type>/<Name>[, <Type2>/<Name2>,...]};
ExtSvc	{}	List of external services to be explicitly created by the ApplicationMgr (see section 10.2). Format: {<Type>/<Name>[, <Type2>/<Name2>,...]};
OutStream	{}	Declares an output stream object for writing data to a persistent store, e.g. { DstWriter }; See also Table A.10
DLLs	{}	Search list of libraries for dynamic loading. Format: {<dll1>[,<dll2>,...]};
HistogramPersistency	"NONE"	Histogram and N-tuple persistency mechanism. Available options are "HBOOK", "ROOT", "NONE"
Runable	"AppMgrRunable"	Type of runable object to be created by Application manager

Table A.1 Standard Options for the Application manager [ApplicationMgr]

Option name	Default value	Meaning
EventLoop	"EventLoopMgr"	Type of event loop: "EventLoopMgr" is standard event loop "MinimalEventLoop" executes algorithms but does not read events
OutputLevel	MSG::INFO	Same as MessageSvc.OutputLevel. See Table A.2 for possible values
The last two options define the source of the job options file and so they cannot be defined in the job options file itself. There are two possibilities to set these options, the first one is using a environment variable called JOBOPTPATH or setting the option to the application manager directly from the main program[b]. The coded option takes precedence.		
JobOptionsType	FILE	Type of file (FILE implies ascii)
JobOptionsPath	jobOptions.txt	Path for job options source

a. A basic DataObject object is created as event root ("/Event")

b. The setting of properties from the main program is discussed in Chapter 2.

Table A.2 Standard Options for the message service [MessageSvc]

Option name	Default value	Meaning
OutputLevel	0	Verboseness threshold level: 0=NIL,1=VERBOSE, 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL, 7=ALWAYS
Format	% F%18W%S%7W%R%T %0W%M	Format string.

Table A.3 Standard Options for all algorithms [<myAlgorithm>]

Any algorithm derived from the Algorithm base class can override the global Algorithm options thus:		
Option name	**Default value**	**Meaning**
OutputLevel	0	Message Service Verboseness threshold level. See Table A.2 for possible values
Enable	true	If false, application manager skips execution of this algorithm
ErrorMax	1	Job stops when this number of errors is reached

Table A.3 Standard Options for all algorithms [<myAlgorithm>]

Any algorithm derived from the Algorithm base class can override the global Algorithm options thus:		
Option name	**Default value**	**Meaning**
ErrorCount	0	Current error count
AuditInitialize	false	Enable/Disable auditing of Algorithm initialisation
AuditExecute	true	Enable/Disable auditing of Algorithm execution
AuditFinalize	false	Enable/Disable auditing of Algorithm finalisation

Table A.4 Standard Options for all services [<myService>]

Any service derived from the Service base class can override the global `MessageSvc.OutputLevel` thus:		
Option name	**Default value**	**Meaning**
OutputLevel	0	Message Service Verboseness threshold level. See Table A.2 for possible values

Table A.5 Standard Options for all Tools [<myTool>]

Any tool derived from the AlgTool base class can override the global `MessageSvc.OutputLevel` thus:		
Option name	**Default value**	**Meaning**
OutputLevel	0	Message Service Verboseness threshold level. See Table A.2 for possible values

Table A.6 Standard Options for all Associators [<myAssociator>]

Option name	**Default value**	**Meaning**
FollowLinks	true	Instruct the associator to follow the links instead of using cached information
DataLocation	""	Location where to get association information in the data store

Table A.7 Standard Options for Auditor service [AuditorSvc]

Option name	**Default value**	**Meaning**
Auditors	{};	List of Auditors to be loaded and to be used. See section 10.7 for list of possible auditors

Table A.8 Standard Options for all Auditors [<myAuditor>]

Any Auditor derived from the Auditor base class can override the global Auditor options thus:		
Option name	**Default value**	**Meaning**
OutputLevel	0	Message Service Verboseness threshold level. See Table A.2 for possible values
Enable	true	If false, application manager skips execution of the auditor

Table A.9 Options of Algorithms in GaudiAlg package (see Section 3.5)

Algorithm name	Option Name	Default value	Meaning
EventCounter	Frequency	1;	Frequency with which number of events should be reported
Prescaler	PercentPass	100.0;	Percentage of events that should be passed
Sequencer	Members		Names of algorithms in the sequence
Sequencer	BranchMembers		Names of algorithms on the branch
Sequencer	StopOverride	false;	If true, do not stop sequence if a filter fails

Table A.10 Options available for output streams (e.g. DstWriter)

Output stream objects are used for writing user created data into data files or databases. They are created and named by setting the option `ApplicationMgr.OutStream`. For each output stream the following options are available		
Option name	**Default value**	**Meaning**
ItemList	{}	The list of data objects to be written to this stream, e.g. { /Event#1 , Event/MyT racks/#1 };
Preload	true;	Preload items in ItemList
Output	""	Output data stream specification. Format: { DATAFILE='mydst.root'TYP='ROOT' };
OutputFile	""	Output file specification - same as DATAFILE in previous option
EvtDataSvc	EventDataSvc	The service from which to retrieve objects.
EvtConversion-Svc	"EventPersisten-cySvc"	The persistency service to be used
AcceptAlgs	{}	If any of these algorithms sets filterflag=true; the event is accepted
RequireAlgs	{}	If any of these algorthms is not executed, the event is rejected
VetoAlgs	{}	If any of these algorithms does not set filterflag = true; the event is rejected

Table A.11 Standard Options for persistency services (e.g. EventPersistencySvc)

Option name	Default value	Meaning
CnvServices	{}	Conversion services to be used by the service to load or store persistent data (e.g. "RootEvtCnvSvc")

Table A.12 Standard Options for conversion services (e.g. RootEvtCnvSvc)

Option name	Default value	Meaning
DbType	""	Persistency technology (e.g. "ROOT")

Table A.13 Standard Options for the histogram service [HistogramPersistencySvc]

Option name	Default value	Meaning
OutputFile	""	Output file for histograms. Histograms not saved if not given.
RowWiseNTuplePolicy	"FLOAT_ONLY"	Persistent representation of NTuple data types. Other possible value is "USE_DATA_TYPES". See Section 9.2.3.2 for details
PrintHistos	false	Print the histograms also to standard output (HBOOK only)

Table A.14 Standard Options for the N-tuple service [NTupleSvc] (see Section 9.2.2)

Option name	Default value	Meaning
Input	{}	Input file(s) for n-tuples. Format: { FILE1 DATAFILE='tuple1.typ' OPT='OLD' , [FILE2 DATAFILE='tuple2.typ' OPT='OLD' ,...]}
Output	{}	Output file(s) for n-tuples. Format: { FILE1 DATAFILE='tuple1.typ' OPT='NEW' , [FILE2 DATAFILE='tuple2.typ' OPT='NEW ,...]}
StoreName	"/NTUPLES"	Name of top level entry

Table A.15 Standard Options for the Event Collection service [TagCollectionSvc] (see Section 9.3.1)

Option name	Default value	Meaning
Output	{}	Output file specification. See Section 9.3.2 for details
StoreName	"/NTUPLES"	Name of top level entry

Table A.16 Standard Options for the standard event selector [EventSelector]

Option name	Default value	Meaning
Input	{}	Input data stream specification. Format: "<tagname> = <tagvalue> <opt>" Possible tags are different depending on input data type. For Event data, see Section 7.10.2 For Event Collections, see Section 9.3.2.3
FirstEvent	1	First event to process (allows skipping of preceding events)
PrintFreq	10	Frequency with which event number is reported

Table A.17 Event Tag Collection Selector [EventCollectionSelector]

The following options are used internally by the EventCollectionSelector. They should not normally be used directly by users, who should set them via the "tags" of the EventSelector.Input option

Option name	Corresponding tag of EventSelector.Input	Default value	Meaning
CnvService	SVC	EvtT upleSvc	Conversion service to be used
Authentication	AUTH	""	Authentication to be used
Container		"B2PiPi"	Container name
Item		"Address"	Item name
Criteria	SEL	""	Selection criteria
DB	DATAFILE	""	Database name
DbType	TYP	""	Database type
Function	FUN	"NTuple::Selector"	Selection function

Table A.18 Standard Options for Random Numbers Generator Service [RndmGenSvc]

Option name	Default value	Meaning
Engine	HepRndm::Engine<RanluxEngine>	Random number generator engine
Seeds		Table of generator seeds
Column	0	Number of columns in seed table -1
Row	1	Number of rows in seed table -1
Luxury	3	Luxury value for the generator
UseTable	false	Switch to use seeds table

Table A.19 Standard Options for Particle Property Service [ParticlePropertySvc]

Option name	**Default value**	**Meaning**
ParticlePropertiesFile	($LHCBDBASE)/cdf/particle.cdf	Particle properties database location

Table A.20 Standard Options for Chrono and Stat Service [ChronoStatSvc]

Option name	**Default value**	**Meaning**
ChronoPrintOutTable	true	Global switch for profiling printout
PrintUserTime	true	Switch to print User Time
PrintSystemTime	false	Switch to print System Time
PrintEllapsedTime	false	Switch to print Elapsed time (Note typo in option name!)
ChronoDestinationCout	false	If true, printout goes to cout rather than MessageSvc
ChronoPrintLevel	3	Print level for profiling (values as for MessageSvc)
ChronoTableToBeOrdered	true	Switch to order printed table
StatPrintOutTable	true	Global switch for statistics printout
StatDestinationCout	false	If true, printout goes to cout rather than MessageSvc
StatPrintLevel	3	Print level for profiling (values as for MessageSvc)
StatTableToBeOrdered	true	Switch to order printed table

A.1 Obsolete options

The following options are obsolete and should not be used. They are documented here for completeness and may be removed in a future release.

Table A.21 Obsolete Options

Obsolete Option	**Replacement**
EventSelector.EvtMax	ApplicationMgr.EvtMax (Table A.1)

Appendix B
Design considerations

B.1 Generalities

In this chapter we look at how you might actually go about designing and implementing a real physics algorithm. It includes points covering various aspects of software development process and in particular:

¥ The need for more thinking before coding when using an OO language like C++.

¥ Emphasis on the specification and analysis of an algorithm in mathematical and natural language, rather than trying to force it into (unnatural?) object orientated thinking.

¥ The use of OO in the design phase, i.e. how to map the concepts identified in the analysis phase into data objects and algorithm objects.

¥ The identification of classes which are of general use. These could be implemented by the computing group, thus saving you work!

¥ The structuring of your code by defining private utility methods within concrete classes.

When designing and implementing your code we suggest that your priorities should be as follows: (1) Correctness, (2) Clarity, (3) Efficiency and, very low in the scale, OOness

Tips about specific use of the C++ language can be found in the coding rules document [11] or specialized literature.

B.2 Designing within the Framework

A physicist designing a real physics algorithm does not start with a white sheet of paper. The fact that he or she is using a framework imposes some constraints on the possible or allowed designs. The framework defines some of the basic components of an application and their interfaces and therefore it also specifies the places where concrete physics algorithms and concrete data types will fit in with the rest of the program. The consequences of this are: on one hand, that the physicists designing the algorithms do not have complete freedom in the way algorithms may be implemented; but on the other hand, neither do they need worry about some of the basic functionalities, such as getting end-user options, reporting messages, accessing event and detector data independently of the underlying storage technology, etc. In other words, the framework imposes some constraints in terms of interfaces to basic services, and the interfaces the algorithm itself is implementing towards the rest of the application. The definition of these interfaces establishes the so called master walls of the data processing application in which the concrete physics code will be deployed. Besides some general services provided by the framework, this approach also guarantees that later integration will be possible of many small algorithms into a much larger program, for example a reconstruction program. In any case, there is still a lot of room for design creativity when developing physics code within the framework and this is what we want to illustrate in the next sections.

To design a physics algorithm within the framework you need to know very clearly what it should do (the requirements). In particular you need to know the following:

¥ What is the input data to the algorithm? What is the relationship of these data to other data (e.g. event or detector data)?

¥ What new data is going to be produced by the algorithm?

¥ What s the purpose of the algorithm and how is it going function? Document this in terms of mathematical expressions and plain english.[1]

¥ What does the algorithm need in terms of configuration parameters?

¥ How can the algorithm be partitioned (structured) into smaller algorithm chunks that make it easier to develop (design, code, test) and maintain?

¥ What data is passed between the different chunks? How do they communicate?

¥ How do these chunks collaborate together to produce the desired final behaviour? Is there a controlling object? Are they self-organizing? Are they triggered by the existence of some data?

¥ How is the execution of the algorithm and its performance monitored (messages, histograms, etc.)?

¥ Who takes the responsibility of bootstrapping the various algorithm chunks.

For didactic purposes we would like to illustrate some of these design considerations using a hypothetical example. Imagine that we would like to design a tracking algorithm based on a Kalman-filter algorithm.

1. Catalan is also acceptable.

B.3 Analysis Phase

As mentioned before we need to understand in detail what the algorithm is supposed to do before we start designing it and of course before we start producing lines of C++ code. One old technique for that, is to think in terms of data flow diagrams, as illustrated in Figure A.1, where we have tried to decompose the tracking algorithm into various processes or steps.

Figure A.1 Hypothetical decomposition of a tracking algorithm based on a Kalman filter using a Data flow Diagram

In the analysis phase we identify the data which is needed as input (event data, geometry data, configuration parameters, etc.) and the data which is produced as output. We also need to think about the intermediate data. Perhaps this data may need to be saved in the persistency store to allow us to run a part of the algorithm without starting always from the beginning.

We need to understand precisely what each of the steps of the algorithm is supposed to do. In case a step becomes too complex we need to sub-divide it into several ones. Writing in plain english and using

mathematics whenever possible is extremely useful. The more we understand about what the algorithm has to do the better we are prepared to implement it.

B.4 Design Phase

We now need to decompose our physics algorithm into one or more Algorithms (as framework components) and define the way in which they will collaborate. After that we need to specify the data types which will be needed by the various Algorithms and their relationships. Then, we need to understand if these new data types will be required to be stored in the persistency store and how they will map to the existing possibilities given by the object persistency technology. This is done by designing the appropriate set of Converters. Finally, we need to identify utility classes which will help to implement the various algorithm chunks.

B.4.1 Defining Algorithms

Most of the steps of the algorithm have been identified in the analysis phase. We need at this moment to see if those steps can be realized as framework Algorithms. Remember that an Algorithm from the view point of the framework is basically a quite simple interface (initialize, execute, finalize) with a few facilities to access the basic services. In the case of our hypothetical algorithm we could decide to have a master Algorithm which will orchestrate the work of a number of sub-Algorithms. This master Algorithm will be also be in charge of bootstraping them. Then, we could have an Algorithm in charge of finding the tracking seeds, plus a set of others, each one associated to a different tracking station in charge of propagating a proto-track to the next station and deciding whether the proto-track needs to be kept or not. Finally, we could introduce another Algorithm in charge of producing the final tracks from the surviving proto-tracks.

It is interesting perhaps in this type of algorithm to distribute parts of the calculations (extrapolations, etc.) to more sophisticated hits than just the unintelligent original ones. This could be done by instantiating new data types (clever hits) for each event having references to the original hits. For that, it would be required to have another Algorithm whose role is to prepare these new data objects, see Figure A.2.

The master Algorithm (TrackingAlg) is in charge of setting up the other algorithms and scheduling their execution. It is the only one that has a global view but it does not need to know the details of how the different parts of the algorithm have been implemented. The application manager of the framework only interacts with the master algorithm and does not need to know that in fact the tracking algorithm is implemented by a collaboration of Algorithms.

B.4.2 Defining Data Objects

The input, output and intermediate data objects need to be specified. Typically, the input and output are specified in a more general way (algorithm independent) and basically are pure data objects. This is

TrackingAlg
HitPreprocessor
SeedFinder
Tracker
StationProcessor
HitSet
nHitSet
ProtoTrack Set
Track Set
Hit
nHit
PTrack
PTrack
Event Data Store

IAlgorithm
Algorithm
DataObject
Station
TrackingAlg
HitPreprocessor
SeedFinder
StationProcessor
Tracker
HitSet
nHitSet
ProtoTack Set
TackSet
Hit
nHit
PTrack
Track

Figure A.2 Object diagram (a) and class diagram (b) showing how the complete example tracking algorithm could be decomposed into a set of specific algorithms that collaborate to perform the complete task.

because they can be used by a range of different algorithms. We could have various types of tracking algorithm all using the same data as input and producing similar data as output. On the contrary, the intermediate data types can be designed to be very algorithm dependent.

The way we have chosen to communicate between the different Algorithms which constitute our physics algorithm is by using the transient event data store. This allows us to have low coupling between them, but other ways could be envisaged. For instance, we could implement specific methods in the algorithms and allow other friend algorithms to use them directly .

Concerning the relationships between data objects, it is strongly discouraged to have links from the input data objects to the newly produced ones (i.e. links from hits to tracks). In the other direction this should not be a problem (i.e from tracks to constituent hits).

For data types that we would like to save permanently we need to implement a specific Converter. One converter is required for each type of data and each kind of persistency technology that we wish to use. This is not the case for the data types that are used as intermediate data, since these data are completely transient.

B.4.3 Mathematics and other utilities

It is clear that to implement any algorithm we will need the help of a series of utility classes. Some of these classes are very generic and they can be found in common class libraries. For example the standard template library. Other utilities will be more high energy physics specific, especially in cases like fitting, error treatment, etc. We envisage making as much use of these kinds of utility classes as possible.

Some algorithms or algorithm-parts could be designed in a way that allows them to be reused in other similar physics algorithms. For example, perhaps fitting or clustering algorithms could be designed in a generic way such that they can be used in various concrete algorithms. During design is the moment to identify this kind of re-usable component or to identify existing ones that could be used instead and adapt the design to make possible their usage.

Appendix C
Job Options Grammar

C.1 The EBNF grammar of the Job Options files

The syntax of the Job-Options-File is defined through the following EBNF-Grammar.

Job-Options-File =

{Statements} .

Statements =

{Include-Statement} | {Assign-Statement} | {Append-Statement} | {Platform-Dependency} .

AssertableStatements =

{Include-Statement} | {Assign-Statement} | {Append-Statement} .

AssertionStatement =

#ifdef | #ifndef .

Platform-Dependency =

AssertionStatement WIN32 <AsertableStatements> [#else <AssertableStatements>] #endif

Include-Statement =

#include string .

Assign-Statement =

Identifier . Identifier = value ; .

Append-Statement =

Identifier . Identifier += value ; .

Identifier =

letter {letter | digit} .

value =

boolean | integer | double | string | vector .

vector =

{ vectorvalue { , vectorvalue } } .

vectorvalue =

boolean | integer | double | string .

boolean =

true | false .

integer =

prefix scientificdigit .

double =

(prefix <digit> . [scientificdigit]) |
(prefix . scientificdigit) .

string =

{char} .

scientificdigit =

< digit> [(e | E) < digit>] .

digit =

<figure> .

prefix =

[+ | -] .

figure =

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .

char =

any character from the ASCII-Code

letter =

set of all capital- and non-capital letter

C.2 Job Options Error Codes and Error Messages

The table below lists the error codes and error messages that the Job Options compiler may generate, their reason and how to avoid them.

Table 15.1 Possible Error-Codes

Error-Code	Reason	How to avoid it
Error #000 Internal compiler error	-	This code normally should never appear. If this code is shown there is maybe a problem with your memory, your disk-space or the property-file is corrupted.
Error #001 Included property-file does not exists or can not be opened	* wrong path in #include-directive * wrong file or mistyped filename * file is exclusively locked by another application * no memory available to open this file	Please check if any of the listed reasons occured in your case.
Warning #001 File already included by another file	The file was already included by another file and will not be included a second time. The compiler will ignore this #include-directive and will continue with the next statement.	Remove the #include-directive
Error #002 syntax error: Object expected	The compiler expected an object at the given position.	Maybe you mistyped the name of the object or the object contains unknown characters or does not fit the given rules.
Error #003 syntax error: Missing dot between Object and Propertyname	The compiler expect a dot between the Object and the Propertyname.	Check if the dot between the Object and the Propertyname is missing.
Error #004 syntax error: Identifier expected	The compiler expected an identifier at the given position.	Maybe you mistyped the name of the identifier or the identifier contains unknown characters or does not fit the given rules.
Error #005 syntax error: Missing operator += or =	The compiler expected an operator between the Propertyname and the value.	Check if there is a valid operator after the Propertyname. Note that a blank or tab is not allowed between += !

Table 15.1 Possible Error-Codes

Error-Code	**Reason**	**How to avoid it**
Error #006 String is not terminated by a	A string (value) was not terminated by a .	Check if all your strings are beginning and ending with . Note that the position given by the compiler can be wrong because the compiler may thought that following statements are part of the string!
Error #007 syntax error: #include-statement is not correct	The next token after the #include is not a string.	Make sure that after the #include-directive there is specified the file to include. The file must be defined as a string!
Error #008 syntax error: #include does not end with a ;	The include-directive was terminated by a ;	Remove the ; after the #include-directive.
Error #009 syntax error: Values must be separated with ,	One or more values within a vector were not separated with a , or one ore more values within a vector are mistyped.	Check if every value in the vector is separated by a , . If so the reason for this message may result in mistyped values in the vector (maybe there is a blank or tab between numbers).
Error #010 syntax error: Vector must end with }	The closing bracket is missing or the vector is not terminated correctly.	Check, if the vector ends with a } and if there is no semicolon before the ending-bracket.
Error #011 syntax error: Statement must end with a ;	The statement is not terminated correctly.	Check if the statement ends with a semicolon ; .
Runtime-Error #012: Cannot append to object because it does not exists	The compiler cannot append the values to the object.propertyname because the object does not exist.	Check if the refered object is defined in one of the included files, if so check if you writed the object-name exactly like in the include-file.
Runtime-Error #013 Cannot append to object because Property does not exists	The compiler cannot append the values to the object.propertyname because the property does not exist.	Check if there was already something assigned to the refered property (in the include-file or in the current file). If not then modify the append-statement into a assign-statement. If there was already something assigned, check if the object-name and the property-name are typed correctly.

Table 15.1 Possible Error-Codes

Error-Code	Reason	How to avoid it
Error #014 Elements in the vector are not of the same type	One or more elements in the vector have a different type than the first element in the vector. All elements must have the same type like the first declarated element.	Check declaration of vector, check the types and check, if maybe a value is mistyped.
Error #015 Value(s) expected	The compiler didn t find values to append or assign	Check the statement if there exists values and if they are written correctly. Maybe this error is a result of a previous error!
Error #016 Specified property-file does not exist or can not be resolved	The compiler was not able to include a property-file or didn t found the file. A reason can be that the compiler was not able to resolve an environment-variable which points to the location of the property-file.	Check if you are using enviornment-variables to resolve the file, if they are mistyped (wether in the system or in the #include-directive) or not set correctly.
Error #017 #ifdef not followed by an identifier	The #ifdef-statement is not followed by the assertion-identifier (WIN32).	Add WIN32 after the #ifdef-statement.
Error #018 identifier in #ifdef / #ifndef not known	The assertion-identifier used in the #ifdef- /#ifndef-statement is not known. At the moment there can only be used WIN32!	Change identifier to WIN32.
Error #019 #ifdef / #ifndef / #else / #endif doesn t end with a ;	A semicolon was found after the #ifdef- / #ifndef- / #else- / #endif-statement. These statements don t end with a semicolon.	Remove the semicolon after the #ifdef / #ifndef / #else / #endif-statement.

Appendix D
The ATLAS Development Model

D.1 Overview

The goal is to fulfill the roles of:

- A component library, which is a black-box with a single entrypoint for declaring factories for components (Algorithms, Services, etc.). A component library is not designed to allow it's internals to be accessed - it just provides factories for it's components. These are accessed via job options files at run-time. Thus access to component libraries is a run-time requirements, not a compile or link-time requirement.
- A linkable library, that provides classes that can be inherited from and linked against, either statically, or dynamically at run-time.
- Satisfy the requirement of being able to sub-class from an Algorithm or Service. This mixed the roles of 1. and 2. above.

Note that the term installed library is used interchangeably with linkable library.

D.2 Strategy

The overall strategy is to provide three patterns that support the three cases described above. The three patterns are:

component_library

installed_library

dual_use_library

These are discussed in detail in the following section. The developer of a package should apply one of these as appropriate. In order to simplify things, the rules are:

1. Packages that create Algorithms should use the dual_use_library pattern.
2. Packages that create utility classes that are designed to be used directly or as base classes should use the installed_library pattern.
3. Packages that create a Service should use the component_library pattern. In general a single package should not create multiple services.

D.3 Component Libraries

The syntax for the component_library pattern within a package Pkg is:

```
library Pkg <list of files>
apply_pattern component_library
```

Within such a package, the component factories are setup using files Pkg_entries.cxx and Pkg_load.cxx which can either reside in the src directory for backwards compatibility, or in the src/component directory for consistency with packages using the dual_use_library pattern (see next section). In the latter case, the library should be built using a line of the form:

```
library Pkg <list of files> -s=components *.cxx
```

The required syntax of these files is unchanged from existing Athena documentation. However it is recommended that the the format be modified as described in the Dual Use Library section, in conjunction with removing the factory code from each Algorithm or Service header file.

Note that component libraries consist of a single library, having the same name as the package name.

D.4 Dual Use Libraries

The syntax for the dual_use_library pattern within a package Pkg is:

```
apply_pattern dual_use_library files=<list of files>
```

The list of files should be an explicit list in quotes, or may be wildcarded:

```
files="Foo.cxx Bar.cxx"
files=*.cxx
```

Two component declaration files must reside in the src/components directory. They are:

Pkg_entries.cxx

Pkg_load.cxx

The former (Pkg_entries.cxx) must contain an entry for each Algorithm or Service that is declared by this package. An example is:

```
#include "Pkg/MyAlgorithm1.h"
#include "Pkg/MyAlgorithm2.h"
#include "GaudiKernel/DeclareFactoryEntries.h"

DECLARE_ALGORITHM_FACTORY( MyAlgorithm1 );
DECLARE_ALGORITHM_FACTORY( MyAlgorithm2 );
DECLARE_FACTORY_ENTRIES( Pkg ) {
   DECLARE_ALGORITHM( MyAlgorithm1 );
   DECLARE_ALGORITHM( MyAlgorithm2 );
}
```

The latter (Pkg_load.cxx) must contain an entry for the package itself. An example is:

```
#include "GaudiKernel/LoadFactoryEntries.h"
LOAD_FACTORY_ENTRIES( Pkg )
```

In conjunction with these files, the factory declarations that used to appear at the head of Algorithm and Service .cxx files should be removed. An example of the declaration that should be removed is:

```
#include "GaudiKernel/AlgFactory.h"

static const AlgFactory<MyAlg>       s_factory;
const IAlgFactory& MyAlgAlgFactory = s_factory;
```

These lines are obsolete, having been replaced by the DECLARE_ALGORITHM_FACTORY macros used by the Pkg_entries.cxx file.

Note that dual use libraries cause two separate libraries to be created:

libXxx.so Component library

libXxxLib.so Linkable/Installed library

This is backwards compatible with our existing kludge and existing job options files and Python scripts can remain unchanged since the component library has the same name. If another packages subclasses a class from this package, then the header files are exported, as is the libXxxLib.so library, which is the library to be linked against.

D.5 Installed or Linkable Libraries

The syntax for the installed_library pattern within a package Pkg is:

library Pkg <list of files>

apply_pattern installed_library

This creates a single library with the same name as the package which is designed to be linked against. It also exports header files.

Appendix E
Package and Directory Structure

E.1 Subsystem Package Organization

In the following, Xxx is some sensible subdivision of the system - detector subsystem, tracking, particle id,etc.

```
XxxAlgs       Algorithms (and optionally helper classes)
XxxSvc        A Service (and optionally helper classes)
XxxUtils      Utility/Helper classes (optional)
XxxEvent      Event data
XxxDetDescr   Detector Description data
XxxConditions Conditions data
```

Figure 15.1 Package Organization

A finer subdivision of packages is also supported, if this minimizes dependencies or overloading of a single package, thus.

```
XxxSimEvent
XxxRawEvent
XxxRecEvent
```

Figure 15.2 Alternative Package Organization

Similarly for Algorithms and Conditions data. Possibilities for Algorithms could be phases of the reconstruction chain, or separating those associated with the generation of calibrations from those associated with normal event processing etc. Similarly Conditions data could separate out calibration from alignment data.

In the above, a utility or helper class is an algorithmic class that can be used by classes from multiple other packages. If they are designed to be used by a particular Algorithm or Service, they can reside in

the same package as that. However, if they have a more general usefulness, they should reside in separate packages, perhaps not even as part of the Xxx subsystem, but in a more general package structure (e.g. Tools or Utilities).

E.2 Utilities Package Directory Structure

The structure for packages that contain classes that are utility or helper or algorithmic code is as follows (using XxxUtils as an example):

```
XxxUtils/cmt                        Normal CMT directory
        /XxxUtils                   Public Header files
        /src                        All source files and private header files
```

Figure 15.3 Utilities Package Directory Structure

Notes:

1. These packages create a shared library that is designed to be linked against.
2. The `installed_library` pattern for shared libraries should be used as described in Appendix 14.4.2.

E.3 Algorithm and Service Package Directory Structure

The structure for packages that declare Algorithms or Services is as follows (using XxxAlgs as an example):

```
XxxAlgs/cmt                         Normal CMT directory
       /XxxAlgs                     Public Header files
       /src                         All source files and private header files
       /src/components              Component library .cxx files
       /share                       Job Options files etc.
                                    [ this name is historical from SRT days]
```

Figure 15.4 Algorithm and Service Package Directory Structure

Notes:

1. These packages should use one of two patterns:
    1. `component_library` - for simple component libraries
    2. `dual_use_library` - for Algorithms or Services that are capable of being inherited from.

   Thesze are described in detail in Appendix 14.4.1 and Appendix 14.4.3.

E.4 Data Package Directory Structure

The structure for the XxxEvent, XxxDetDescr and XxxConditions packages is as follows (using XxxEvent as an example):

```
XxxEvent/cmt                       Normal CMT directory.
        /XxxEvent                  Header .h files
        /src                       Source .cxx files
        /src/components            Component library .cxx files
```

Figure 15.5 Data PackagePackage Directory Structure

Notes:

Appendix F
Standard ATLAS Patterns and Variables

F.1 Overview

This appendix describes the standard ATLAS patterns that are declared in the AtlasPolicy package.

F.2 Platform Environment Variables

Standard values of the CMTCONFIG environment variable

CMTCONFIG	Platform	Compiler	Options
Linux-gcc-dbg	Linux	gcc 2.95.2	debug
Linux-gcc-opt	Linux	gcc 2.95.2	optimized (-O2)
Linux-gcc-prof	Linux	gcc 2.95.2	profiled (-pg) optimized (-O2)
Solaris-gcc-dbg	Solaris	CC 5.1/2	debug
Solaris-gcc-opt	Solaris	CC 5.1/2	optimized (O2)
Solaris-gcc-prof	Solaris	CC 5.1/2	profiled (-pg) optimized (-O2)

The current Solaris compiler, although reporting itself as CC 5.1, has been patched to the CC 5.2 functionality.

F.3 Patterns controlling include paths

include_path

Useage:

apply_pattern include_path [extras="<dirs>"]

Description:

Adds the list of directories specified by the "extras"

argument to the -I include search path. The directories

should be specified relative to the cmt/ directory.

no_include_path

Useage:

apply_pattern no_include_path

Description:

Disables the default -I include path such that none is

setup. This pattern may be combined with the "include_path"

pattern to override the default include search path.

F.4 Patterns controlling library creation

installed_library

component_library

dual_use_library

These are described in the Appendix "Installed, Component and Dual Use Libraries".

default_library

default_installed_library

F.5 Patterns controlling linker options

default_linkopts

default_no_share_linkopts

installed_linkopts

F.6 Patterns for establishing a run-time environment

declare_runtime

Useage:

apply_pattern declare_runtime [extras="<files>"]

Description:

Declares that the .txt and .py files in the share/ directory should be installed in the target run/ directory. This default may be extended by specifying the optional "extras" argument.

declare_runtime_extras

Useage:

apply_pattern declare_runtime_extras [extras="<files>"]

Description:

Similar to "declare_runtime" but does not declare any files by default. The list of run-time files should be declared using the "extras" argument.

install_runtime

Useage:

declare_pattern install_runtime

Description:

Creates the run/ directory in the calling package, and installs all files that were declared by other packages

using the "declare_runtime" or "declare_runtime_extras"

patterns.

Appendix G
References

[1] GAUDI User Guide
http://lhcb-comp.web.cern.ch/lhcb-comp/Components/Gaudi_v6/gug.pdf

[2] GAUDI - Architecture Design Report [LHCb 98-064 COMP]

[3] HepMC Reference

[4] Python Reference

[5] StoreGate Design Document

A
AIDA 181
    see Interfaces
Algorithm 15
    Base class 15, 21
    branches 29
    Concrete 21, 25
    Constructor 23, 25
    Declaring properties 24
    Execution 26
    Filters 29
    Finalisation 26
    Initialisation 23, 26
    Nested 28
    sequences 29
    Setting properties 24
Algorithms
    EventCounter 30, 116
    Prescaler 30
    Sequencer 29
Application Manager 16
Architecture 13
Associators 152
    Example 155
B
Branches 29
C
Checklist
    for implementing algorithms 28
CLHEP 181
Component 13, 173
    libraries 173, 176
Converters 157
D
Data Store
    Histograms 95
DataObject 15
DECLARE_ALGORITHM 174
DECLARE_FACTORY_ENTRIES 174
E
endreq, MsgStream manipulator 124
Event Collections 105
    Filling 106

Reading Events with 108
Writing 105
EventCounter algorithm. See Algorithms
Examples
Associator 155
F
Factory
for a concrete algorithm 24
Filters 29
FORTRAN 15
and shareable libraries 179
G
getFactoryEntries 173
Guidelines
for software packaging 169
H
HBOOK
Constraints on histograms 96
For histogram persistency 97
Limitations on N-tuples 100, 101, 102, 104
Histograms
data service 95
HTL 181
Persistency service 97
HTL 181
I
Inheritance 21
Interactive Analysis
of N-tuples 110
Interface 13
and multiple inheritance 17
Identifier 18, 171
In C++ 17
Interfaces
AIDA 95, 181
IAlgorithm 17, 21, 23, 26
IAlgTool 146
IAssociator 152
IAuditor 132
IConversionSvc 159
IConverter 159
IDataManagerSvc 16
IDataProviderSvc 16, 99
IDataProvideSvc 95

IHistogram1D 95
IHistogram2D 95
IHistogramSvc 17, 95
IIncidentListener 135
IMessageSvc 17
in Gaudi 170
INTupleSvc 99
INtupleSvc 17
IOpaqueAddress 159
IParticlePropertySvc 125
IProperty 17, 22
IRunable 18
ISvcLocator 23
IToolSvc 150
Navigating between 172
Introspection 137
J
Job Options
see also Properties
Job options 115
L
Libraries
Component 173, 176
containing FORTRAN code 179
Linker 177
LOAD_FACTORY_ENTRIES 175
M
Message service 122
Monitoring
of algorithm calls, with the Auditor service 131
statistical, using the Chrono&stat service 129
Monte Carlo truth
navigation using Associators 152
N
NAG C 182
N-tuples 99
Booking and declaring tags 101
filling 102
Interactive Analysis of 110
Limitations imposed by HBOOK 100, 101, 102, 104
persistency 103, 107
reading 103
Service 99

P
Packages
  Dependencies of Gaudi 169
  Guidelines 169
PAW
  for N-Tuple analysis 103
Persistency
  of histograms 97
  of N-tuples 103, 107
Prescaler algorithm. See Algorithms
Profiling
  of execution time, using the Chrono&Stat service 128
  of execution time, with the Auditor service 131
  of memory usage, with the Auditor service 131
Properties
  Accessing and Modifying 118
R
Random numbers
  generating 132
  Service 132
Retrieval 150
ROOT 182
  for histogram persistency 98
  for N-Tuple analysis 103, 110
S
Sequencer algorithm. See Algorithms
Sequences 29
Services 16
  Auditor Service 130
  Chrono&Stat service 127
  Histogram data service 95
  Histogram Persistency Services 97
  Incident service 135
  Introspection service 137
  Job Options service 115
  Message Service 122
  N-tuples Service 99
  Particle Properties Service 124
  Random numbers service 132
  requesting and accessing 113
  ToolSvc 143, 149
  vs. Tools 143
StatusCode 26

T

Tools 143
- Associators 152
- provided in Gaudi 151
- vs. Services 143

ToolSvc, see Services