ATLAS muon reconstruction from a C++ perspective: a road to the Higgs
Hendriks, P.J.

Citation for published version (APA):
Hendriks, P. J. (2000). ATLAS muon reconstruction from a C++ perspective: a road to the Higgs

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
3.1 Global Architecture

The goal of the software described in this thesis is to perform the stand-alone reconstruction of events in the ATLAS muon spectrometer. Hence, following the domain decomposition as presented in the previous chapter, it belongs to the muon system domain. However, from a global viewpoint it doesn’t look any different than any other reconstruction program, independent of the language in which it is written. A detector description, an event structure,
the implementation of a reconstruction algorithm, etc. are all easily visible. The difference with most other programs is that here they are strictly separated, and only see each other through a handful of interface classes. Furthermore, in adherence to good design practices [27], anything that is not specific to the ATLAS muon reconstruction has been split off.

All these observations are especially true of the way the reconstruction algorithm is implemented. To explore the full potential of OO and C++, it has been designed using everything they have to offer. Along the way, the main design goals have been flexibility, extensibility, reusability and robustness in the face of change. Or in other words, the design tries to adhere to the Open Closed Principle, which states that the software must be extensible without requiring change to the existing code [28].

In fact, the ATLAS muon reconstruction algorithm is nothing more than a blueprint, defining how to use the building blocks supplied by two independent, general-purpose packages, viz. the Detector Reconstruction Toolkit or DRT and the Generic Dataview Library or GDL. The DRT defines a diverse set of general reconstruction classes such as error points and cones, tracks and magnetic field implementations. In addition, it performs a number of related tasks such as track fitting and the propagation of tracks through a magnetic field.

The second package, the GDL, is a novel library that incorporates the dataflow principle into an object-oriented design, and provides a framework in which data-driven algorithms can be implemented in a straightforward and intuitive way. It is the core of the reconstruction; controlling it and defining its logic.

A third independent package holds the utilities [29, 30]. It is a library of general-purpose classes that provide support for such diverse things as commands and callbacks, smart pointers, named parameters and a basic Component Object Model (COM). The latter is explained in section 3.3.1, while some of the other classes are mentioned in the parts of this chapter to which they bare relevance.

Many of the paragraphs that follow go into considerable detail. For those of you not interested in this, the beginning of each section gives an overview of the functionality offered by the corresponding package. The subsequent subsections can then be skipped without losing the ability to comprehend the rest of this thesis. Also, some of the more important terms are explained in the glossary (see appendix C).

3.2 AMBER

The ATLAS Muon Barrel and Endcaps Reconstruction program or AMBER performs the stand-alone reconstruction of events in the muon spectrometer. But from the start it has been designed with flexibility in mind. This also makes it a framework, aimed at facilitating the development of reconstruction algorithms and the building of complete programs for the ATLAS muon spectrometer and beyond. Special care has been taken to shield it from the ATLAS specific definitions for the detector description and the event structure, and to decouple the different subpackages from each other as much as possible.

In this section an overview of these subpackages is given, loosely following the steps taken when processing an event, up to the moment when the actual reconstruction starts. That
will be the topic of the next chapter. It is impossible to go into all the details here, and in many cases therefore only the top-level classes are shown.

3.2.1 Integration into Arve

Following the ATLAS strategy at the time, AMBER is integrated into the Arve framework. It uses its control system to steer the processing of events, its geometry and detector description classes to represent the muon spectrometer and to simulate its behaviour, and its graphics and console windows to output the results. In AMBER's main function, the first two lines exist to create and initialize Arve, as well as its core package called Gismo (see listing 3.1), while the other commands deal with the initialization of the main AMBER classes, beginning with System.

```cpp
Arve app(world_size);
new Gismo(world_size, 0, &Arve::instance())->display();

// System setup
amber::System().initialize();
amber::System().reconstructor(new amber::Reconstructor());

// Event sources
amber::EventSimulator* simulator = new amber::EventSimulator(
    new amber::SingleParticleGenerator("mu+", Point(0, 0, 0)));
amber::EventGenerator::instance()->add(simulator);
amber::EventGenerator::instance()->add(new amber::G3EventLoader());
app.run();
```

**Listing 3.1** AMBER initialization (main function).

Within Arve, each subsystem is represented by a Module class whose function it is to construct the system and to control all outside access to it. In the case of the muon system domain, this task is performed by a combination of the System class and its nested class Module (see figure 3.2). The former is a monostate class, which means that all objects share the same state\(^1\). As a consequence, System's constructor can not be used to initialize its state, and the initialize method has to be called instead (line 4 in listing 3.1). Its foremost tasks are to create a new instance of its nested Module class, in order to incorporate AMBER into the Arve framework, to read one or more parameter initialization files, and to construct the detector description.

---

1. The monostate pattern [31] has been selected in favour of the singleton pattern (see section 3.2.2 for a description), because only the former can be used in conjunction with inheritance. This means that specialized system classes can inherit from System.
In addition to the detector tree, System also stores a link to the reconstructor. Any class inheriting from ReconstructorBase and implementing its clear and execute methods will do. This base class, through its nested Viewer and Printer classes, provides the link to Arve's GUI, ensuring that the results of the reconstruction are both displayed on the screen and printed on the console (see section 3.2.4).

Figure 3.2 External interface of AMBER as seen from Arve. For an explanation of the syntax, see appendix A.1.2.

Figure 3.3 Event loading and simulation.
3.2.2 Event

After the creation of the System class, the various event sources are constructed (see figure 3.3). Like the System class, EventGenerator is a descendant of Arve’s Module, and is therefore executed during each pass through Arve’s event loop. The generator is implemented as a singleton [32], which limits the number of objects that can exist at run-time to exactly one. Its instance method grants access to that single object for, among others, the System class: EventGenerator is added to the list of modules that have to be executed first before System can run its reconstructor.

The event generator class stores an arbitrary number of EventSources, which are responsible for the actual generation of the events. Two such sources are implemented by AMBER, viz. the G3EventLoader, which reads in events generated by the ATLAS simulation software based on Geant 3, and EventSimulator, which uses Arve’s internal simulation (the Gismo class).

The EventSimulator in turn contains a DetectorResponseSimulation object that visits the detector hierarchy and updates the results of the simulation. DetectorResponse descendants exist for simulating detector inefficiencies, adding noise and taking into account the finite resolution of the various detectors. The sequence of calls that are executed when the EventSimulator is called upon to generate an event is shown below.

![Sequence diagram](image)  

**Figure 3.4** Sequence diagram showing some of the classes involved in simulating an event. The syntax is explained in appendix A.1.3.
1. First a test is performed to check whether the EventSimulator can generate a new event, i.e. to see whether it is enabled and the number of events has not reached the total amount requested by the user.

2. When a new event is to be created, the call is forwarded to the only instance of Arve’s Gismo class. It generates an event and propagates it through the detector.

3. After the digits have been added to the detector, its response is modified to take into account effects like detector inefficiencies.

4. As DetectorResponseSimulation is a DetectorVisitor, it is passed on to the root of the detector tree (see the visitor pattern in [32]). During its traversal it passes through all detectors.

5. Upon receiving the visitor, each detector calls it back with itself as an argument.

6. From within the visit method, all DetectorResponse objects are executed with only the call to DetectorInefficiency shown here.

7. The do_execute method belongs to the DetectorResponse base class. It checks whether its specific type of response modification is enabled before passing execution on to its descendant.

8. Finally, DetectorInefficiency loops over all digits in the detector and for each one decides based on a random number whether to keep it or not. If a digit is to be deleted, the detector’s remove method is called.

At the end of the event generation the digits are stored in the detector hierarchy, each one in the detector to which it belongs. During high-luminosity running, the number of these digits can become very large and they are therefore designed to be as lightweight as possible. In addition to the detector response information, e.g. the drift distance in the case of the MDT digits, they only store their element number (1...n) and a pointer to the detector to which they belong.

Figure 3.5 MDT digit structure.
3.2. AMBER

(see figure 3.5). All remaining information like their position and dimension is calculated from their containment in that detector. These calculations are performed by the detector element classes, which for the MDTs is the MDTTube.

In the case of simulated data, the truth information is also stored in the digit2. The TruthInformation base class stores general information such as a kine index, while MDTTruth keeps track of the MDT specific data like the real drift distance, the coordinate along the wire and the time-of-flight correction.

Also shown in figure 3.5 is the class Plottable. It is part of the ATLAS graphics design, and identifies Digit as being plottable on a graphics scene. This will be explained in section 3.2.4.

3.2.3 Detector Description

The term detector description is applicable to two different concepts. It can be used to describe the data, i.e. the actual geometrical parameters of the detector. And it can be related to the so-called metadata, i.e. a description of the logical structure of the detector. Within AMBER, the detector description is a combination of both: It is the structure within the program that is built based on the logical description of the detector, but also serves as the front-end to the geometrical properties, granting access to it and at the same time hiding its internal details. In addition, it is also the place in which the events are stored, providing the reconstruction with a uniform view of the data independent of their origin or type.

The full logical structure of the detector, which consists of the sensitive volumes, the dead material and their respective parents has to be present for Arve’s internal simulation. Therefore, AMBER’s detector description is built on top of the structure defined by Arve (see figure 3.6). Because the reconstruction attaches itself to the sensitive leaves of the detector (see section 3.2.3) and as a result only sees the parts of the whole detector hierarchy that it needs to see, there is no need to have two separate logical descriptions, one for the simulation and the other for the reconstruction.

Arve makes the distinction between media, volumes and detectors. Detectors are objects that know how to react to a particle crossing them, but they know nothing about their position or size. That is the task of the volume classes such as Box and Tube. Finally, the media classes complete this picture by combining the other two and by building a detector tree through the application of the composite pattern [32].

To interface to this design, AMBER defines two classes, viz. Medium and Detector3. In addition to their role of shielding the other AMBER classes from the details of Arve, they also store the official name of the medium, respectively the detector [15]. The generic Medium class is used for every part of the detector; the only exception to that rule being Spectrometer, which represents the root of the muon detector hierarchy. It is different, because it is responsible for

---

2. The pointer to the MDTTruth object can be removed with the help of a preprocessor directive, thereby eliminating any unnecessary overhead when running with real data.

3. Because AMBER’s source code resides within the amber namespace, there are no name clashes with the corresponding classes in Arve, or in any other part of the ATLAS software.
the construction of the whole detector tree. As can be determined from figure 3.7, the construction sequence is as follows:

1. Based on the value of a named parameter, Spectrometer creates a parser. The one shown here reads the ATLAS Muon Database or AMDB [33]. It then builds a GeometryDescription object containing the name, coordinate transformation, dimension and internal structure of each item in the detector hierarchy.

2. Spectrometer subsequently passes this description on to a DetectorBuilder, with itself as the parent to which the detector tree must be attached.

The division of this process into two separate steps (the parsing and the building) has been done to keep the impact of changes to the input format to a minimum. Furthermore, by defining a separate DetectorBuilder class instead of giving the medium and detector classes build methods, the design is more flexible because it minimizes the external dependencies of the detector description classes.

4. Named parameters are provided by the utilities package [30]. Based on the name of the parameter, they search for its value in a database.
3.2. AMBER

The final class in the detector description structure is LayerDetector. It represents one layer of detector elements, the type of which depends on the template parameter Technology (MDT, RPC, etc.). They store the digits and serve as the aforementioned entry points the reconstruction can hook onto.

This leaves the only classes in figure 3.6 that have not yet been mentioned, viz. the descriptors. These are interfaces, hiding the implementation details of the geometrical description of the detectors, and thereby making the detector description independent to changes made therein. In the case of MediumDescriptor, the functionality focuses on transformations between the local and global coordinate systems, while for the DetectorDescriptor class template the emphasis lies on the digitization process. The latter class also provides access to a descriptor for each detector element in the layer. The MDTTube, RPCStrip, TGCWire, TGCStrip, CSCWire and CSCStrip classes (their exact type is part of the Technology template parameter) provide information about the position and dimension of the particular element they represent. They are generated on the fly, and are not stored in the detector.

3.2.4 Graphics

We end this tour of AMBER with a short look at how its objects are displayed. The official ATLAS graphics design contains four main interfaces (see figure 3.8). A Plottable is an object that can be displayed on an AbstractScene. For each combination of plottable and scene there exists a PlottableRep class that knows how to display the former on the latter. And lastly, the PlottableModel class is used to glue everything together by creating the correct representation for each plottable/scene pair.

To interface the scenes with the display capabilities of Arve, the ArveGraphicsScene and ArveConsoleScene classes have been written. The latter has two base classes separating it from
AbstractScene: AsciiScene is an interface defining the normal std::ostream operators [19], while AsciiStreamScene stores a pointer to an output stream to which it forwards all messages. ArveConsoleScene then only serves to hide from the user the details of obtaining the output stream corresponding to Arve’s console.

To implement the PlottableModel interface, AMBER defines the GraphicsModel class template. It instantiates either an AsciiRepresentation or ArveRepresentation object, with both being specialized for every class that is plottable.

For example, in the detail of Arve’s event display shown in figure 3.9 a part of a barrel station is depicted. In it, the Arve representations of the plottables RPCdigit and MTDigit as well various reconstruction results are drawn. The dashed lines form the boundaries of the region of activity based on the RPC digits (see also section 4.1), and the line inside of it is the reconstructed track. The MDT hits, i.e. the digits that were found to be part of the track, are plottables as well, so that they are displayed in a different color than the unused digits.

Figure 3.8 Implementation of the ATLAS graphics scheme within AMBER [34].

Figure 3.9 Detail of Arve’s event display.
3.3 Detector Reconstruction Toolkit

The Detector Reconstruction Toolkit or DRT is, as the name suggests, a toolkit of classes that are useful in the reconstruction of physics events. They were thought general enough to be separated off from AMBER. Along with a few classes that deal with geometrical entities such as error points and cones, the bulk of the DRT is related to tracks. Its three major subpackages deal with the track classes themselves, the track fitters (see chapter 4) and the propagation of tracks through a magnetic field.

3.3.1 The Track Package

The track is the most central concept in the reconstruction of a high-energy physics experiment. Consequently, its applications are diverse, and so are the properties and functionalities assigned to it by different programs, or even by different sections within one program. Trying to come up with a single closed design to fit all these cases is doomed from the start. Hence, the first requirement of any track package must be formulated as:

1. **Algorithm independence**
   The track classes must be general enough to be used by all reconstruction packages.

From this requirement alone, it follows that the track package can not consist of an explicit implementation, but instead can only define a framework; an extensible structure on top of which each program can implement its own classes. The following functional requirements only serve to make this framework complete, flexible and internally coherent [35].

2. **Querying a track**
   A track must supply the following information:
   - The track fit parameters, including their errors.
   - The elements associated to a track such as its hits and vertices. A user must be able to supplement this list with additional types of his own.
   - The quality of the track (fit).
   - A user-definable type or status, e.g. to record whether the track came from a reconstruction of the hits in the muon or inner detector, which algorithm was used to find/fit the track, which magnetic field was used in the fit, etc.
   - The truth information for Monte Carlo generated tracks.

3. **Updating a track**
   A user must be able to update any of the fields listed in requirement 2.

4. **Comparing tracks**
   It must be possible to determine whether one track is better than another, with
the user being able to define what “better” means.

5. **Track selection and ordering**
   The track package must supply the architecture for selecting and ordering tracks based on their query and compare methods (see requirements 2 and 4).

6. **Combining tracks**
   It must be possible to combine any two tracks, possibly of different type, with the framework making this as convenient as possible.

Requirements 2 to 5 are addressed in the next two paragraphs in which a first level design is presented. However, because that design does not fulfil requirements 1 and 6, it will be further enhanced in subsequent paragraphs.

**Basic Design**

The basic track package is one that can only be used by a single program. All the required functionality is there, but the contents of all classes and their interdependencies are explicitly defined, and there is very little flexibility.

![Diagram of track classes](image)

**Figure 3.10** Basic design of the track classes.

The Track class itself is basically a container, storing information without providing any real functionality. This is necessary even in a design where all the functionality required of a track is known, because the number of methods of the Track class would otherwise proliferate. To this end, different classes are defined to hold the track parameters, its quality and the truth information. The fourth class, TrackConstituent, is the abstract base class of everything that can be associated with a track. Examples of this are hits, inert material (multiple scattering points) and vertices. As all these constituents are known by the program, the visitor pattern [32]
in the form of the TrackConstituentVisitor interface is an appropriate way to add functionality to them without cluttering up their interfaces.

**Helper Classes**

To allow for the diversity of operations that can and will be done to and with tracks, such as their building, extrapolation and fitting, they are separated off into an unlimited number of independent helper classes. These must of course be defined by the user, but for the selection and ordering of tracks (requirement 5) a template framework comparable to the D0 cuts package [36] can be defined.

**Traits Design**

The problem with the basic design presented above is that it does not satisfy the requirement of algorithm independence. Different programs have to define their own track package, even if they share most of the design. To solve this unnecessary duplication of code, the track package is made user-modifiable by introducing the "traits" [37]. The Traits class is nothing more than a collection of four type definitions, viz.

- **constituent_type**: The base class of the track constituents.
- **identifier_type**: The type by which all track constituents can be uniquely identified.
- **parameter_type**: The parameter set used by the track.
- **quality_type**: The quality (base) class of the track.

The type of the truth information has not been added to this, since it is based on the general Geant 3/4 format. However, if it should be needed in a later stage, it would be a trivial matter to add.

The Track and TrackConstituent classes are now parameterized with the Traits class, resulting in the design presented in class diagram 3.11. Because the Track class has become a template, it is beneficial to have a common base class e.g. as an interface to the outside world. This task is fulfilled by the Trajectory class. Its two methods, which are depicted in diagram 3.11 are related to the propagation of the trajectory through a magnetic field and are discussed in section 3.3.2.

The TrackParameters, TrackConstituent, and TrackQuality classes still exist, but they have now become interfaces from which the user can derive his own, or he can choose to use completely different classes. This second option would not have been possible without the traits. The first one of course already existed in the basic design, but the problem would then have been the absence of direct access to the user-defined descendants outside of the visitor pattern\(^5\).

\(^5\) Except by using dynamic casts.
One other addition is the TrackStatus class. It serves as the base class of an entire hierarchy of user-definable status classes. These classes do not have to have any state, as their functionality can be compared with that of the members of an enumerated type, with the difference that an enumeration can not be extended once it has been defined.

The TrackConstituentVisitor class also still remains, but is now an implementation of the acyclic visitor pattern [38]. Instead of defining just one visitor base class, it defines such a class for each constituent type. This not only makes a non-templated visitor class possible, but more importantly, it eliminates the otherwise necessary dependencies between the different constituent types.

**COM Design**

The Traits make it possible for the Track package to become a general toolkit, to be used by multiple programs. However, a specific Track<Traits> implementation in most cases still only makes sense for one application. If one wants to combine the results of two or more programs, say e.g. the tracks found in the muon spectrometer with those in the inner detector, then one is forced to add conversion constructors or operators in one or both of the classes, or to introduce wrapper classes. This would be a very inflexible approach, making one program dependent on possibly many others.
One way to solve the problem of how to transparently access the information supplied by any number of (unknown) sources is to use a Component Object Model (COM) [39]. It is based on the principle that an object can make its functionality available through a number of interfaces. The calling party then only sees the interface he is interested in, and not the implementation behind it. And what is most important, interfaces can be added and removed without breaking the code that does not use them, and without even having to recompile it.

The central base class of these COM interfaces, as well as of the COM-enabled classes themselves, is the IUnknown class (see figure 3.12). It defines a number of query_interface methods, which return the requested interface when either the object itself, or the object that is hidden behind the interface supports it. The simplest way to turn a class into a COM-enabled one is to inherit it from the COMObject base class, with Object the type of the class that is to become COM-enabled. It manages the list of interfaces that its descendant Object implements. The second main class in diagram 3.12 is the COMImplementation template, which takes care of the administrative tasks required of an implementation of a COM interface.

![Diagram 3.12 Implementation of the COM model as it is used by the track package. Note that a class name in italics doesn't embody a real class, but a type of class instead. In the DRT for example, Object can represent Track or TrackConstituent.](image)

To add the Component Object Model to the track package, both Track and TrackConstituent are turned into COM-enabled classes by inheriting from COMObject. In the case of the Track class this is done to meet requirement 6 (combining tracks). The TrackConstituent class has been changed to support COM in order to have another way in

---

6. COM has been developed by Microsoft, and just as other similar solutions like CORBA, it is in its full form far too bulky for such a simple thing as a track package. Hence, the COM model used here is a simplified version, implemented by the utilities package.
addition to the (acyclic) visitor pattern to add functionality to its descendants. For an example of its use, see the track fits in sections 4.2.3 and 4.4.

### 3.3.2 Track Propagation in a Magnetic Field

Being able to define tracks is one thing, but they are pretty useless without their accompanying helper classes. One very important helper package is the propagation of a track through a magnetic field. The requirements on such a package are fairly straightforward [40]:

1. **Field value**
   The package shall describe the magnetic field anywhere in the detector.

2. **Field gradient**
   The package shall provide the gradient of the magnetic field at all places where it is able to provide a field value.

3. **Tracking by step**
   The package shall be able to extrapolate a track including its error along a given distance taking into account the effect of the field, while ignoring physics effects such as multiple scattering, energy loss and particle decay.

4. **Tracking to a surface or volume**
   The package shall be able to extrapolate a track to its intersection point with a surface or volume.

The first two requirements are easily fulfilled by the class hierarchy topped by the MagneticField interface (see figure 3.13). It declares query methods for the value and gradient of the magnetic field at any point in space. Two descendants, one for a constant field and the other for an ASCII-based field map, have currently been implemented by the DRT. The actual propagation of a track is built around the MagneticFieldTracker class. It is a static class, i.e. no instances can be created and all clients see the same static state. It performs no real work, but is merely an engine executing the appropriate, user-definable classes around it. It does this in a three-step process:

1. First, it selects the step size based on the maximum allowed error per step and the gradient of the magnetic field.
2. Then it approximates the track and its errors by a local helix as defined by the LocalTrackParameters class.
3. Finally, it extrapolates the helix over a distance equal to the chosen step size (SimpleStepTracking, a descendant of TrackingAlgorithm), or using the Runge-Kutta algorithm. This RungeKuttaTracking class is used by default. In addition, the transport matrix of the local helix errors is calculated for the current step, and is added to a running aggregate maintained by the tracker.
These steps are repeated until the end of the propagation as defined by requirement 3 or 4 is reached, at which time MagneticFieldTracker updates the original parameters and their errors.

The last track-propagation requirement is satisfied by the Interceptor classes, which calculate the intersection of a track with the surface or volume defined by that interceptor. An interceptor template following the Template Method pattern [32], as well as implementations to work with the Surface and Volume classes of Arve have been defined. An example use of an interceptor is described in sequence diagram 3.14:

1. When a user wants to propagate a track to the surface of a cylinder, he creates a SurfaceInterceptor with the cylinder as an argument.
2. Then the interceptor is executed with the track as an argument.
3. The interceptor queries the cylinder to determine the position of the track relative to the location of the cylinder.
4. As long as the track has not intersected with the cylinder (the sign of its relative position has not changed), the track is propagated by the default step size.
5. When the track enters or leaves the cylinder, the propagation is reversed with a step size equal to half the distance to the surface as returned by the latter's how_nerar method. This process is continued until a certain accuracy has been
achieved. Finally, track is updated to reflect the intersection point.

Instead of calling the execute method of the interceptor directly, this calculation can also be started by invoking the MagneticFieldTracker's `propagate_to` function. This duplication merely exists to complete the tracker's interface.

### 3.4 Generic Dataview Library

Reconstruction programs like most other software that is algorithm based, are to a large extent dataflow oriented: Starting with a certain set of data, a number of successive operations are performed to reach the sought-after results. The way these problems are generally solved is to build lists of objects, and then to write the functions that operate on them and create new lists. This decoupling of containers that store the data, and algorithms that work on it, is also present in the Standard Template Library (STL) [41] and is called generic programming\(^7\).

The containers and algorithms work together through so-called iterators. An iterator is an object that refers to a specific value within a container, and each container must supply two such iterators, one to its first value and the other to its end. Iterators come in five different flavours (input, output, forward, bidirectional and random-access [41]), each one with its own

---

\(^7\) The STL is part of the C++ standard library and it implements generic programming through the use of templates.
well-defined functionality. The algorithms base themselves on this functionality, and have therefore no need to know anything about the underlying container.

The Generic Dataview Library or GDL uses these iterator specifications to define its dataviews, which are basically iterators that adapt other iterators. Consider for example the simple algorithm shown in figure 3.15. It depicts the creation of track segments out of two hits, one from each of the detectors. These detectors are the real containers. They store the hits and supply the required iterators. The Combinatorials dataview that follows them is an iterator adaptor. Internally it stores two iterators, one to the current value of each detector. And its corresponding value is the pair created out of these current hits. Similarly, Segment Builder is an adaptor with one input. It transforms that input value, i.e. the pair of hits, and creates a track segment out of it. So in effect, a dataflow network as shown in figure 3.15 is nothing more than a chain of iterator adaptors linked together. Each of the adaptors represents another view on the data, hence the name dataview.

**Figure 3.15** Example of a small GDL network. For an explanation, see appendix A.2.

A dataview is in many respects identical to a component, in that it completely decouples its interface from its implementation. The former is a combination of the iterator type it belongs to and the type of its output. The latter is the whole upstream network, i.e. its inputs. It can consist of only a single or more than a thousand dataviews, but the behaviour of the dataview remains the same. This means that a dataview completely encapsulates its upstream network.

An important feature of the dataviews is that they are of the data-pulling type. This means that it is only on the request of the user that something happens. For example, calling operator++ on the Segment Builder of figure 3.15 results in a call to Combinatorials to look for the next pair of hits. It does this by advancing its internal iterator to Detector2 by one, or when that iterator is at the end of the detector, to reset it to the first value and to advance the iterator to Detector1 by one. This is contrary to the data-pushing approach in which every time a value changes, a number of registered functions are called. This would mean that the data and not the user is in control. In this scenario, whenever a hit is added to one of the detectors, Combinatorials would be called automatically. As can already be seen from this simple example, this would lead to a much more complicated programming logic.

Another consequence of this feature is that the dataview network is based on lazy evaluation. In the example above, the track segments are only built on request. When the querying of Segment Builder stops after the first segment, the others are never calculated. This is a major advantage over the list-filling approach in which an operation is applied to a whole list of values before the next operation is performed.
In a dataview network, the copying of data is reduced to a minimum. All dataview values are passed on through the network as references. And when a new object is to be created, it is stored in a reference-counted pointer. This has the added benefit that the object is automatically deleted when it is no longer needed.

### 3.4.1 Core Implementation

The standard way to define an iterator adaptor is to parameterize it with the type of the iterator it connects to. Although this would work fine in small programs, it doesn’t scale very well. The reason for this is that almost every dataview would be a separate instantiation of the adaptor template, with the template argument containing the whole upstream network. As a result, the compile time and program size would increase with every new dataview that is used. So instead, the dataviews form a class hierarchy with common base classes at the top (see figure 3.16).

![Dataview classes](image)

**Figure 3.16** Dataview classes.
dataviews can then refer to these base classes, and don’t have to know the exact type of the connections.

The type-independent behaviour of the dataviews is defined by the DataViewBase class. It stores a name and type, both of which can be removed from the program by setting a preprocessor directive, as well as a list of clients. These NotificationInterface descendants are notified when the dataview becomes invalid or is deleted. A large part of this list is formed by the downstream dataviews that adapt it, i.e. the dataviews that are connected to its output.

The most important methods of DataViewBase are listed in figure 3.16. Through the reset method it resets itself and the upstream dataviews to an empty state. For example, in the case of a container the method deletes its contents. The execute method can be used to execute a user-defined command: DataViewCommand is a typedef for the CommandFunction1R class template from the utilities package, taking a DataViewBase as its argument and returning a boolean value. The command is passed up through the network until a dataview is found that can handle it, or until the end of the chain is reached.

The remaining methods of DataViewBase serve to set the dataview to its begin or end state, or to test whether it is in one of those two states. In a normal STL application, the begin and end iterators are created by the corresponding methods of a container. But in the GDL the containers are hidden behind an unknown number of dataviews, and hence the dataviews must define this functionality themselves.

The second base class of the dataview hierarchy is the DataView class itself. It is parameterized with an iterator tag (forward, bidirectional or random-access functionality) and a value type, and is specialized on the former. As can be deduced from figure 3.16, the three dataview specializations inherit from each other so that e.g. a random-access dataview can be interpreted as a bidirectional one. The inheritance relationship is moreover inclusive to increase the flexibility when implementing specific dataviews (see also the next section).

All operators of the DataView class are abstract as they are to be filled in by the specific implementations. This causes some performance degradation but as explained above, this can not be avoided. Also, only the pre-increment and decrement operators are supported. The post-increment and decrement operators require the creation of a copy of the dataview, and hence of the whole upstream network, and that is an operation that could be very costly.

Another consequence of using dataview base classes is that internally pointers are used everywhere. To shield the user from this, dataviews can be wrapped inside DataViewReference objects. Like the DataView class it is specialized for the different tags, and the common behaviour has been factored out into a base class (DataViewReferenceBase in this case) to prevent code duplication.

As a dataview is in most cases an iterator adaptor, it must have one or more connections to other dataviews; its upstream network. It acquires the functionality for storing those connections by inheriting from a connection-type specific base class (see figure 3.17). Inheritance has been chosen in favour of aggregation, because it requires the least amount of effort on the part of the writers of the dataview implementations: They do not have to define any methods to access the connections, and don’t have to be concerned with maintaining their state.

All connection base classes inherit virtually from DataViewBase, thereby granting them access to the state of the dataview. They are also all parameterized with a generic argument for
the type of the connection. Within the GDL, the connection is always of type DataView, i.e. Connection<DataView<Tag, T>> objects are stored by the SingleConnection, DualConnection and ConnectionList base classes. But by leaving the type generic, the door is left open for a direct link to a dataview implementation, i.e. a class inheriting from DataView.

As a final remark, note that the connection base classes own their connections. This means that whenever a dataview is deleted, so is its upstream network. The only exception to this rule is when one of the connected dataviews is shared, i.e. when there are multiple connections pointing to it.

### 3.4.2 Toolkit

In addition to the core library, the GDL also comes with a toolkit of standard dataviews. They are used throughout the reconstruction software described in the next chapter, and a short overview is therefore appropriate.

---

8. By using the type of the dataview implementation directly, the resulting program will once again suffer from code bloat, i.e. ever increasing compile times and file sizes. However, when speed is of the essence this might not be deemed to be a problem.
Containers

Containers are used to store items either permanently (PersistentContainer) or temporarily (Container). In fact they are not real containers but instead dataviews, i.e. iterators to the values of the containers. The actual container is hidden behind a ContainerStubInterface pointer (see figure 3.18). This allows PersistentContainer to use different container implementations. For example a LayerDetectorStub, a descendant of ContainerStubInterface, exists to interface to AMBER’s detectors (see also sections 3.2.3, 4.1 and 4.2).

![Figure 3.18 Container classes.](image)

The second advantage of this separation between dataview and container is that it permits multiple PersistentContainer dataviews to share the same underlying container, thereby preventing the unnecessary duplication of its contents. The default implementation is formed by the ContainerImp class in conjunction with its ContainerStub. As can be determined from figure 3.18, ContainerImp knows about multiple ContainerStubs, which it all notifies when one of them changes the contents of the container. However, the ContainerStubs are the ones that own the ContainerImp object, and not vice versa, and when the last stub is deleted, it takes the container with it.

To come back to the dataviews, PersistentContainer is a random-access dataview without any connections, and whose contents is not affected by the reset method. Instead it defines push_back, erase and clear methods to manually alter the data it contains. Next, the Container dataview combines the functionality of PersistentContainer with a connection to another dataview. When queried for the first time, it loops over all the values of that connection and stores them in the container. This is useful when one wants to save intermediate results that are too expensive to be recalculated.
Transformers

The Transformer<Tag, From, To> dataview transforms the values of a connected dataview into new values of the type To, and has a Tag that is identical to the one of the connection. Like most other dataviews the real work is done by a derived implementation class called TransformerImp. It is parameterized with the type of the transformation to perform, so that any function class defining the appropriate function operator

To operator() (const From& arg) const

can be used. This separation leads to a friendlier interface for the user.

![Diagram of Transformer classes.]

In this particular case, the increment and decrement operators are implemented by DefaultImplementation, which simply increments, respectively decrements the connected dataview as stored in its SingleConnection base class.

To complete this picture, the function

```cpp
template <class Source, class Operation>
TransformerImp<typename Source::tag, Operation>*
transform(const std::string& name, Source* source, const Operation& op)
```

is provided to easily create a transformer dataview.

In addition to Transformer, the GDL also defines a BinaryTransformer class. It has two connections, the second of which is passive. This means that its state is not changed by the binary transformer, and only its current value is used as a second argument to the transformation operation. For the remaining part, Transformer and BinaryTransformer are identical.
Filters

The `Filter<Tag, T>` dataview filters the values of a connected dataview and lets through only those for which a user-defined predicate evaluates to true. The iterator tag of the filter is equal to the minimum of the tag of the source dataview and bidirectional, because it is not possible to implement the step operators, `operator+= (n)` and `operator-= (n)`, more efficiently than by calling `operator++`, respectively `operator--` n times.

`SortedFilter` is similar to `Filter`, but it only works on a random-access input whose values are sorted. By supplying two predicates, which define the lower and upper bound of the valid range of input values, the connection can be binary searched, increasing the speed of the program. This range is determined the first time the dataview is queried, after which the dataview has a random-access functionality.

Finally, `BinaryFilter` is to `Filter` what `BinaryTransformer` is to `Transformer`: It has a second, passive connection whose value it also passes on to the filter predicate.

ContainerModifiers

A `ContainerModifier` dataview can only connect to a `Container`, and is capable of updating the latter’s contents as a whole, and not just one value at a time like e.g. the `Transformer` does. Its descendant `ContainerModifierImp` is parameterized with a unary function that must take a `Container` as its argument. When the dataview is queried for the first time, this function is applied to the container. One such operation, viz. `Sorter`, is supplied by the GDL and it sorts the container’s contents.

When a `ContainerModifier` is connected to a dataview that is neither a `Container` nor another `ContainerModifier`, an intermediate `Container` dataview is created on the fly.

Wrapper

The `Wrapper` dataview creates for every value of its connection a new object that wraps that value. The Wrapper is automatically followed by a `Container` to store the wrapper objects as their state would otherwise be lost when the program continues with the next value. One wrapper object supplied by the GDL is `Used`, which adds a “used”-flag to the original object.

To complement the Wrapper, the `unwrap` function is provided, which creates a `Transformer` that returns the original, wrapped object.

Combinatorials

Two dataviews exist to create the combinatorials of the values of two connections. `Combinatorials` builds and returns all possible combinations of the values, while its counterpart `SortedCombinatorials` employs a selection criterion. It requires that the second connection is of the random-access type and that it is sorted for each value of the first connection. Its descendant `SortedCombinatorialsImp<Tag, Low, High>` is then able to perform a binary search with the help of the `Low` and `High` predicates (cf. `SortedFilter`).
The output of the combinatorials dataviews is a pair of reference-counted pointers to the current values of the two inputs.

**Merger**

The Merger adapts multiple dataviews, and dynamically merges their values into a single stream. The tag and value type of the first connected dataview determine the type of the Merger, and all subsequent connections must provide at least the same functionality as that first one. Connections can moreover be added and removed on the fly.

When using some of the dataviews presented above, the example network shown in figure 3.15 can be coded as follows\(^9\):

```cpp
gd1::DataView<gd1::random_access, TrackSegment>* algorithm;
algorithm = gd1::transform("Builder",
    gd1::combinatorials("Combine", Detector1,
        Detector2),
    build());
```

**Listing 3.2** Program to build the example network of figure 3.15.

The only function that is to be supplied by the user is `build`, which must define the algorithm to turn two hits into a track segment. When the hits in `Detector2` are sorted, a `SortedCombinatorials` dataview can be used instead of the `Combinatorials`, which would speed things up considerably when the number of hits in the second detector is large.

### 3.5 Conclusion

The pursuit of the Open Closed Principle has led to an ensemble of software packages that are far more general than the original task for which they were developed, i.e. that of muon reconstruction in the ATLAS detector. This makes it possible for e.g. the classes of the Detector Reconstruction Toolkit to be adopted by the rest of the ATLAS software community. Especially the track package has been found to be flexible enough for most people to be comfortable with it. In addition, the classes responsible for the propagation of tracks in a magnetic field are being evaluated by ATLAS. They are somewhat slower than the highly optimized Fortran version, but improvements are still possible. Independent of this, the track-propagation package has also been successfully ported to the software of the D0 experiment, requiring only minimal changes that have to do with their different `Point` and `Vector` classes.

\(^9\) All GDL classes and functions reside in the `gd1` namespace.
The dataflow networks of the Generic Dataview Library are so general that they go beyond the realm of physics. In fact, similar principles are found in commercial packages like Open Inventor, but these are in most cases not object oriented but instead of a procedural nature. As more and more people in the ATLAS collaboration become better acquainted with C++, it is hoped that more complicated looking software like the GDL will be more widely used. Because its principles correspond so well with the nature of reconstruction algorithms, it presents a really intuitive way of programming them.

Of course all packages are already incorporated into the ATLAS software as part of the AMBER program, which is the main component of the muon system domain. Because the official architecture is in a constant state of flux, AMBER itself will require continuous updating. For example, the Arve framework will need to be abandoned in favour of Paso [42], and with it the detector description, event structure and graphics will have to be changed. This will be a non-trivial task, but fortunately AMBER is layered such that these changes will impact only small sections of the program. Also, it is hoped that by making this step AMBER will be able to directly access the GEANT-simulated events [43, 44], thereby making a direct comparison with other programs possible.

As a final note, all of this flexibility must of course come at a price. As was already mentioned for the track-propagation package, this price is a decrease in program speed, caused by the requisite abstract (virtual) functions present in the various interface classes. A cost, which is small and which in our opinion is far outweighed by its benefits.