Final Report of the ATLAS Reconstruction Task Force

Véronique Boisvert, Paolo Calafiura, Simon George (chair), Giacomo Polesello, Srini Rajagopalan, David Rousseau

Public release, 22 September 2003

Contents

1 Introduction

The reconstruction task force (RTF) has been charged with performing a top-down design iteration on the ATLAS reconstruction software, considering in particular the granularity of algorithms and event data model, and requirements coming from both offline and high level trigger. One important aim is to find where possible common solutions to common problems across LVL2, EF and offline reconstruction. The list of tasks derived from this brief and from consultations with the software, physics and high level trigger communities is given in the appendix.

This is the final report of the RTF and marks the conclusion of its work. It follows on from two interim reports. Most sections are unchanged from the second interim report. Sections which differ significantly are: Event Data Model, recommendations for common interfaces, section 3.2; Navigation, section 5; Steering, section 6.

The RTF was unable to complete work in two areas in time for this report. Although much progress has been made on composite navigation and seeding mechansims for steering, prototyping in these areas raised some problems which have yet tobe completely solved. In this situation it does not make sense to describe a design that is not yet fully worked out or to make detailed recommendations. It is hoped that this work will be continued to completion outside the RTF and that design documents on these topics will be forthcoming. Meanwhile the navigation and steering sections contain plenty of other recommendations which should be useful until the outstanding work is completed.

This report is aimed at developers with some experience of C++ and OO A&D but who are not expert software engineers. Hence some sections are rather technical but they do still take time to explain some fairly well known design patterns. The data flow section has intentionally been kept less technical as it addresses the reconstruction design at a more abstract level and applies some of the recommendations from other sections, which should be understandable with less programming experience.

The report begins with a discussion of the general question of modularity: what does it mean for ATLAS reconstruction software, what are the use cases for modularity, what granularity is needed, what constraints does this place on the design of algorithms. The section concludes with some recommendations.

The next section of the report tackles some design issues for the event data model. It is there to provide background on some designs, which are used in the dataflow section that follows. First is the event data model classification into subdomains. Definitions of the subdomains are provided as they are used throughout the document. There is a detailed discussion on the subject of common interfaces, and the choice of design patterns to achieve this. Measurement units and reference frames are discussed in the context of CLHEP. The separation of event and non-event data addresses issues that were raised both in the RTF open meeting feedback and from performance studies of the current EDM.

The main area covered in this document is the high level design of the event data model and dataflow. This is where the main brief of the RTF is addressed. The event data model is broken down into domains: subsystem reconstruction, combined reconstruction and preparation for analysis. The high-level view of the collaboration between these domains is shown. Each of the domains is then described in some detail, with diagrams to show the event data model and the algorithmic modules which link the event data together. The event data and algorithms modules are described at a functional level. This is the result of several iterations including discussions with experts outside the RTF. Significant emphasis has been placed on finding common solutions across the whole EDM, factoring out common tools and getting the right level of modularity. An additional subsection discusses integration of the fast and full reconstruction: what is the motivation for this and at what stage in the dataflow common interfaces should be introduced. It also recommends ways maximise the use of tools for both fast and full simulation.

There is a section dedicated to navigation, wherein the different types of association are identified and design solutions are discussed with some examples.

Finally there is a section on steering. This makes some basic recommendations about structuring the reconstruction software and highlights issues to be addressed in order to facilitate re-use of algorithmic code between offline and HLT. There is also a detailed analysis of the problems faced in building a network of algorithms, a discussion of a solution based on conventions for specifying their input/output data keys

which has implications for the EDM infrastructure, and concludes with some practical recommendations.

The complete task list in the appendix shows the priorities that were placed on different topics when the RTF began its work.

2 Modularity

This section explores the motivations for modularity and the idea of baseline reconstruction, and draws some requirements from this.

Modularity is a term used frequently to describe a property that is desired of the reconstruction software, but what does it mean? A "module" can be defined in general as a standard unit that performs some well-defined task and adheres to a specified interface. The way that the task is performed is hidden such that it can be changed without affecting other modules. Another way to see this is that alternative modules can be made, which perform the same task in different ways, and it would be possible to switch between them without affecting the rest of the system.

For the ATLAS reconstruction, identifying the tasks and defining the interfaces will allow the software to become truly modular. The Athena framework provides basic interfaces (Algorithms and AlgTools) for modules, which allow them to be loaded, instantiated and configured on demand at run-time. Many services are provided within the Athena framework to factor out common functions (which are not part of the reconstruction data flow) and perform them in a uniform way. Building on top of this, the definition of the ATLAS event data model and the reconstruction data flow itself are the remaining essential ingredients to achieving real modularity in the reconstruction.

Modularity can exist at different levels. The reconstruction is already modular if for example, one can choose which version of the complete muon reconstruction to run, independently of the downstream combined reconstruction. But this is a course-grained modularity. Fine granularity allows choice between individual tasks of the reconstruction. This implies defining the interfaces (event data model) and tasks (reconstruction modules) in quite some detail.

The requirement to be able to define a "baseline" reconstruction is one of the major motivations for fine-grained modularity. The idea of a baseline is that one can choose a set of algorithms to use for a certain production purpose, for example a data challenge, prompt reconstruction run, reprocessing pass, calibration job, reconstruction from fast simulation. Normally one would like to choose a single algorithm for each specific task, e.g. track refitting or unpacking clusters from raw data - this is what is meant by a baseline.

Why only one algorithm?

- It avoids an explosion of permutations for the combined reconstruction, when there are competing multiple algorithms for each task that creates reconstructed objects.
- It minimises the amount of processing time by avoiding reconstructing the same information in many ways.
- It keeps the reconstructed data output to a manageable size.
- It simplifies the EDM by not having to save information returned by each competing algorithm.

However, it is recognised that there is the opposite requirement for development and testing, to be able to run several algorithms that do the same thing and compare them.

Fine granularity gives the user more flexibility in his or her choice of the optimum reconstruction for some purpose. Say for example that there are two packages that perform reconstruction in the same subdetectors. If the reconstruction granularity is "coarse" then one might have to choose between these two packages, which may both have good and bad features. The ideal situation is to have sufficiently fine granularity that one can choose the best components of each package. This choice may differ depending on the application, type of physics, etc.

Another example where fine granularity helps is when different algorithms are found to be optimal for different physics. For example you might prefer TrackFitterA (better impact parameter resolution) for b-jets and TrackFitterB (better brem-fit) for electrons. Rather than running both packages in full, a finer-grained reconstruction would make it possible to run only the part(s) of both packages (the track fitting code in this instance) which make the crucial difference. For the non-critical parts, one baseline package could be run. When combined with a seeding, one could even choose to run only the part appropriate to the type of seed, thus avoiding duplication completely.

It is important for modularity that algorithms are not written with any assumptions about the number and origin of downstream objects. When several equivalent algorithms are being run for the purposes of comparison, downstream algorithms should not need to be adapted to take account of this. The "steering" level of the reconstruction should handle the combinatorics of multiple algorithms, rather than the algorithms themselves. In the case of tools, the choice of tool is configurable for each algorithm, and multiple instances of an algorithm configured with different tools can be set up and run if required.

For example, two track fitting algorithms must input the same type of cluster objects and must output the same 'Track' object. This will allow the downstream algorithm, such as egammaRec to toggle between the two 'Track' outputs with ease. Furthermore, the downstream algorithm or EDM must not be extended to handle the multiple 'Track' fitters internally, but must be run twice - once configured with 'Track' from one algorithm and once with the other.

The precise definition of the reconstruction baseline for a given run is beyond the scope of this document. It must presumably be decided by comparing algorithms in their ability to do a certain physics task effectively, taking in to account the constraints of the available computing resources.

2.1 Recommendations on modularity

- ATLAS should recognise the concept of baseline reconstruction. The reconstruction software must accommodate the opposite use case of running several equivalent algorithms side-by-side for comparison.
- The reconstruction needs a high level "steering" to take care of the combinatorics introduced by multiple algorithms performing the same task upstream. Algorithms must be written so they are independent of what has been done upstream, so they can be called multiple times with their input configured by the steering level.
- The granularity of the software modules should be sufficient to meet the example use cases and requirements discussed above.
- Design must proceed at subsystem level to define each module of the reconstruction software and produce detailed specification of the function and interface of each module. This report will go on to identify the main modules and their interactions, but the detailed design will follow on afterwards.

3 Event Data model design

3.1 Classification of the Event Data Model

It has been found to be convenient to classify the EDM into the following domains.

- **Raw data model**
  This domain covers the Event Data Model up to the output of the RODs (RDOs) and includes data produced by the atlas simulation. The Raw Data Flow is documented in Reference [7].
- **Subsystem reconstruction data model**
  This covers the Event Data produced by individual sub-systems: inner-detector, calorimeter and

muon systems. While these sub-systems may share common interfaces or concrete classes, the content of the output data is a result of processing information within each of these sub-systems.

- **Combined reconstruction data model**
  This includes Event Data produced as a result of combining information across the three primary sub-systems as described above. The output is typically the first candidate objects such as egamma, jets, vertex, MissingET etc.

- **Analysis data model**
  This includes Event Data produced as a result of a identifying the candidate objects using analysis specific algorithms. The Event Data produced at this stage includes identified electrons, photons, b-jets and such.

- **Truth data model**
  This includes the EDM necessary to define the simulated truth information. This has not yet been considered in this document.

These domains do not map directly on to the content of the persistent event data at different levels of summary, because this can vary with time. Instead, the domains are meant to help give an overview of the data flow and the way the reconstruction is organised.The domains are not intended to constrain the data available to users and developers. For example, it is perfectly reasonable for an algorithm working mainly in the analysis domain to navigate back to data in the subsystem reconstruction domain.

3.2 Design patterns to achieve common interfaces

This section concerns the use of common interfaces in the event data model. The situations are identified in which common interfaces would be useful. There is some discussion of the issues, weighing up the pros and cons of different solutions, and some initial recommendations are made.

Common interfaces can serve several purposes in the event data model. The first is to give a common "look and feel" to the EDM classes. The second role of common interfaces is to provide polymorphism where there is a use case for this. A third issue is how to provide classes for different providers of the data (e.g. fast and full simulation, LVL2 trigger) which do not provide the same information, while giving a single way to access to the basic information that is common to all providers.

The solutions to these problems will be familiar to experienced software engineers. They are explained a little in the following paragraphs to introduce them to ATLAS software developers who may not be aware of all the possibilities.

The requirement of common classes and extended interfaces may be achieved in several ways. One possibility is to use inheritance to define an abstract base class with the common functionality (e.g. 4-momentum) and then derive separate classes which need to extend this in orthogonal ways, such as classes produced by the full reconstruction and fast simulation. Users who want the possibility to migrate between fast simulation and full reconstruction without changing their algorithm would have to write their code using the base class[1]. Only experience will show how typical this scenario is.

A second possibility is the use of templated classes. This has its usefulness in some applications, especially when there are a limited number of differences between the various instantiated types. For example, an object of class "A" can be made from objects of either class "X", "Y", or "Z". Object A must hold a navigable link to its parent (i.e. X, Y or Z) but in all other respects, it presents the same interface independent of its parent. In such cases A can be templated (A<T>) where T is defined at run-time to be X, Y or Z. Internally A would hold a pointer (or a DataLink) to X, Y or Z. Note that such templated classes can be sub-classed from if necessary, but the use of the templates prevents sub-classing when the only need to do so is to establish navigation to the parent.

[1] It should be noted that StoreGate allows recording of an object by its base type and retrieving it by its concrete type through the use of *symLinks*. It also allows recording a `vector<base>` types and retrieving it as a `vector<concrete>` types. This prevents the client from down-casting, but it is just hidden within StoreGate so type safety remains along with the inherent performance penatly

Templates can be used in another way: one can write templated helper functions (like STL algorithms), which allow additional functionality to be provided for classes without complicating to their own interfaces. For example, if several classes all have a standard method to provide a 3-momentum, a templated function can be written to provide a 4-momentum for any of these classes, and a given mass hypothesis.

Another possibility is to use the adaptor pattern [2]. A common base class is provided as above, but the sub class is an adaptor, which presents the extended interface to the user but implements this by forwarding requests to the real object produced by the reconstruction. It contains a pointer to this object. This has the advantage that classes already existing don't need to be modified; as such it might be a good interim approach to allow classes to work together until a common design is achieved. Users can move to using the adaptors as part of the migration path. This has the disadvantage that it does not in itself encourage movement towards a common interface like the top-down imposition of a common base class would, and it requires additional maintenance work, since each class now has an accompanying adaptor which must be kept up to date with it.

It is impossible to recommend one option as a generic solution for all use cases. Performance issues need to be studied for these approaches, but unless there is a severe performance penalty, design considerations should be paramount. There was a study some time ago of the time overheads of virtual inheritance [1]. A first look at this with current compiler and platform shows that the virtual table lookup overheads should be considerably smaller than the computations typically performed by these methods in the combined reconstruction domain[2]. Anyway, it should be noted that the classes which are candidates for the common interface solution are mainly in the combined reconstruction and analysis domains which are not in a time-critical part of the system. Even where there may be a slight deterioration in performance, its added value against such costs need to be studied. The RTF has studied one specific use-case: a common approach for reconstruction data classes to provide their 4-momentum, and has made a recommendation (see section 3.2.1).

Functional requirements are the most important aspect of the software to get right in the first design iteration. Quantitative time constraints should be included in the functional requirements. During the first iteration the primary consideration should be to avoid design mistakes that cripple the timing performance rather than achieving the best possible performance right away. A cyclic approach should then be taken of identifying bottlenecks and making incremental speed improvements, until the required performance is reached. Note that much of the reconstruction software is already beyond the initial design iteration and is rightly addressing performance now. Along with the RTF recommendations, experience regarding the performance should be taken into account in the next iteration of the development cycle.

3.2.1 Four-momentum interface

Some of the existing calorimeter clustering algorithms, such as KT, works solely in the 4-momentum space. Their input in principle can be CaloCells, CaloClusters, Tracks, Truth particles or any object that can provide a Hep4Momentum. The clustering algorithm itself is generic working in 4-momentum space and not knowing the identity of the parent particle. This is an example for a need to develop reusable generic algorithms in terms of four momentums. There are also additional requirements, particularily arising from the combined reconstruction and analysis domains, that require the need to calculate the four momentum from many objects. It is highly desirable to enforce a uniform standard convention for data classes to specify its four-momentum and related kinematical quantities. This will allow development of generic algorithms and analysis algorithms can expect a uniform data model convention in this regard.

There are a few approaches to handle this generality, which the RTF have considered. We describe some of these approaches and our final recommendations. Two possible approaches are:

a) Write an algorithm that converts CaloCell into an intermediate object that defines the 4-momentum interface. The copy algorithm must be written for each type of object to be converted.

[2]On a 1 GHz machine (lxplus) with code in Athena, compiled with standard ATLAS optimised flags, it was seen that calling a method via virtual inheritance costs 15 ns. This can be contrasted with: creation/deletion of an object (4 double data members) on the stack takes 220 ns; new/delete of an object on the heap takes 480 ns; computation of pt from px,py takes 80 ns; computation of pseudorapidity from px py takes 330 ns.

b) Each of these objects adhere to a common IHep4Momentum interface.

There are variations to these two basic approaches and they both come with pro's and con's.

The first approach requires an additional step that not only results in a large number of additional objects in transient (and possibly persistent) memory, but also additional computational time. It however has the advantage that it provides a clean decoupling and does not impose constraints on data classes.

The second approach clearly avoids the additional step, and hence the additional number of objects appearing in memory. It also helps in writing additional generic algorithms that are common amongst many objects. An example of this is a helper class to calculate ΔR between two objects. Rather than writing these helper classes for every type of object, a single helper class may be written with an interface:

```
double deltaR::calc(IHep4Momentum& mom1, IHep4Momentum& mom2)
```

This promotes code re-usability and avoids developers making common mistakes such as forgetting to take phi-wrapping into account. However, the second step also comes with a weakness. All the necessary information required to calculate the 4-momentum may not be present. For example CaloCell has to make the assumption of a massless particle originating from the vertex. As the reconstruction proceeds, additional information such as the vertex position and particle type become known, hence it is much more sensible for a final electron or muon object to provide a 4-momentum. It is therefore difficult to retain the same assumptions for all reconstruction objects.

Some of the questions that the RTF have considered before arriving at their final recommendation include:

- Who are the clients of the 4-momentum of these early reconstruction objects. Why do they need the 4-momentum?
- Would they calculate the 4-momentum anyway, even if a common interface is not provided (such as will be done in clustering algorithms). In such cases, how hard is it to document the assumptions made in an algorithm compared to the assumption made by the data classes themselves.
- What is the impact on performance and memory when sub-classing from a common interface.
- Is this a scalable solution? Will there be a demand for extending the interface beyond the need for a 4-momentum?

Note that external classes such as HepMC cannot be changed to inherit from a standard ATLAS base class. In this case, discussion with the authors is advocated to agree to a standard base class; meanwhile adaptors can be used as an interim solution.

The RTF also prototyped both possibilities to gain better insight into the pro's and con's of the two approaches described above. Based on these studies, the RTF recommends that reconstruction data objects that are accessed by downstream algorithms sub-class from the IHep4Momentum interface as described below. These include CaloCells, EnergyClusters, output Track classes and the combined reconstruction EDM. The ambiguity in assumptions that must be made to implement the IHep4Momentum interface is partly resolved by clear established policies and partly by clients making use of default concrete 4-momentum classes provided by the architecture group (see next paragraph). The sub-classing coherently ties the components of the sub-system and combined reconstruction.

A design for the IHep4Momentum classes is proposed (see Figure 1) with the following requirements in mind:

- enforce a uniform access to kinematic quantities across reconstruction output objects;
- do not add any data members to the objects;
- avoid unnecessary conversion (for example pseudorapidity to 3-momentum to pseudorapidity);

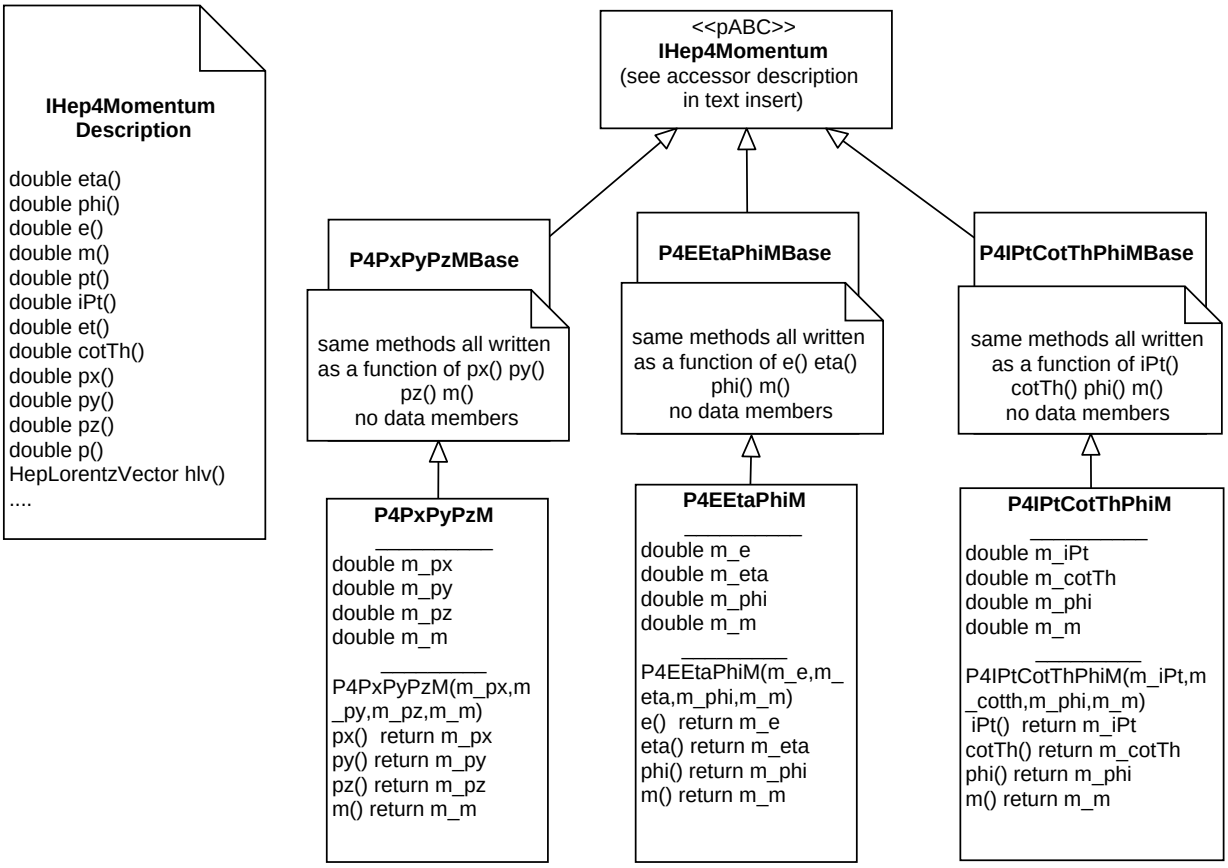

Figure 1: Class design for the four-momentum interface. IHep4Momentum is the pure abstract base class which defines the interface. Three intermediate base classes, shown at the top, define methods that are optimal for a given parameterisation of the data. At the bottom there are four concrete classes as an example of how the interface and base classes would be used in reconstruction data classes.

- ease the work of object developer.

An abstract base class IHep4Momentum lists virtual accessors to kinematical quantities, including a conversion to HepLorentzVector. Each type of reconstruction data class (CaloCell, Track, ...) has "natural" kinematical parameters; for example, energy, pseudorapidity and phi for a calorimeter cell, inverse pT, cottan(theta) and phi for a track. To complete the 4-momentum, the mass needs to be present. In most cases it would follow a well defined convention, for example zero mass for a cell and pion mass for a track. For each set of natural parameters one defines a base class (for example P4EEtaPhiMBase for a calorimeter cell) deriving from IHep4Momentum where all the accessors (except the one corresponding to the natural parameters) of IHep4Momentum are implemented with respect to the natural parameters' accessors. The base class is still abstract without any data members, and the natural parameter accessors are still not defined.

The reconstruction data classes would then sub-class from their corresponding abstract base class: P4EEtaPhiMBase for CaloCell, or P4IPtCotThPhiMBase for Track and so on, and implement the natural parameters' accessors. Concrete classes P4EEtaPhiM, P4IPtCotThPhiM, P4PxPyPzM and so on are also provided to serve as intermediate objects in calculations or to save 4-momentum of a given object, and can also be used as base class when inheriting the natural parameters' data members is indeed desired.

By doing this, using 4-momentum of a track or cell will no longer require the knowledge to re-write efficient code for the conversion from the natural parameters. Furthermore, helper classes (invariant mass, ΔR computation, etc...) can also be written as a function of IHep4Momentum and used for a wide class of objects. In particular, the change of representation, for example the computation of pseudorapidity will implicitly be done only for those objects for which the pseudorapidity is not a natural parameter, even if different types of objects are mixed.

3.2.2 Recommendations on common interface design

- Classes to provide standard common kinematic and position quantities must be defined, e.g IHep4momentum. All classes in the reconstruction and analysis domains should make use of these classes rather than providing their own arbitrary interfaces for the same functionality. Standard names for the methods to access them should be defined and become part of the ATLAS coding standard.
- There should be further studies of the time cost associated with different design options for use-cases identified as time-critical.
- Detailed examples of realistic user analysis code working on the results of fast and full simulation are needed to determine whether polymorphism can be usefully applied in this case.
- Functional requirements are the most important aspect of the software to get right in the first design iteration. The design should always take into account time constraints, initially by trying to avoid design mistakes that cripple the performance, then by going through cycles of optimisation.

3.3 Units and reference frames

3.3.1 Units

There is currently not a unique set of units in ATLAS software. Most reconstruction algorithms use the Geant3 units (centimetres and GeV), while some others use the so-called CLHEP units (millimetres and MeV) as Geant4 is, meanwhile at the Generators level an hybrid convention is used (millimetres and GeV). The use of multiple unit sets has been causing confusion and difficulties which will be recurrent if this state remains. When it was suggested in June 2002 to move to CLHEP units, there was an outcry from the physicists against MeV, finding it was a weird unit to use for 14000000 MeV centre-of-mass energy collision.

It is hence highly desirable that ATLAS decides for a unique unit set which would be used by default everywhere, except in a few well defined places beyond its control. In particular, tool and service interfaces and reconstruction event data model classes would use this unit set.

To ease the migration, and to handle the conversion at the interfaces where it is needed, a mechanism like the CLHEP unit header can be put in place. This is a set of constants (like *mm*, *cm*, GeV) defined in a header which allows to convert from/to the default units to any.

It should be noted that using CLHEP classes for e.g 4-momentum vectors absolutely does not make it mandatory to adopt CLHEP units, which only appear in one header file. If CLHEP units are not used, it will be possible to avoid collisions/confusion by defining an ATLAS unit header with constants like *amm*, *acm*, $aGeV$.

3.3.2 CLHEP

CLHEP [9] classes are already widely used in ATLAS software in particular the 4-momentum, 3D points and vectors classes, whereas the matrix package is seldom used due to its current poor performance.

Adopting these classes as a common language can only be beneficial for the readability and maintenance of the code. The CLHEP project is now part of the LCG project, so that long term support is guaranteed. A workshop has taken place at CERN in the beginning of 2003 (see http://proj-clhep.web.cern.ch/proj-clhep/Workshop-2003/summary.html), in view of a major new release towards the end of 2003.

In ATLAS, a number of deficiencies of these classes has been identified, for example the fact that there is no choice of data members: these are Px, Py, Pz and E in double precision. If a vector is created from E, eta, phi (which is natural e.g. for calorimeter reconstruction) and retrieved as E, eta and phi this involves conversion to/from Px, Py, Pz, E which is inefficient. Another issue is that there is no abstract base class for these classes, but this has been rejected during the CLHEP workshop on the basis of performance.

It is strongly suggested that ATLAS becomes more actively involved in the development process of these CLHEP classes, so that the possible performance issues in ATLAS use cases are lifted. The goal should be to make sure CLHEP meets ATLAS's requirements so that it can be recommended for use in ATLAS software.

3.3.3 Reference frame

So far ATLAS reconstruction is working in the most natural frame, centred on the detector centre and aligned on the beam line. However in real life different frame might need to be used. For example, a natural frame for tracking is the one where the z axis is the solenoid symmetry axis, which will not exactly coincide with the detector symmetry axis. The real beam line is another useful axis. For the calorimeter, the natural eta/phi parameters are really position rather than direction, and assume the particle comes from the centre of the detector. Converting into usable direction requires the knowledge of the primary event vertex, and possible calorimeter misalignment. For these reasons, there should be a mechanism (yet to be defined) which will avoid the ambiguity for any instance of any objects in the EDM and ease the conversion between reference frame.

It is recommended that a single natural reference frame be chosen for the combined reconstruction, analysis and anything down stream of that in the reconstruction data flow. In the subsystem reconstruction domains, the most natural frame for each package can be chosen freely for internal use, but the final output of the subsystem must be transformed into the frame specified by the combined reconstruction.

3.3.4 Recommendations on units and reference frames

- ATLAS reconstruction should adopt a unique unit set, and implement a mechanism à la CLHEP to ease migration and conversion to the few packages following different units.
- CLHEP 4-momentum and vector classes should be used by default.

- ATLAS should take a more active role in the CLHEP development process.
- A mechanism to specify the reference frame needs to be designed.
- A standard reference frame for combined reconstruction and analysis must be chosen.

3.4 Separation of event and non-event data

This section discusses the separation of Event Data (ED) from Detector Description (DD) and other non-event data.

It has been observed that the creation of reconstruction input objects (e.g. calo cells, indet clusters, muon clusters) can take a long time every event. Investigation of this problem revealed that most of the time was being spent not in the object creation overhead, but in re-computation of non-event (detector description) data. This is wasted time because the detector description doesn't usually change from event to event.

The Calorimeter subsystem has been taken as an example to illustrate the problem and explore solutions, but this equally applies to event data in all subsystems. A CaloCell currently contains both Event Data (ED) (Energy, time) and Detector Description (DD) data (cell position as eta, phi). It also contains an identifier of course. Users of CaloCell find it natural that this event and non-event data are available together through the CaloCell. This is because it belongs to a category of objects in the calorimeter subsystem EDM (along with classes like cluster and tower) which have clients in both the subsystem reconstruction, combined reconstruction and analysis domains, but, uniquely and coincidentally, it has the same granularity as the raw data (LArRawChannel, TileRawChannel). Most classes in the subsystem reconstruction domain are composites or derivations with a coarser granularity than the raw data and computed position parameters.

Currently, the solution to this requirement on the CaloCell is implemented by getting the DD data and copying it into the cell on construction. DD data is accessed via the Identifier helper classes every event. The helper classes are called to compute the geometry information every event even though it does not change - thus adding a significant unnecessary overhead. Storing DD data in the cell makes the cell objects larger and more time-consuming to construct. Furthermore it is unnecessary to copy the DD info every event as is does not change that frequently.

Reminder: DD data depend on time-varying conditions. They will only be changed or rebuilt when a time boundary of their conditions validity interval is crossed. The interval of validity service (IOVSvc) is provided to handle this automatically. In the trigger and prompt reconstruction this will happen infrequently, while in analysis jobs which may run over summary data from a whole year, it is likely that there will be few events with identical conditions. The current ATLAS Detector Description model will be described in [3].

There is a different case for composite objects that don't have the same granularity as the readout, and for which there are not a simple set of fixed positions to be populated, for example calorimeter clusters or inner detector space points. Here, the result of the position calculation is likely to be worth caching for performance reasons, but it should be calculated using the current conditions whenever the object is instantiated.

3.4.1 Recommendations on the separation of event and non-event data

- Event and non-event data should be separated:
  a) by not directly copying or caching non-event data in event data objects;
  b) by not providing access to non-event data via event data methods, except in the cases described below.
- ED should use a central optimised memory management service (such as the datapool), if the memory allocation overhead is likely to be significant.

- separate objects containing DD information should be stored in the Detector Store and arranged to be updated automatically when time-varying conditions it depends on become invalid. The granularity of these objects will be the same as detector elements in the geometry model.
- Some DD information can be made available through the ED class interface, provided that:
  - Pointers, not data, are cached in the ED objects.
  - Methods are only provided if they are used frequently, so the justification is not just convenience but improved performance too.
  - There should be several clients for the methods.
  - There must be a limit to some small number of methods to access DD data in this way.
- Classes with special requirements, such as RDOs which are constrained to represent the byte stream, should not provide DD information through their interface. They should just provide their identifiers.
- For classes which are commonly used in the analysis domain, the case is strong for the class to present 4-momentum-like information, which probably include information derived from DD.
- In general, clients can get DD information by taking the identifier of the ED object and looking up the corresponding DD via the Detector Description Service.
- Coordinate, frame and unit transformations will be done by helpers, not the ED or DD classes themselves.
- A couple of examples should be taken to apply these recommendations to as an exercise, feed back any problems, and to sort out implementation details.
- Where a pointer to DD information is cached in an ED class, there are implications for the persistency of the ED object. The pointer should not be persistent, so it will need to be initialised somehow after the object is constructed from the persistent store. The database group should consider this.

4 Reconstruction Data Flow

4.1 Overview

An overview of the reconstruction data flow is shown in figure 2. Dataflow is from top to bottom in the figure, with event data on the left and algorithms on the right. The event data model domains are those described in section 3.1. The algorithms which connect these domains have two types of relationship to them. In general, algorithms will both use and create objects in their own domain, while they may also use objects from preceding domains but cannot create or modify objects there. Of course algorithms cannot have any relationship to downstream domains.

Most event data will have some relationship to the truth event data via associations which are described in the navigation section. These relationships are omitted from the top level diagram for clarity. Similarly other types of relationships between data in different domains is not shown as it is not informative at this level of abstraction.

In fact, data flow is not completely linear as implied above; there are many cases where earlier steps are repeated as part of the refinement of the data. For example, inner detector tracks can be refitted with an electron hypothesis if they are found to match a calorimeter cluster during the combined reconstruction of e/gamma candidates. While this complicates the flow, it does not change the domain model, as the fitting tool which creates a better track parameterisation (in the tracking reconstruction event data) is part of the "tracking" subsystem reconstruction domain. Such relationships between algorithm domains are intentionally omitted from the diagram. There are no particular constraints on this. Algorithms will routinely use tools from other domains to accomplish their goals. More examples are shown in the following sections.

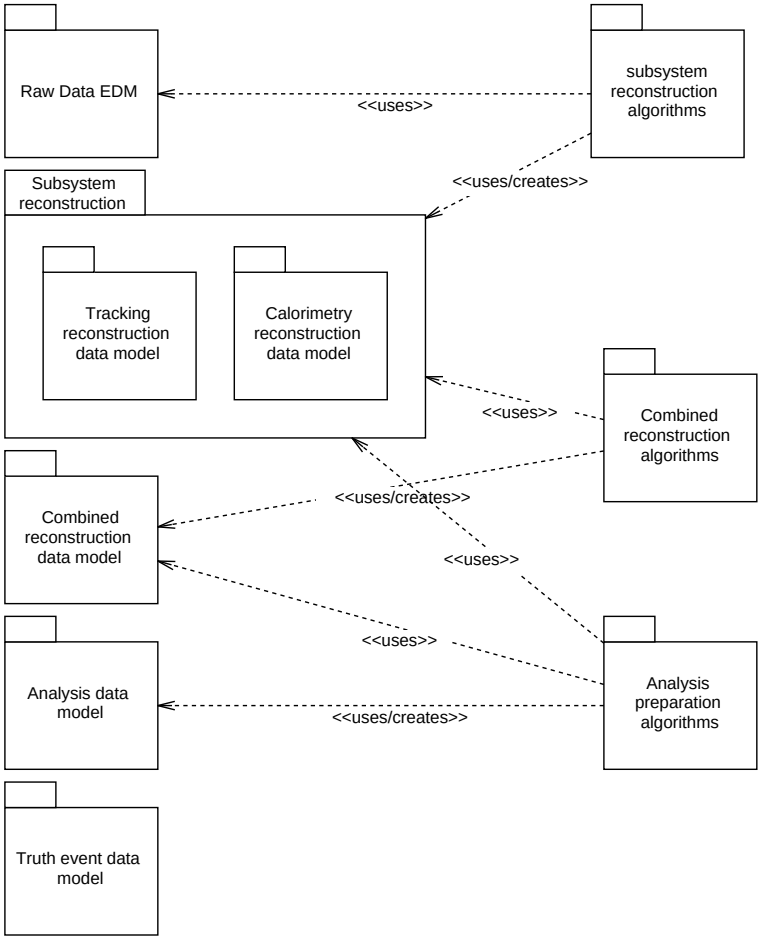

Figure 2: Overview of the reconstruction dataflow. The packages represent the subdomains of the reconstruction event data model and algorithms. The arrows show dependencies between the algorithm packages (on the right) and event data packages (on the left). Not all types of dependency are shown. See text for further details.

The choice of subsystems comes from the following observation. Stating the obvious, there are two types of detector in ATLAS: tracker (Inner Detector and Muon Spectrometer) and calorimeter (LAr and Tile). Even though software development has proceeded in the past in the different detectors more or less independently, there is a lot to gain to have common development in these two areas, in particular common base classes and common tools (at least tool interfaces), starting with the Reconstruction Input Objects (RIO). For example:

- Tracking in the Muon Spectrometer and tracking in the Inner Detector both need to do track finding with different technologies of tracking devices in inhomogeneous magnetic fields. Even though it is very likely that the pattern recognition strategies will remain very different, the track fitting and extrapolating can certainly share the same interface and probably some code.
- In the case of jets or missing E_T, a Tile cell is no more different from a LArg EM barrel cell than a LArg EM endcap cell. Hence software to apply the proper physics dependent calibration, finding neighbours, not to mention the basic interface to energy/eta/phi quantities should be provided independently of specific parts of the calorimeter.

The rest of this subsection describes the dataflow, event data model and algorithms in these subsystem domains, the combined reconstruction domain and the analysis (preparation) domain.

4.2 Calorimeter

The following section described the Calorimeter Event Data Model. At the Raw Data Level, Liquid Argon (LAr) and Tile Calorimeters retain their individual identities. During the first phase of the reconstruction, both sub-systems adhere to a common interface because of a strong requirement that downstream algorithms obtain the energy of a calorimeter cell without knowing their identity.

During the regular data taking, the digitised samples from the calorimeter feed as input to the RODs. These are described by the LArDigit and TileDigit classes (labelled xxxDigit in the figure). The output of the ROD contain the fitted information and are described by the LArRawChannel and TileRawChannel classes (The Calorimeter RDO's, labelled as xxxRawChannel in the figure).

The LAr and Tile RawChannels (RDO's) form as input to the first stage of offline reconstruction. Sometime, the digitised samples may be written out directly, in which case an offline emulation of the ROD processes the digitised sample to make the RDOs.

The first stage of the reconstruction converts the LAr and Tile RDO into CaloCells. These are calibrated cells that can be referenced by all downstream reconstruction algorithms without needing to parse the identity of the cell.

Further calorimeter requirements include the processing of LAr and Tile Hits (the simulated visible energy deposition) which may feed into the digitisation package that produce the input to the RODs or directly into the first stage of reconstruction. The design that the calorimeter raw data model must adhere to is already described in detail in Reference [7]. This section describes the calorimeter data model from the first stage of the offline reconstruction.

4.2.1 Calorimeter Cells

The LAr and Tile Calorimeter EDM converge at the cell level which correspond to the Reconstruction Input Object (RIO) for the calorimeter. Both adhere to the same base class "CaloCell" while detector specific information can be maintained in a sub-class "LArCell" or "TileCell". Since CaloCell corresponds to a physical cell of the calorimeter, they are "Identifiable". They should also sub-class from the interface "IHep4Momentum" which allows them to be used by generic jet reconstruction algorithms. IHep4Momentum, as described in 3.2, allows access to the HepLorentzVector. In the case of CaloCell, the assumption of a massless particle and the nominal z-vertex should be used in the computation of the four-momentum.

CaloCells are calibrated calorimeter cells produced at the first stage of the reconstruction from either raw data objects (RDO) or directly from simulation hits. They hold event type data such as energy, time and χ^2. The ideal position information of a CaloCell can be derived from an identifier and corrected for misalignment in a non-event dependent way. Since such information does not necessarily change from event to event, it should naturally belong in the detector description (DD) store and not be cached physically in CaloCell. A DD object at the cell level should provide the position and other cell specific information after automatically correcting it for any mis-alignments (if such requested) when dereferencing the pointer to such an object. This infrastructure is already in place via the IOVSvc and should be utilised. Algorithms should retrieve this DD object from the detector store to access any cell level description. (This should adhere to the general principles described in section 3.4.)

CaloCells are created either directly from simulation hits (LArHit or TileHit) or can be created from Raw Data Object (output of RODs): LArRawChannel and TileRawChannel. RDO for LAr and Tile are substantially different, so they are not required to adhere to the same common EDM. A RawChannel for Tile corresponds to the output of a pair of photo-multipliers while it represents a single digitised output for LAr. CaloCells are created by different algorithms or converters. There is no restriction for the granularity of the collection for CaloCells. In the high level trigger, the granularity of the collection is optimised for efficient access, while such fine granularity may not be needed in a full reconstruction operation. However it would be desirable to have a similar structure both in the offline and online EDM since access to Calorimeter Cells in regions of interest is also expected in the offline environment (it is the normal access pattern in the online environment).

It is desirable that CaloCells, RDOs, digits and hits are all identified by the same identifier as they all represent the same physical calorimeter cell. It is recognised that real raw data in the bytestream is imprinted with an online identifier (describing the physical crate, FEB, and other hardware bodies via which the cell is read out). The conversion of this online identifier to an offline identifier (that which logically describes the physical location of the cell) should be done during the conversion cycle. Thus such conversion will then be limited to raw data converters and the treatment and analysis of simulation data will remain free of such conversions. Furthermore, it is recognised the cells may be created directly from hits or RDOs. It therefore becomes impractical to carry a physical navigation from cells to its parent within the cell. Since cells, hits, digits, and RDOs carry the same physical identifier, the identifier can be used as the mechanism for navigation. Given an identifier, the client can locate either the hit, digit or RDO (whichever appropriate) in their respective collections. Similarly, the navigation of RDO to Digits can be accomplished in a similar manner.

CaloCells represent the final calibrated cells that is used as an input for various algorithms. It does not form input just to the clustering, but can be input to further downstream algorithms such as Missing ET and Jet Reconstruction. Since several downstream algorithms use CaloCells just as other Calorimeter event data such as CaloTowers and CaloClusters and since there are a few heavily accessed CaloCell detector description information (eta and phi specifically), the CaloCell can cache the pointer to the cell detector description object and provide accessor methods that forward the call to this object. However, since correct mis-alignments must be picked up automatically via the IOVSvc, the cell detector description object must be cached in CaloCell on an event by event basis.

There are 200,000 calorimeter cells, as many as hits, digits and RDOs. Use of a central architecture (such as datapool) for creation of objects and other performance efficient measures such as separation of event and non-event data should be followed as these are fundamental units used not only by several offline algorithms but are used in the high level trigger environment as well. Persistency support should adequately be provided by the central architecture that supports such a data model.

4.2.2 Calorimeter Towers

The next step in the calorimeter reconstruction chain is typically the creation of calorimeter towers that may be necessary for some clustering algorithms. Other clustering algorithms can work directly off cells. There are typically two sets of calorimeter towers:

(a) EM Towers: towers that are formed by summing all the cells in a 0.025 x 0.025 eta-phi window over

the sampling of only the electromagnetic compartment and

(b) Hadronic Towers: towers that are formed by summing all the cells in a 0.1 x 0.1 eta-phi windows across all samplings of the calorimeter.

There are therefore two collections of CaloTower objects in the transient store. However, the algorithm and the collections should not be constrained to only such tower collections: variations in granularity, regions and tower size should be allowed and possible. Since Calorimeter towers are created from a list of CaloCells, they should carry a list of these cells from which they are constructed.

CaloTowers provide the total energy and an energy-weighted position of the tower and an iterator over a list of CaloCells that it is made up of. It also should allow the possibility that the cells that contribute to a tower may carry different weights. There is no foreseen difference for CaloTowers between LAr and Tile and hence no detector specific sub-classes is required. The primary difference amongst the various CaloTower collections is the tower segmentation which is specified the width and depth of the requested tower. The properties of the tower, which does not change from event to event, can therefore be held separately in a CaloTowerSeg object. Furthermore, since all CaloTowers in a collection are created from the same set of properties and it is desirable to access the knowledge on how this tower collection was created, the CaloTowerSeg should be owned by the CaloTowerCollection. It is noted that this structure already exists in the present reconstruction. However, the algorithms need to be restructured to allow creation of CaloTower objects in regions of interest.

4.2.3 Calorimeter Clusters

Calorimeter Clusters are ensembles of elements in a calorimeter, such as cells, towers or clusters themselves. Clustering Algorithms may work of different inputs such as CaloCells and CaloTowers. Certain clustering algorithms (like jet algorithms) may cluster simply in four-momentum space and hence need not have the knowledge of the concrete type. The same or other clustering algorithms may cluster non-calorimeter elements such as tracks and truth particles. It is therefore foreseen that clusterizable elements, such as cells, tracks and such will have a a common interface to allow such generic clustering algorithms to work (several ways to achieve this are discussed in section 3.2. Regardless of the input to the clustering algorithms, the output is a collection of EnergyClusters: a grouping of clusterizable elements. EnergyClusters may be CaloClusters or TrackClusters or other concrete clusterizable elements.

These EnergyClusters should then form as input to all combined reconstruction packages such as e/γ, tau, jet or energyFlow reconstruction. Clustering algorithms should therefore reside in a common package and the same algorithm usable by either EM or Hadronic reconstruction by just reconfiguring the algorithm via the jobOptions file. Downstream produced combined reconstruction objects (egamma, tau or jet) should all be able navigate back to its parent EnergyCluster.

Furthermore, EnergyClusters should be able to navigate back to the constituents it was made from. Since EnergyClusters may be made from different sources (Cell, Towers, Tracks, Truth, ...): there are two ways to accomplish this navigation. Sub-classes of EnergyClusters, such as EnergyClusterFromCell, EnergyClusterFromTrack, etc. would hold the backward navigation or the EnergyCluster could itself be templated. The templated approach is more consistent with the proposal recommended for composite navigation. It also is likely to be more performance efficient saving unnecessary type casting. The disadvantage of a templated class is that many of its clients (including ATLFast) must agree to common design. Since Energy Cluster is a simple object comprising of a list of constituents, this should be possible. Downstream algorithms, such as egammaRec and JetRec should calculate the properties based on the EnergyCluster object and the objects produced must be navigable to the EnergyCluster.

The algorithmic code that performs the actual clustering and corrections should be implemented as AlgTools. The flexibility to allow different sets of clustering algorithms and sets of corrections must exist at this stage. Hence a top level algorithms configured appropriately at run time steers the set of tools. Furthermore, the implemented tools must be usable in the HLT environment and follow recommendations suggested at the steering section. The corrections at this stage are only calorimeter dependent corrections. Any particle dependent corrections are made during the subsequent reconstruction stages when the relevant

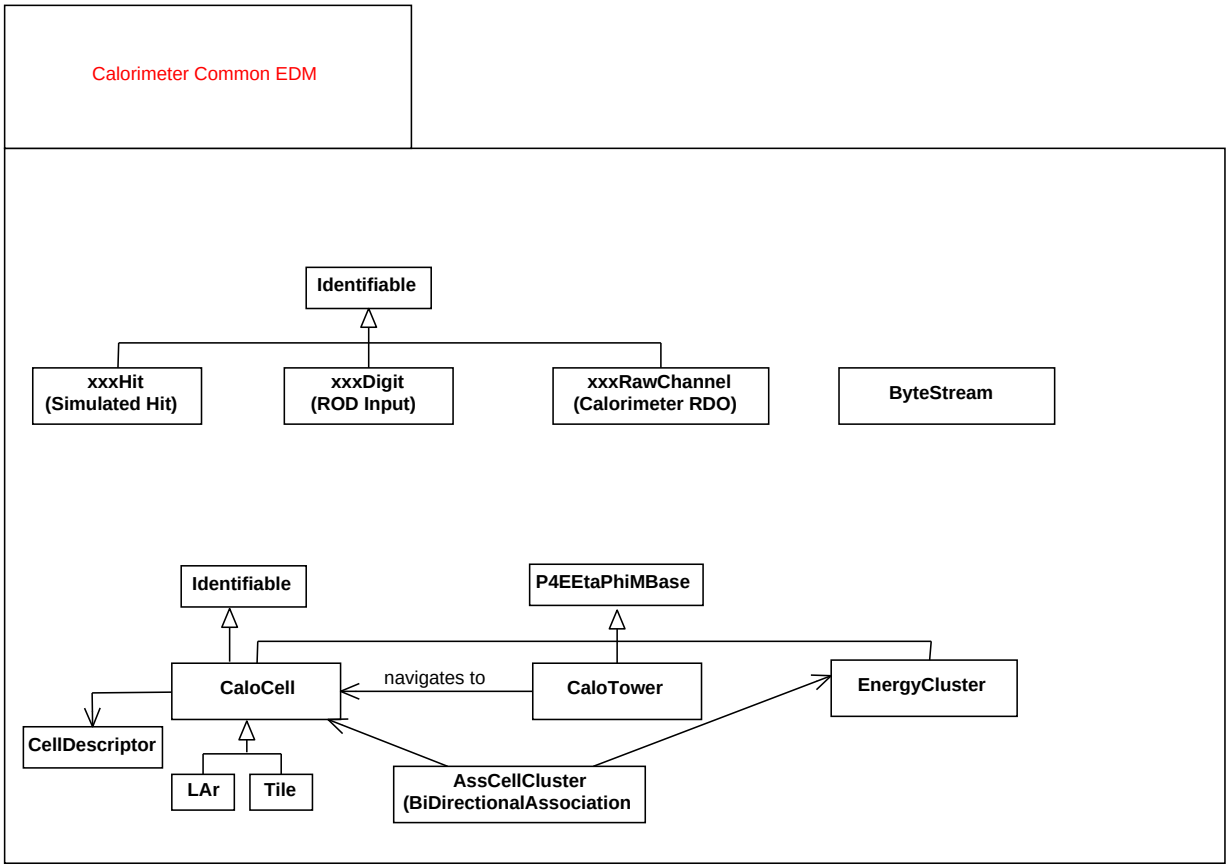

Figure 3: UML representation of the common calorimeter event data model classes. See text for details.

information become available.

Note that a templated EnergyCluster class is likely to make some of the generic algorithms templated as well. The significant deviation from the current scheme of reconstruction is the creation of ProtoJets, an unnecessary intermediate step whose primary functionality is to provide a 4-Momentum interface. This has been achieved by requiring the individual classes to provide this interface.

Reverse navigation between cells and clusters or cells and towers is discussed in the Navigation section. An external association object is created in the transient store and filled at the time of associating a cell to the cluster. This external association object can allow the possibility for a bi-directional association. Downstream algorithm then make use of this object to determine the cluster to which the cell belongs.

4.2.4 Recommendations for calorimeter data flow

- Event and Non-Event separation for CaloCell.
- Implementation of clustering and correction algorithms as Tools
- An EnergyCluster as a replacement for the existing CaloCluster. The final design recommendation of whether it is templated or a base class will be made based on the requirements of the downstream combined reconstruction algorithms.
- Apply the design pattern recommended in section 3.2 to provide the 4-momentum interface needed by combined reconstruction algorithms.

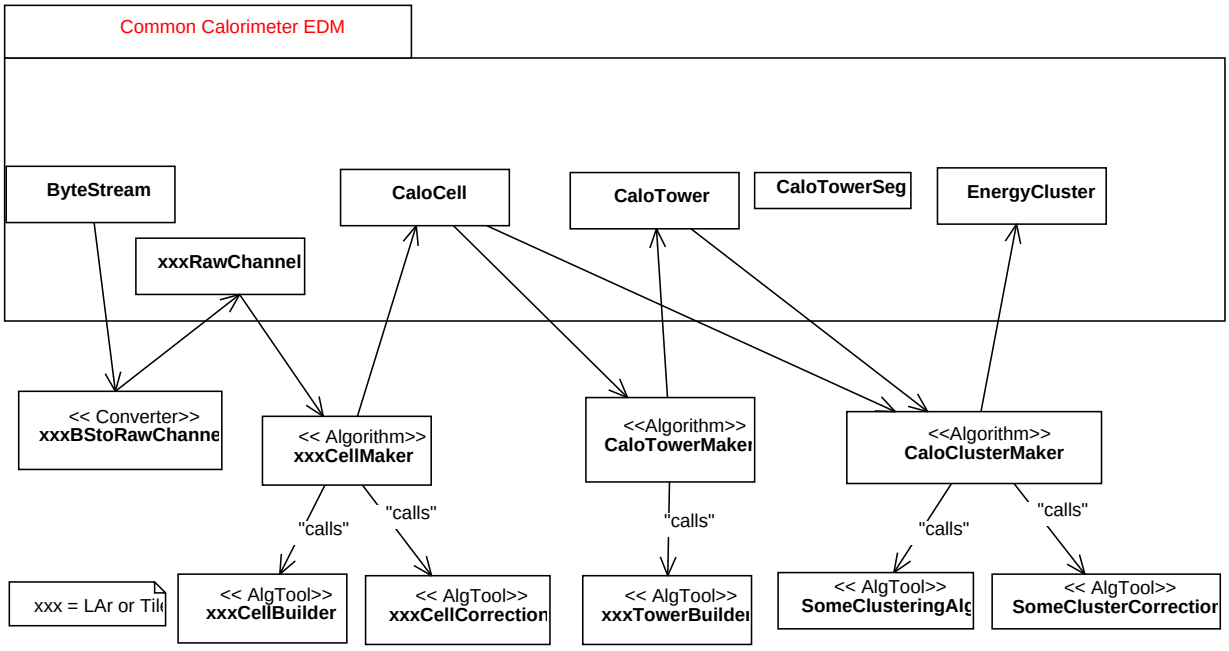

Figure 4: The calorimeter reconstruction steps from input RDOs. All the output of the calorimeter reconstruction steps may be usable by downstream combined and analysis reconstruction. The arrows between EDM and algorithms show the input and output, the arrows between algorithms represent calls.

4.3 Tracking dataflow: Inner Detector and Muon subdetectors

In this section, a proposed tracking flow is presented for all InnerDetector and muon Tracking algorithms. First, figure 5 shows the common classes forming the Event Data Model for the tracking. These come from and follow the requirements for the InnerDetector [8], it is strongly recommended that readers consult this document to learn the definitions of the various classes, and their relationships to each other [3]. We don't describe the content of each class, as this task needs to be tackled by the Inner Detector and Muon communities in order to produce a detailed design. The different TrackParameters are dependent on the reference frame used (solenoidal field frame, beam axis frame or subdetector frame) a discussion on reference frame is presented in section 3.3. We note that there has been several attempts in recent years to have a unique class interface (common RIO class, common Track class) and we recommend to continue this exercise.

4.3.1 The Raw data flow

In figure 6 one finds the proposed dataflow that goes from the raw event either in the bytestream format or already in RawDataObjects, all the way to the SpacePoints. In figure 7 we show the uses relationships of the various Algorithms and AlgTools of the raw data flow. The Atlas dataflow that goes from Simulation to Raw data is presented in [7]. The first step is to create the RawDataObjects(RDOs). According to the Raw Dataflow this can be done through the digitisation or through the ByteStream converters. In the case of Level 2 algorithms, the idea is to bypass the RDO object creation and make directly the ReconstructionInputObjects (RIOs). Examples of RIOs are Clusters for the Silicon detectors and CSC chambers and DriftCircles for the TRT and MDT chambers. The RIOs are common to the Level2, EF and offline domains. However, the different domains have different requirements for this class. For example the Level 2 needs to have adequate speed performance so it might require a lighter object than the other domains. Hence one can imagine that within the RIO package there would be a base class used for Level

[3] Note that the Cluster in the Requirements document is now called Reconstruction Input Object (RIO), so ClusterOnTrack is now RIO_OnTrack. Also, the OutputTrack is here simply called Track but still refers to the object which will be used in the combined reconstruction, and analysis domains.

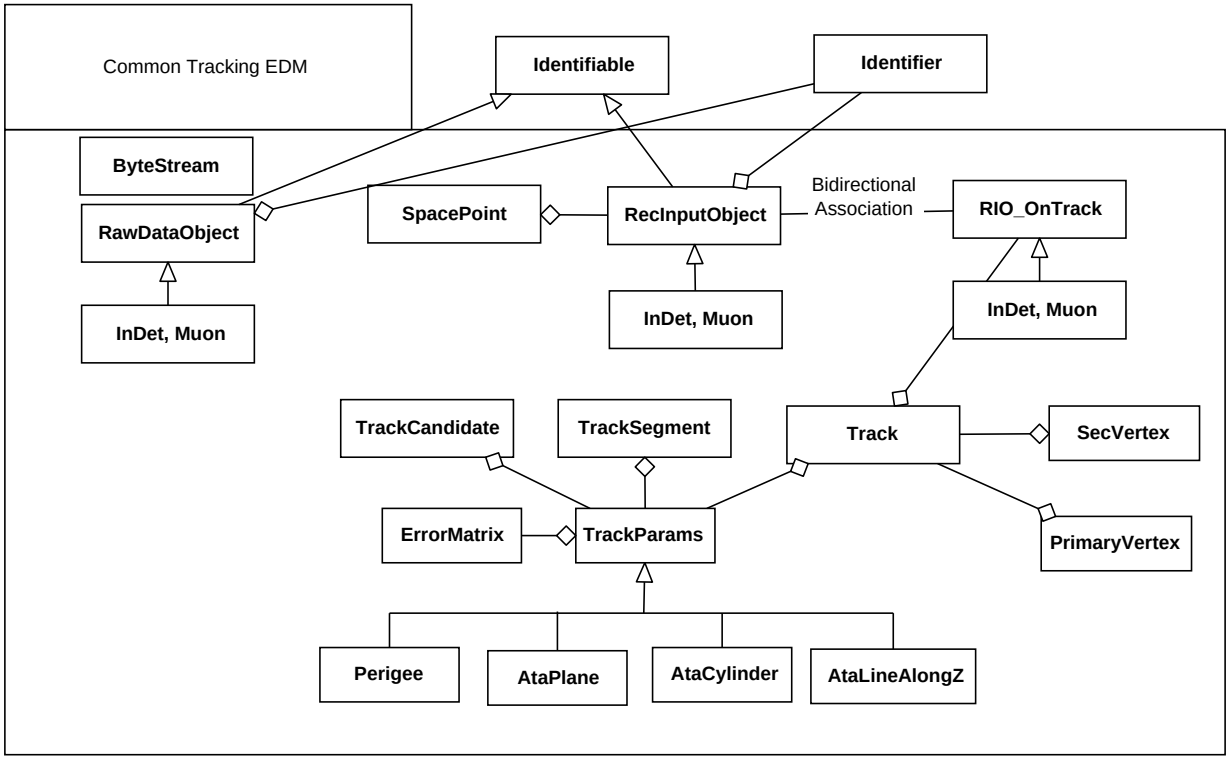

Figure 5: Common Tracking EDM.The InDet, Muon class represent the fact that whenever appropriate there should be concrete classes representing the different subdetectors: Pixel, SCT, TRT, MDT, CSC, RPC and TGC.

2 and a concrete class which would extend the base class, used for EventFilter (EF) and offline (this might be done through another mechanism than inheritance, e.g. aggregation, see section 3.2). It is seen that the RIOs are made through two separate paths: from the converters (for Level2) and from the RDOs (for EF and offline). The common part between these two paths is contained in the RIO_Maker reconstruction Algorithm. The extra part that fills the extended RIO class for EF and offline is contained in the GlobalPositionMaker. The RIO_Maker and the GlobalPositionMaker make use of the Calibration and Alignment services respectively, which access ConditionsData in the database. Finally the SPFormation reconstruction Algorithm creates SpacePoints out of RIOs, for the relevant subdetectors (Pixel (essentially a copy) and SCT but not TRT, for example). It is noted that most of this dataflow is already implemented. We point out here an example of the relevancy of having a single EDM since this allows developers to study easily the boundary between Level2 and EventFilter.

In figure 8 we show the sequence diagram associated with producing SpacePoints from RDOs for the InnerDetector. This sequence corresponds to the case of offline reconstruction, there would be a similar sequence for the case of Level 2 or EF, where the ByteStream converters would be called. There would also be a very similar sequence for the case of muon tracking.

4.3.2 Finding Tracks

The next step in the tracking dataflow is to find tracks in order to produce TrackSegments. Figure 9 shows the proposed dataflow diagram for making such objects while figure 10 shows the modularity. The TrackFinder module is a reconstruction Algorithm which takes as inputs either SpacePoint objects or RIO objects and make TrackSegment objects as output. Within the TrackFinder Algorithm there are one or several AlgTool that are called to do detector specific pattern recognition. The diagram gives possible examples of such AlgTools, but this list is not exhaustive. In general (not always), reconstruction AlgTools take as input physical objects (four-vectors, etc.), in contrast to the reconstruction Algorithms which take

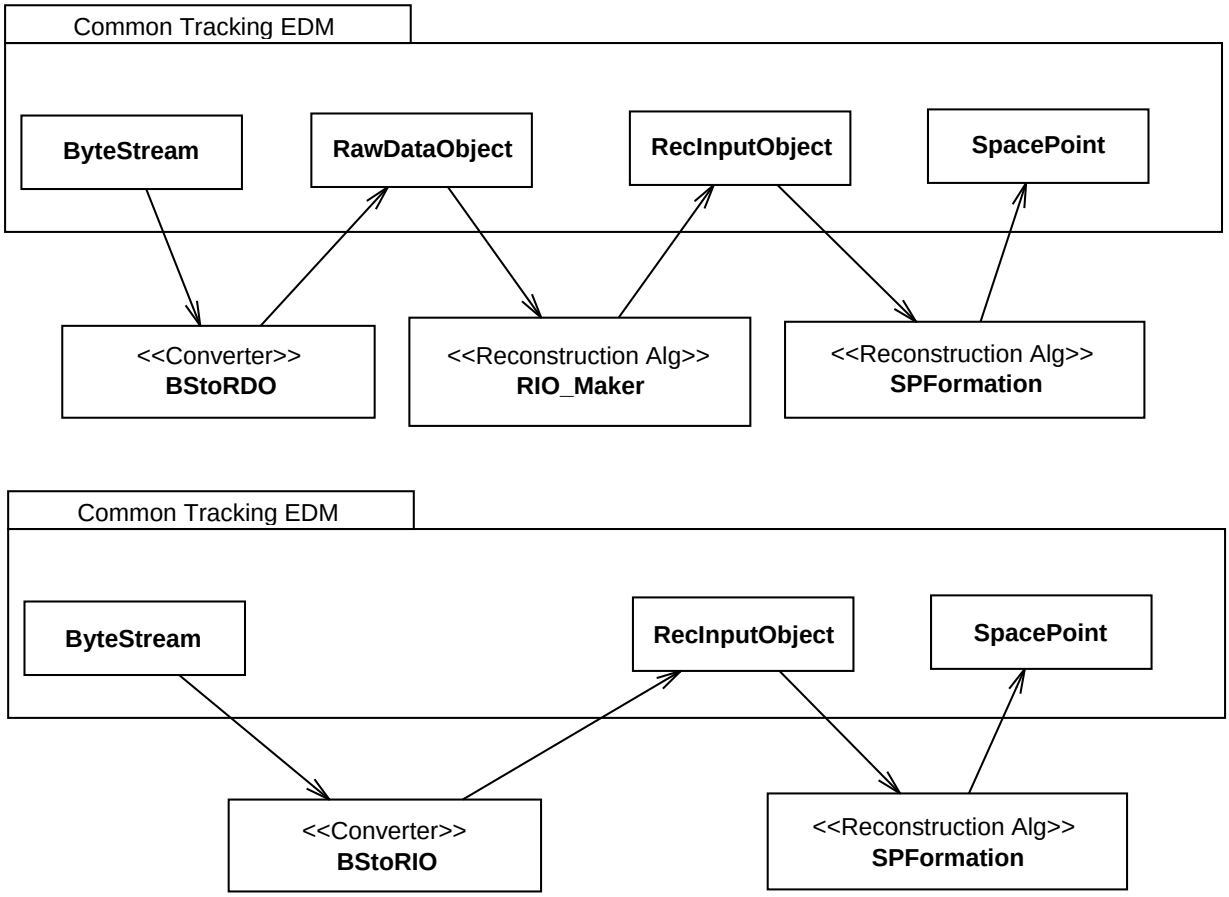

Figure 6: Raw algorithms sequence. Top is for EventFilter/offline and bottom is for Level2. See text for details.

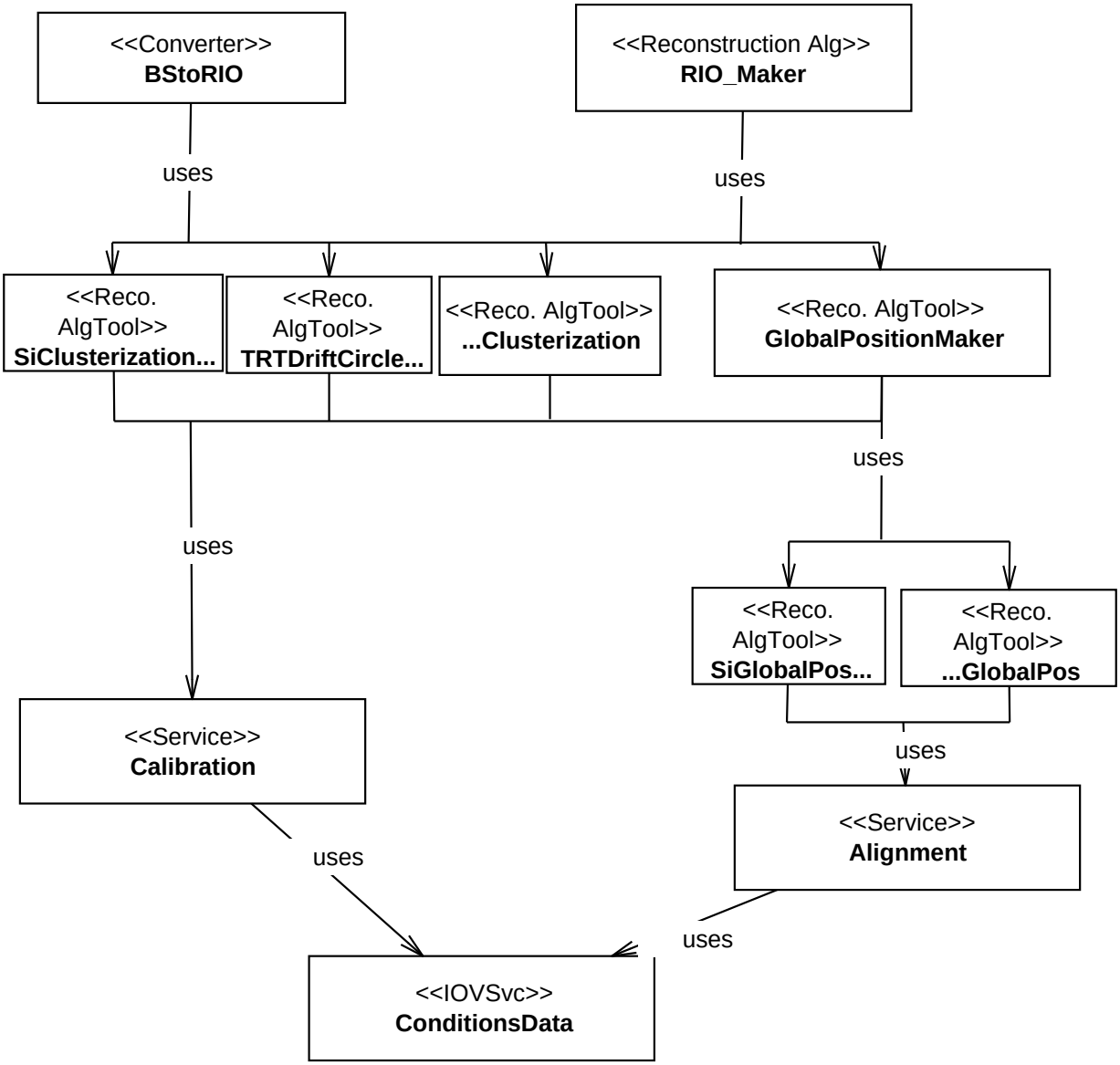

Figure 7: Raw algorithms modularity. The label "uses" should be read as "can use". See text for details.

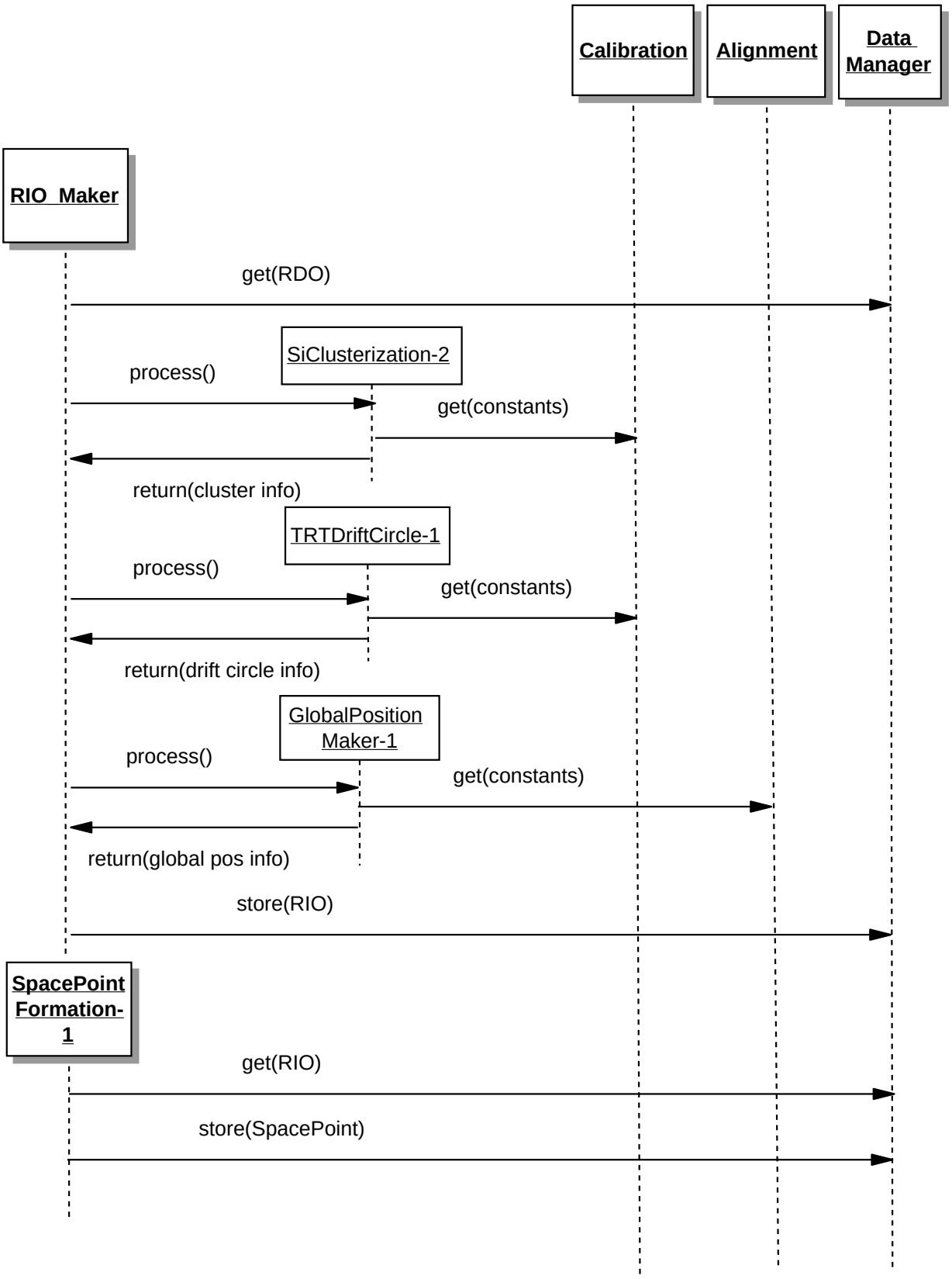

Figure 8: Inner Detector Tracking sequence. Raw data preparation. See text for details.

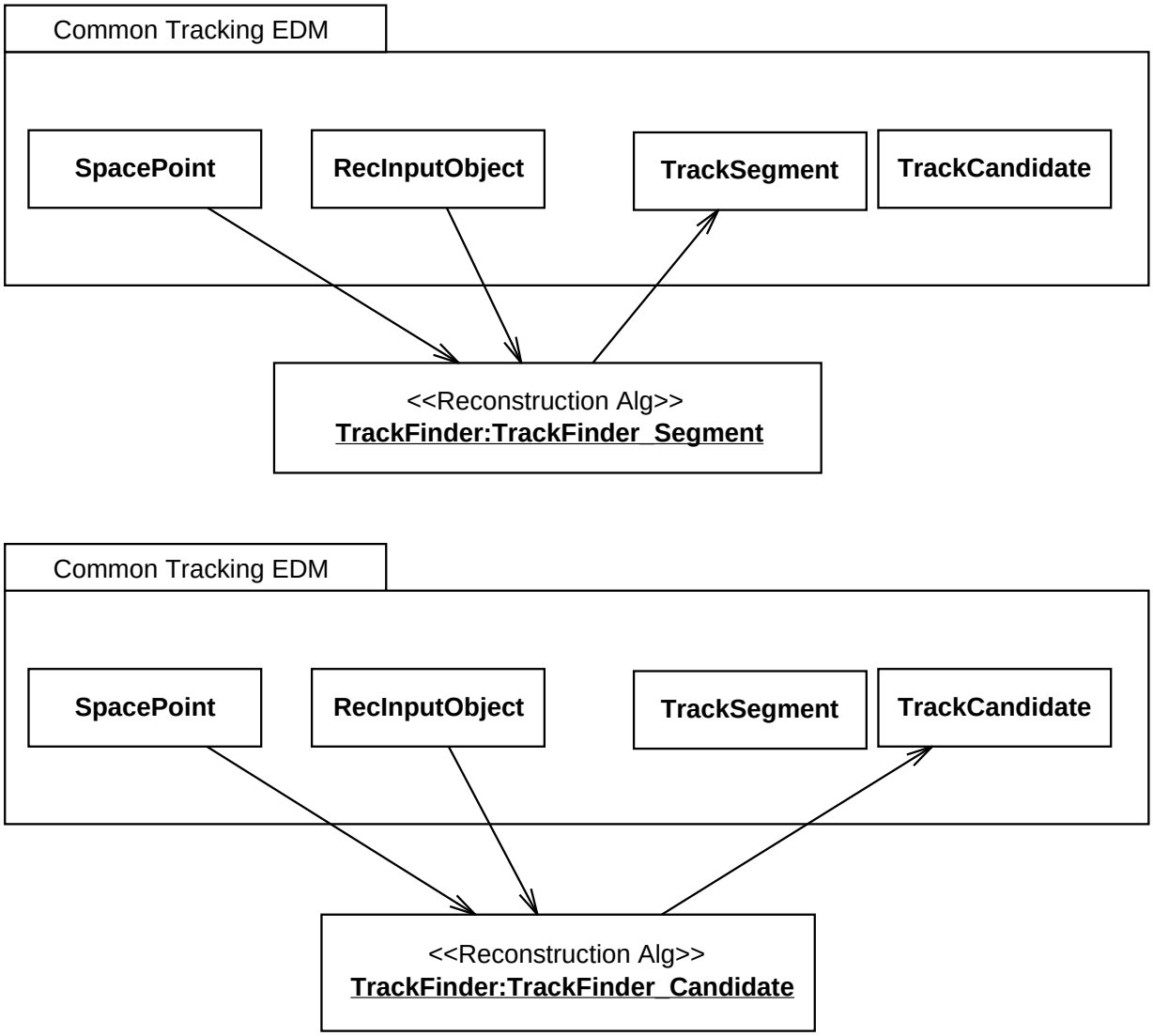

Figure 9: Track Finder algorithms sequence. The notation "object:class" is used. See text for details.

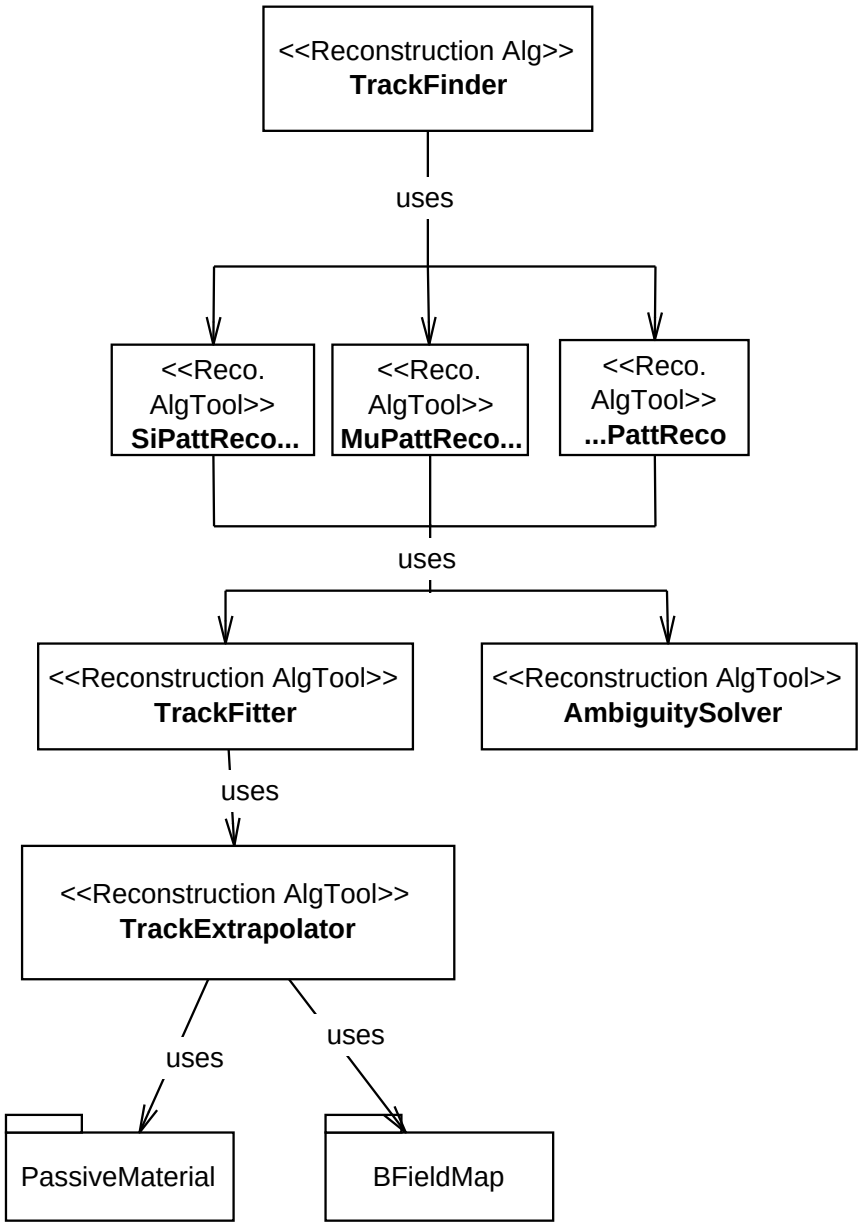

Figure 10: Modules associated with the Track Finder. The label "uses" should be read as "can use". See text for details, for example on the AmbiguitySolver.

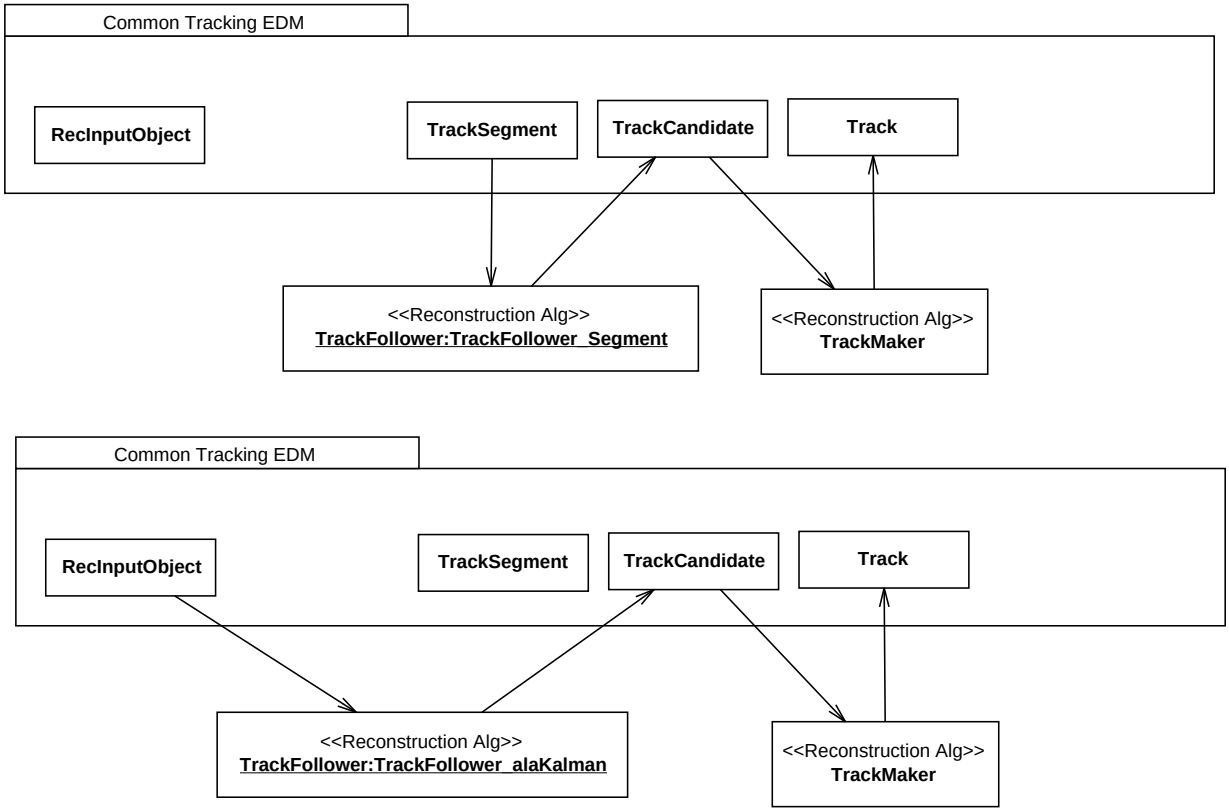

Figure 11: Track Maker algorithms sequence. The notation "object:class" is used. See text for details.

care of extracting the physical components from the objects coming from the common EDM. The TrackFinder Algorithm typically makes fits to decide on a set of TrackSegments. The TrackFitter is the module called to perform fits of tracks, and it can contain different implementation, depending on the level of precision and on the method used for fitting (Kalman fitter vs chi-square, for example). Typically, there are going to be on the order of a few thousands of those TrackSegments, given the combinatorics. If one were to make Tracks or even TrackCandidates out of all of those, this would be very time consuming. It is possible to cut down on the number of possible tracks by calling the AmbiguitySolver during the Track Finding stage. This AlgTool takes into account shared hits among the various proposed TrackSegments (including resolving so-called left-right ambiguities) and outputs the most optimal set of TrackCandidates given the information known about the event at this stage of processing. This possibility in running the TrackFinder is the reason why a possible output of this Algorithm is a collection of TrackCandidates. The AmbiguitySolver may need to call the TrackFitter as well, as indicated in the diagram. The AmbiguitySolver module relies on AmbiguityAssociation objects, which should follow the recommendations of section 5.

The TrackFitter might need to extrapolate tracks in a given volume, for example to the next detector element following an helical trajectory. Another example would be to extrapolate a track through the magnetic field to the beam spot. The TrackExtrapolator is the module which provides this functionality. It needs to know about the Atlas magnetic field (both the solenoidal and toroidal fields) and about the passive material. It is assumed that the TrackExtrapolator implementation will be a collaboration between the tracking community and the detector description community.

4.3.3 Making Output Tracks

The TrackFollower module makes TrackCandidates out of TrackSegments as shown in the dataflow figure 11. The modularity of the Algorithms and AlgTools is shown in figure 12. The main purpose of the

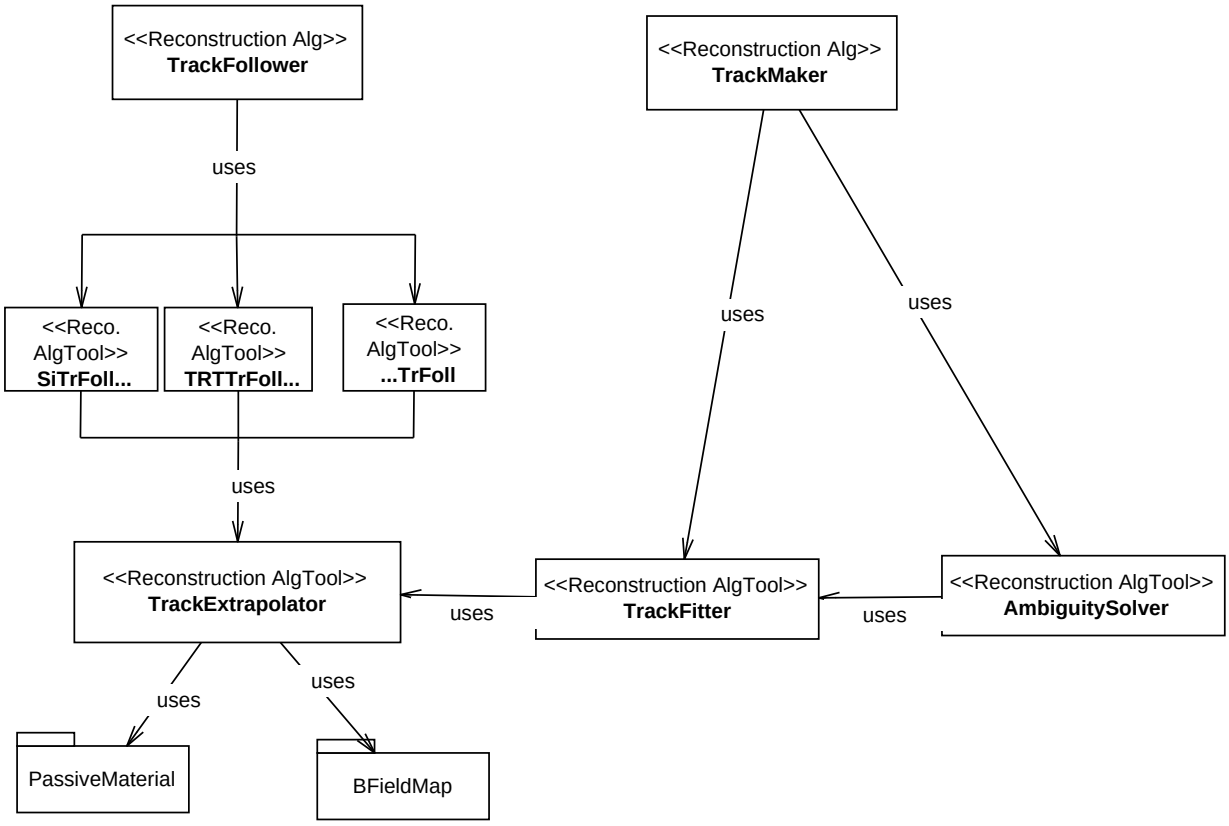

Figure 12: Track Maker algorithms modularity. The label "uses" should be read as "can use". See text for details.

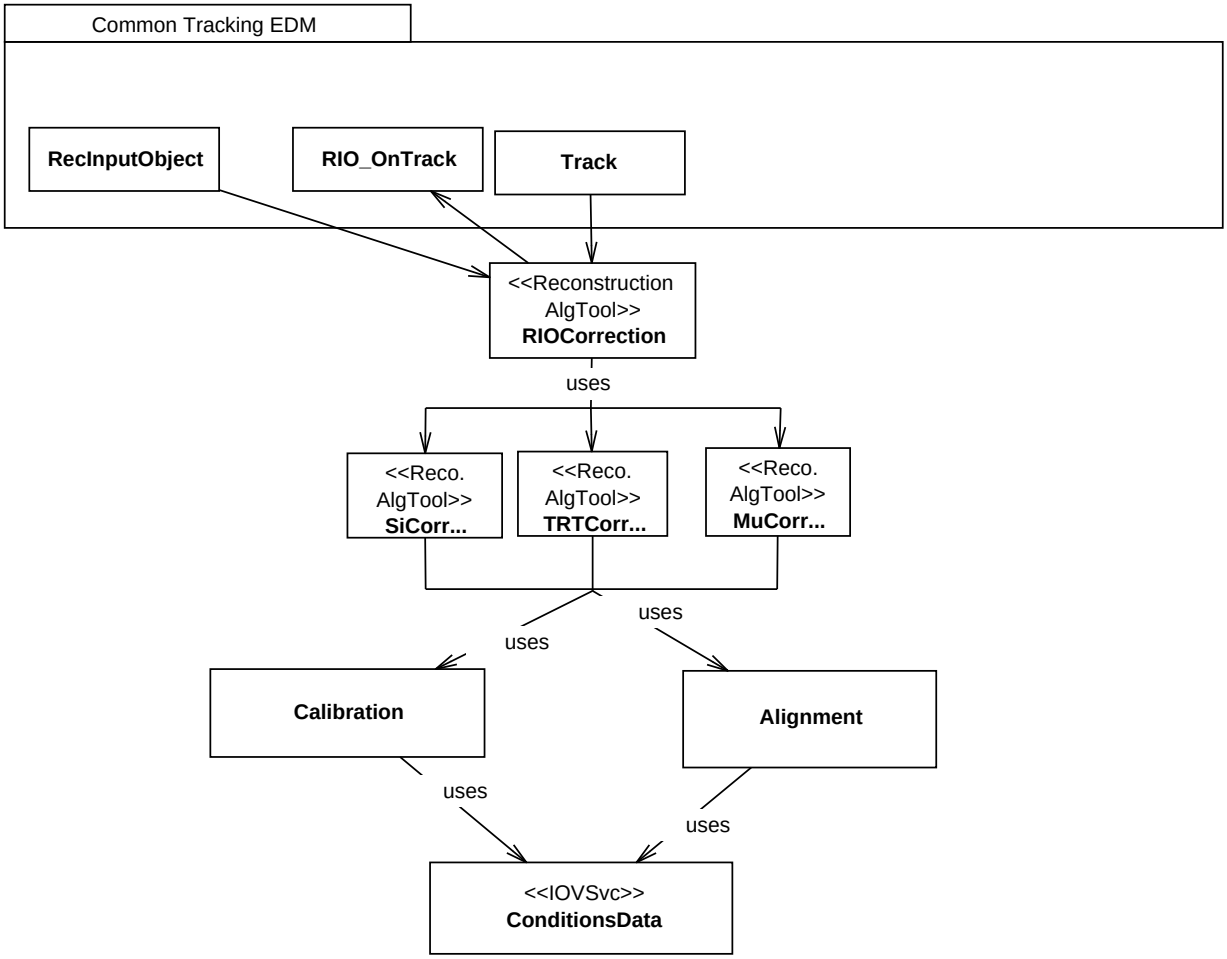

Figure 13: Tracking detector dataflow. RIO correction algorithms sequence and modularity. See text for details.

TrackFollower is to connect each track segments that could come from adjacent subdetectors for example. This module makes use of the TrackExtrapolator described above. Depending on the different tracking techniques chosen one could bypass the TrackFollower step if TrackCandidates were already made after the TrackFinding step. Similarly, if one were to do pattern recognition *a la* Kalman Filter, then one could directly make TrackCandidates without making TrackSegments, since one possible input objects to this module are the RIOs. Finally, there is a reconstruction Algorithm which outputs a collection of Tracks out of the possible TrackCandidates. The TrackFitter and AmbiguitySolver are used at this stage. We note that the AmbiguitySolver is used at different stages since it can depend on the available information at that stage.

Any time a set of Tracks is obtained, RIO_OnTrack can be produced, which contain the right uncertainty on the RIO position given the track direction. The description of this module is shown in figure 13. This RIOCorrection AlgTool might call different AlgTools pertaining to specific subdetectors. Whenever this AlgTool is called within the tracking chain, it will need access to the Calibration and Alignment services. The RIO_OnTrack AlgTool is called at least within the TrackMaker Algorithm, which will then refit the tracks given the produced RIO_OnTrack's to get the optimum TrackParameters. The RIO_OnTrack AlgTool can also be called at an earlier stage and some amount of iterations is typically necessary to arrive at a consistent set of tracks and RIO_OnTracks.

Figures 14 and 15 show examples of a possible sequence for Inner Detector Tracking, using the various modules described in the previous sections. The various reconstruction AlgTools have a label "-some_number" attached to them to show that the user can select the one he/she prefers. The important

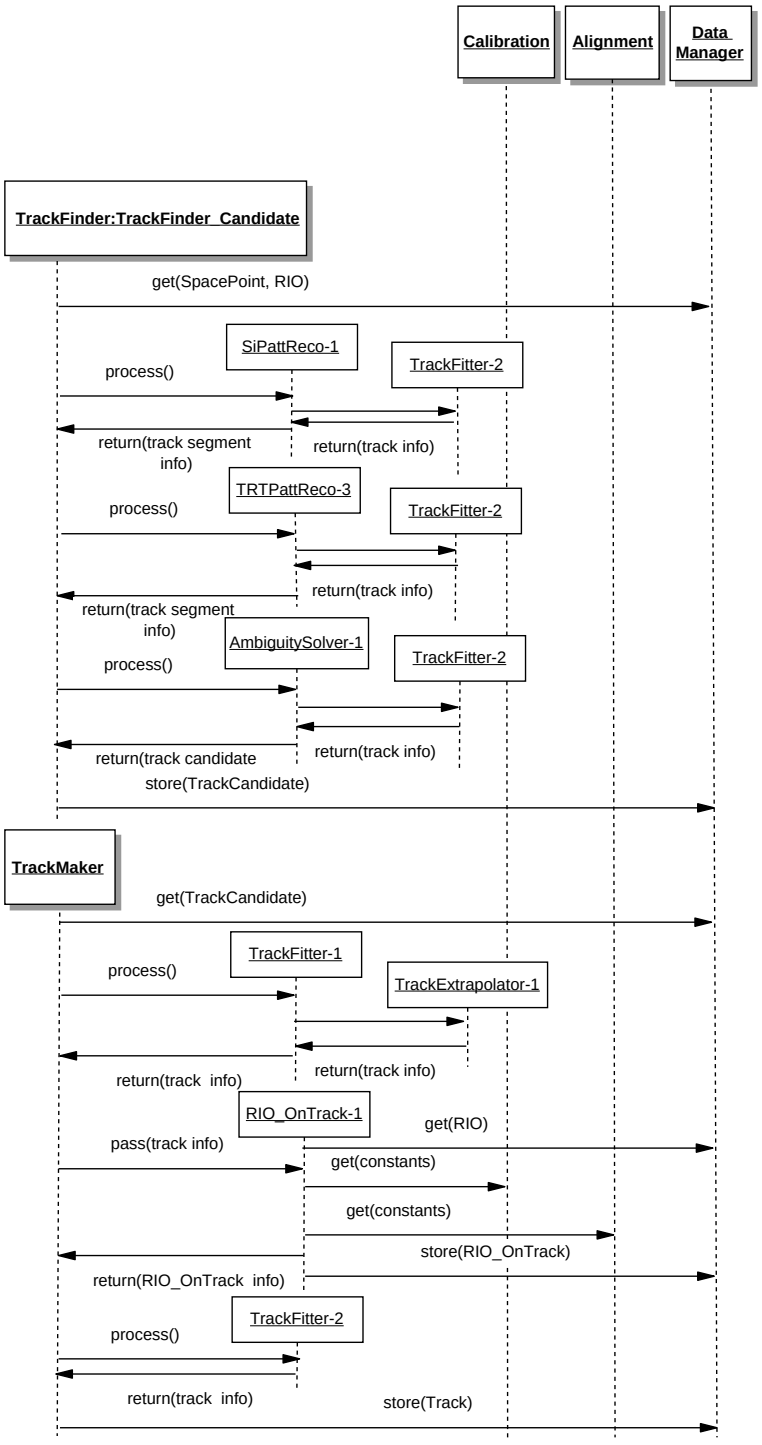

Figure 14: Inner Detector Tracking sequence. An example of a possible sequence for Inner Detector Tracking corresponding to the case of making Track Candidates at the track finding stage.. See text for details.

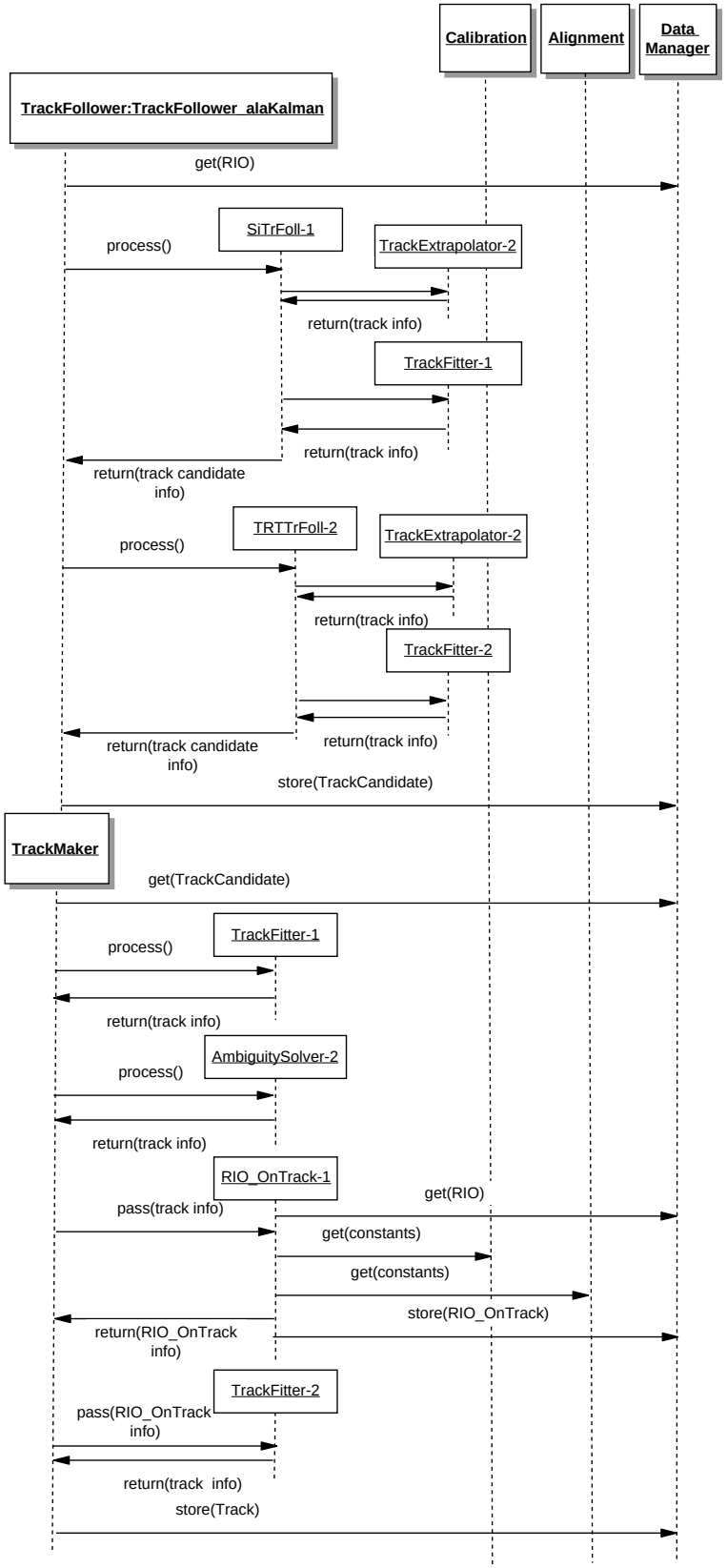

Figure 15: Inner Detector Tracking sequence. An example of a possible sequence for Inner Detector Tracking corresponding to the case of doing a Kalman filtering pattern recognition starting from the RIOs. See text for details.

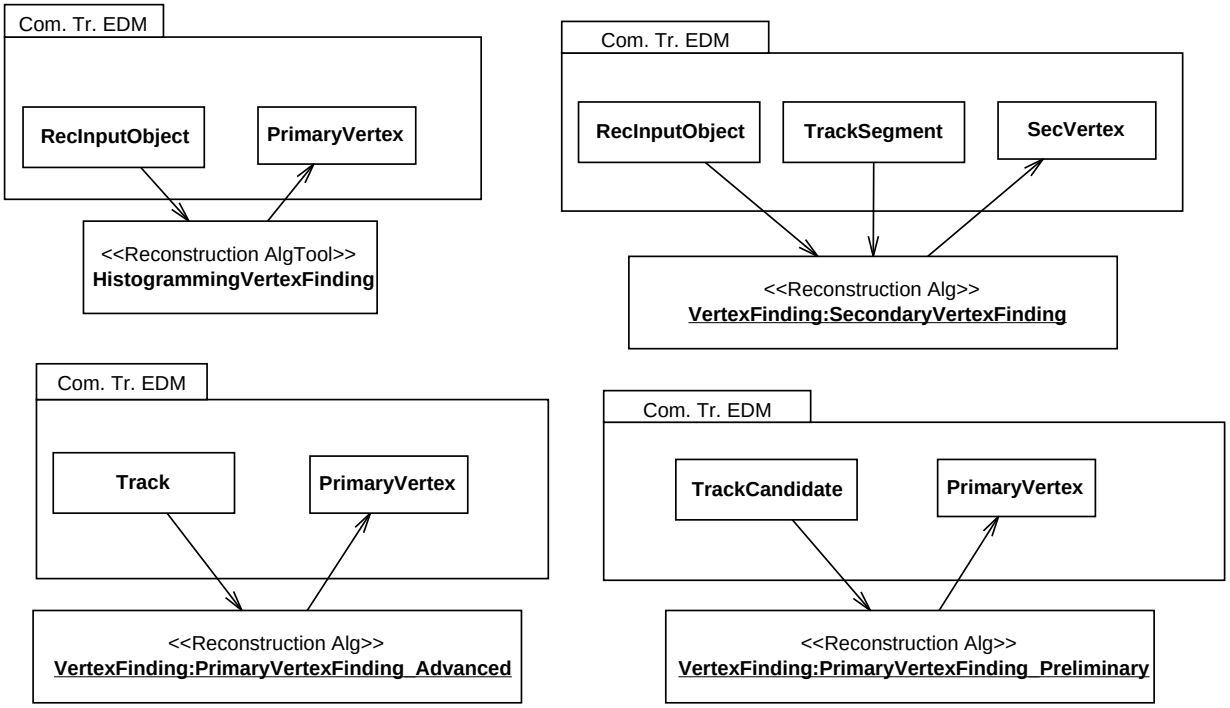

Figure 16: Vertexing algorithms sequence. See text for details on the different cases.

thing is that these AlgTools all have the same interface so that they can be called by the similar Reconstruction Algorithms (like TrackFinder, TrackFollower, etc.). There is the question of how those Reconstruction Algorithms are called and how the overall steering of the Reconstruction Algorithms and their Reconstruction AlgTools works precisely, this will be discussed on the section related with describing the scheme for doing full reconstruction or seeded reconstruction. The idea of having user-preferred reconstruction AlgTools is also connected to the idea of having a baseline for reconstruction, as described in details in section 2.

4.3.4 Finding and Fitting Vertices

Other objects created using tracking algorithms are the primary vertex of the event, and the various secondary vertices. Figure 16 show the dataflow associated with those modules while figure 17 shows the modularity. The VertexFinding module can be called at various stages during the reconstruction or even analysis chain to vertex tracks in order to make a primary vertex. For example, one could call this module toward the start of the tracking chain to put a loose constraint on the track candidates made at the TrackFinding and TrackFollowing stages, cutting down on the combinatorics. Similarly, one could call this module once all the final tracks are made and a final PrimaryVertex object is created to be used by analysis users. Finally, if an analysis user were to redo part of the tracking reconstruction, it would need to recompute an updated PrimaryVertex object. Another way to find a rough primary vertex could use an histogramming techniques based on some key RIOs. The VertexFinding module should also address both the issue of particle decays (Ks, lambdas, etc.) and also hadronic interactions with the material to produce a secondary vertex. It needs to have access to the TrackExtrapolator in the case of material interactions. The VertexFinding uses the VertexFitter AlgTool module to perform kinematic fits using the track inputs. Typically getting such a vertex will result in an updated set of TrackParameters. It is foreseen that there would be a new instantiation of the Track collection corresponding to the different vertices hypotheses.

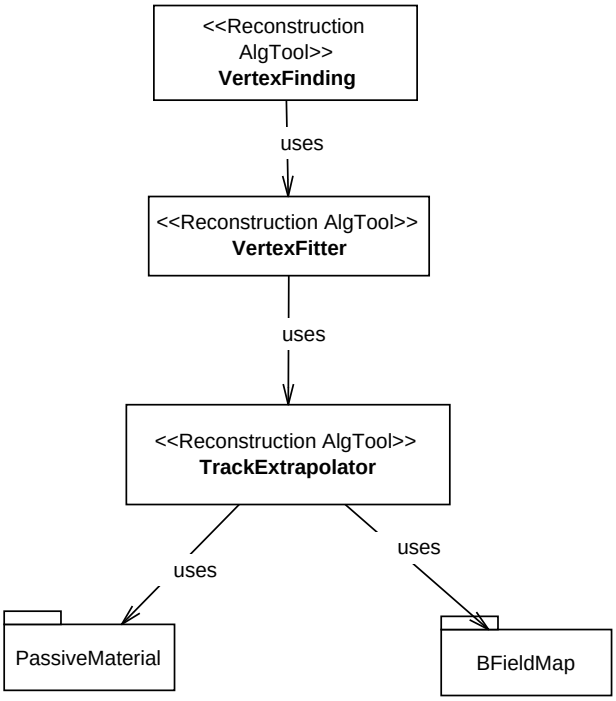

Figure 17: Vertexing algorithms modularity. See text for details.

4.3.5 Recommendations for tracking data flow

- R4.3-1 It is recommended to arrive at a consensus concerning a unique tracking EDM to be used by all tracking modules.
- R4.3-2 It is recommended that each subdetector reasses its situation regarding the implementation of the ATLAS Raw dataflow.
- R4.3-3 It is recommended that each subdetector produces a detailed design, based on the high level design presented in this section, for the subsystem reconstruction data model domain. This implies a strong collaboration between the Inner Detector subdetectors and the Muon subdetctor.
- R4.3-4 It is recommended that in the process of arriving to a detailed design, the existing algorithms like xKalman, iPatRec, MuonBox and Moore be mapped onto the high level design presented in this section.
- R4.3-5 It is recommended that once the existing algorithms have been mapped to the presented design, a set of baselines be decided by the subdetectors corresponding to different running conditions.

4.4 Combined reconstruction

Once the subsystem reconstruction has been performed, the results for the different subsystems will be used to build combined objects. Different particle species have specific patterns of interaction with the different subsystems. The combined reconstruction step thus consists of a series of builder algorithms, shown in Fig. 18 for the different subsystem patterns. Each builder takes as input the objects from the subsystem reconstruction expected to contribute to a given pattern, performs a match among them, and combines the kinematic information from the different input objects to calculate the kinematic quantities of the combined object. It should also calculate from the input objects detailed identification information which is needed for the successive particle identification step. The arrows in the example should be taken as an example, since they are derived from analysis of the current combined reconstruction. In general, a builder can take as input any data from the subsystem reconstruction domains and any data produced by

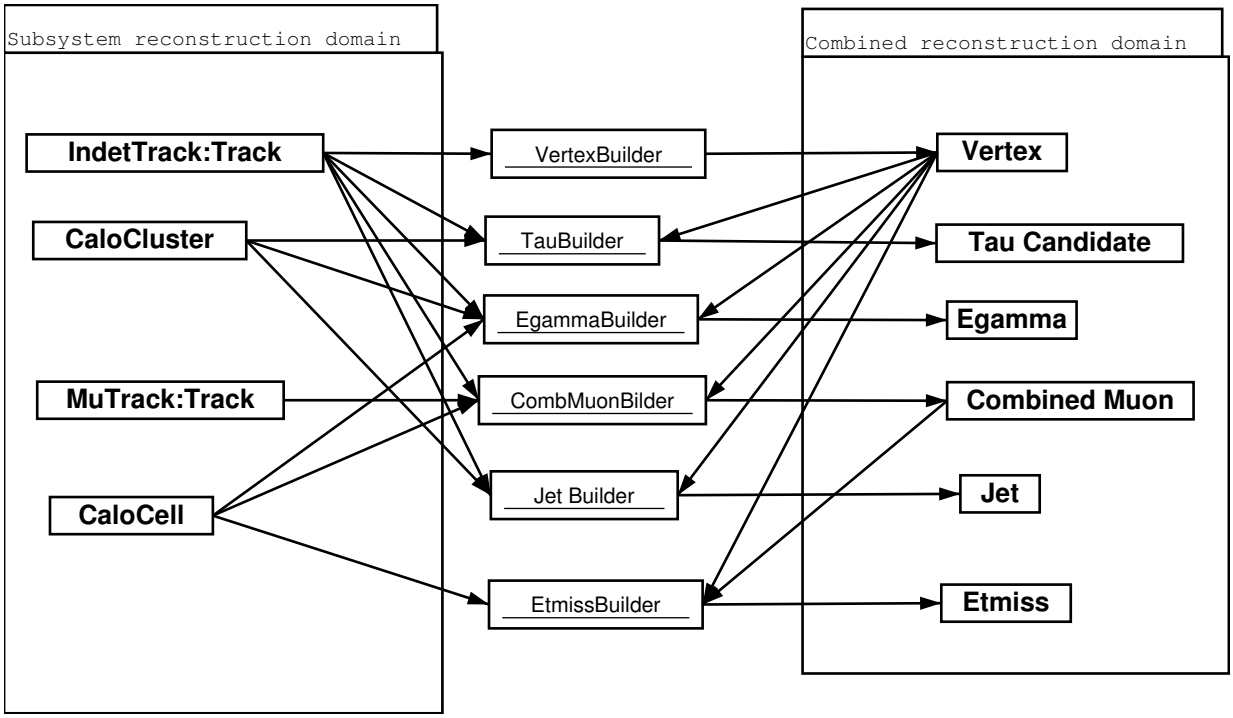

Figure 18: Overview of the combined reconstruction. Arrows indicate data flow. Boxes in between the EDM domains are algorithms. The ordering of algorithms from top to bottom is the order in which they should be executed.

a builder that is run before it. The ordering of the builders can be adapted to meet unforeseen use cases that arise in the future.

There is a general principle that the combined reconstruction should re-use the tools from the subsystem reconstruction as much as possible. Through collaboration with tools in these domains it is possible to create new data objects in the subsystem domains, such as refitted tracks.

4.4.1 Muon combined reconstruction

The goal of combined muon reconstruction is to combined all the measurement on a muon track to arrive at the ultimate precision for the particle parameters, making use of the accurate momentum and angle measurements from the muon spectrometer and impact parameters from the inner detector.

The data flow is shown on figure 19. A reconstruction tool is introduced TrackCombiner, which combines two track measurements in an optimum way, and outputs a new track. The tool can use two alternative strategies, embodied by two other tools: - only the TrackParameters from the tracks are used and combined with a standard matrix chi2 minimisation technique. This is fast and does not need to go back to the RIO_OnTrack's of the two tracks (provided that the helix parameters are given at the same point) [4]. - complete refitting of the RIO_OnTracks, which is not expected to gain on the resolution on the final track but to recover some of the tail by filtering more accurately the RIO_OnTracks [5]. The tool should also have the possibility to correct for the energy loss in the calorimeters either with a parameterisation, or with a measurement of the energy.

It should be noted that TrackCombiner could also be used to combined two tracks, say from the Silicon Tracker and TRT, or even an InnerDetector track and an e.m calorimeter cluster which can also

[4] This approach is named STACO in the Physics TDR

[5] This approach is named MUID in the Physics TDR, it has not been pursued since.

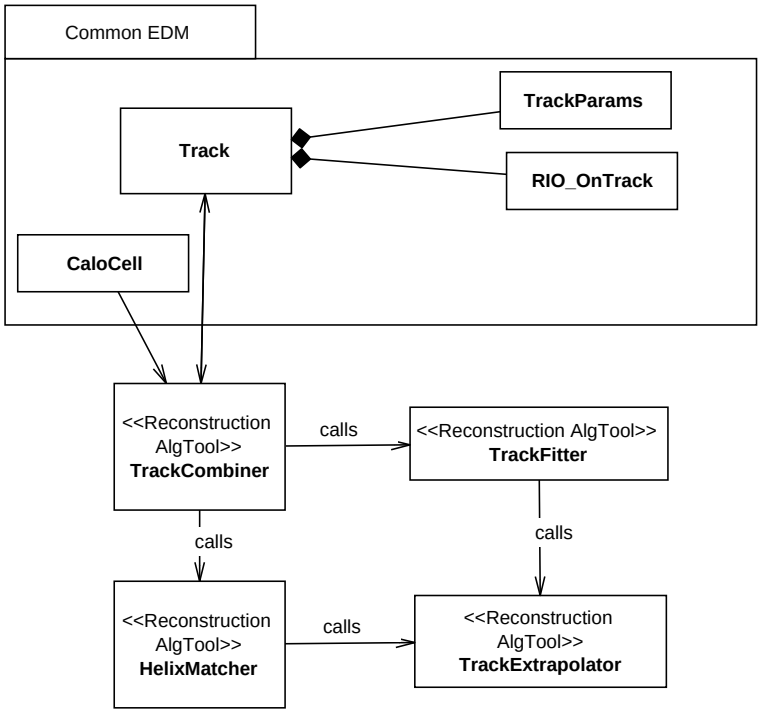

Figure 19: Combined Muon reconstruction dataflow. Unlabelled arrows indicate data flow.

be represented with an TrackParameter (with unknown curvature sign) [6].

4.4.2 E/γ combined reconstruction

The envisaged flow of the e/γ reconstruction is depicted in figure 20. This combined reconstruction step makes use of inputs from Calorimeter (EnergyCluster and CaloCell) and Inner Detector (tracks). The e/γ reconstruction requires the calculation of several variables that are used in the identification process. They include calorimeter variables such as isolation, shower shape; matching the track to the calorimeter clusters and calculating additional variables; handling conversions and finally applying simple cuts and/or calculating likelihoods based on some reference distributions. These are a sequence of independent steps which should be implemented as AlgTools to allow re-usability in several environments. A top level algorithm may drive these tools which updates the egamma objects. At the end of the sequence of execution of the tools, the egamma object, if accepted, is pushed into transient memory.

In certain instances, the execution steps with egammaRec may require re-fit of the sub-system outputs. For example, the track may need to be re-fit to correct from brem effects. While the track fitting itself is a part of the tracking domain, it may well be callable from the combined reconstruction domain. This should result in the creation of a new track in transient memory and the old track must not be overwritten. The egamma object would then point to this new track.

In a parallel example, the e/γ reconstruction may work off an input track as a seed. The clustering algorithm may be called to reconstruct the region around the extrapolated track point to the calorimeter. The clustering algorithm belongs to the calorimeter reconstruction domain - but callable from e/γ reconstruction and results in the creation of a new calorimeter cluster object in transient memory.

The calculation of calorimeter variables related to the egamma object, such as isolation, energy leakage and such, should be isolated to a different object (depicted as EMShower in the figure). The egamma object simply holds EMShower by value. This allows easier maintainability of the code in the egamma class. Similarly, the variables calculated as a result of the match between the track and the cluster is held

[6] this is one of the approach for brem recovery in the Physics TDR

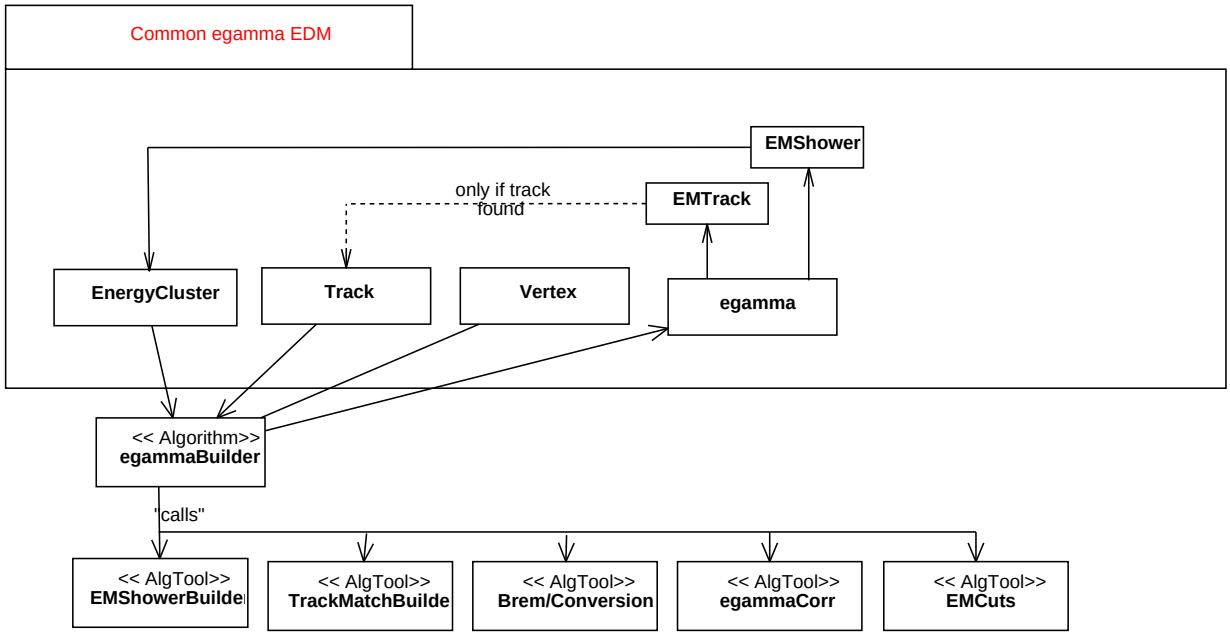

Figure 20: The electron/photon (e/γ) reconstruction utilising inputs from various sub-detectors. The resulting egamma object holds the output of each of its internal reconstruction steps which are navigable to its parent objects.

in an EMTrack object.

The top level algorithm can differ if the reconstruction is done using tracks as seeds or CaloClusters as seeds. They are also expected to differ for reconstruction in regions of interest. However, the tools that perform the real work are common under both modes of operation.

The egamma candidate objects are then used as inputs to the analysis phase where they are further classified as electrons or photons. Egamma candidates are considered as basic objects to any physics analysis that requires electrons or photons. Hence the cuts applied at this stage should not be physics dependent cuts, which is applied only during the next (analysis) phase of reconstruction.

4.4.3 Jet and Tau Reconstruction

The Jet and Tau reconstruction should work in an identical way to e/γ reconstruction, refer to the e/γ section for details. It uses EnergyClusters produced during the sub-system reconstruction phase and possibly other sub-system data model inputs. If it chooses to use additional sub-system inputs in addition to the EnergyCluster, such as adding CaloCells, then the Jet object must be navigable to these objects as well. A top level algorithm drives the various steps (implemented as tools) of the jet reconstruction. Note that the tools shown in the figure suggests one of the possible configurations, including allowing re-clustering if necessary. The set of tools can be chosen and configured at run time and steered by a top level algorithm, allowing the flexibility to develop different tools and compare relative performances. These tools should also be usable in the HLT environment and must follow the recommendations suggested in the steering section.

The major difference between the recommended design and the current implementation is the absence of ProtoJets. In effect, this functionality has been merged with EnergyClusters described in section 4.2.3. Jet-finding algorithms can either accept pre-clusters (via EnergyClusters) or directly any object via its 4-momentum interface (see section 3.2.1 for details). The output of the Jet reconstruction is a Jet object which in the current implementation is a "CombinedJet". This output Jet should be able to navigate back to EnergyClusters and other relevant objects that were used in the Jet Finding process. The use of EnergyClusters as output of the calorimeter clustering establishes a common input to all relevant combined

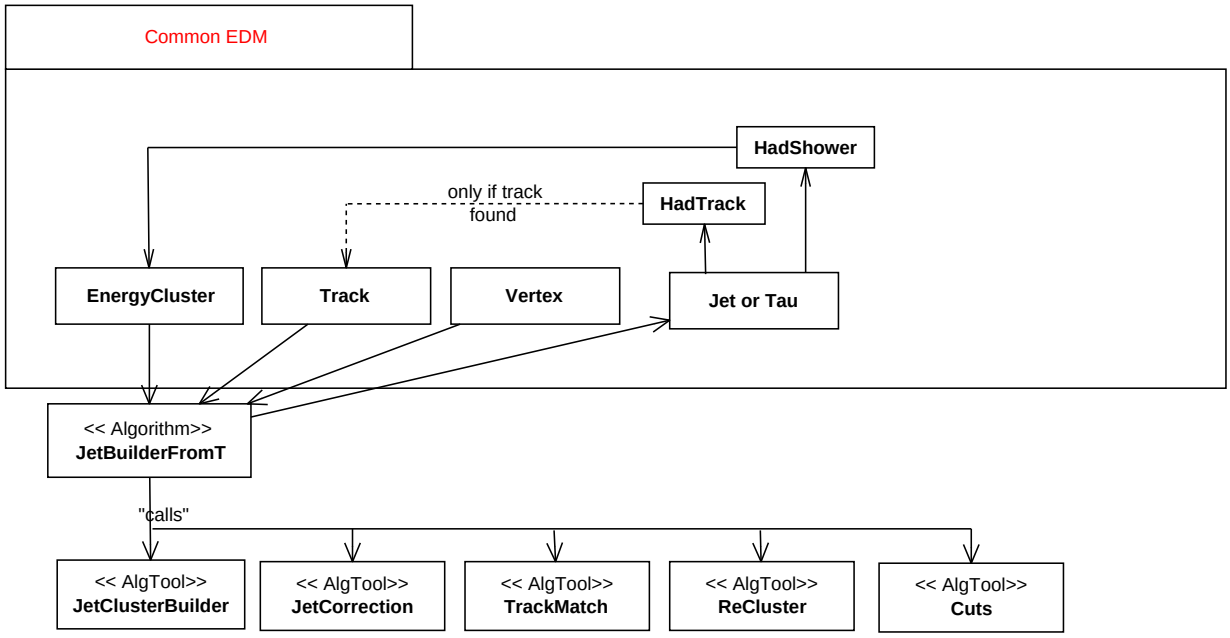

Figure 21: Data flow for the Jet and Tau reconstruction. Arrows between EDM and algorithm represent input-output, between algorithms represent calls.

reconstruction such as e/γ, jet, tau and Missing E_{T}.

4.4.4 Missing E_{T} Reconstruction

The goal of missing E_{T} is to reconstruct the invisible p_{T} of an event from the best estimate of the visible p_{T}. The scheme is shown on figure 22. To first order, this can be accomplished from the sum of the p_{T} of individual cells, taking into account the fact that the main vertex is not necessarily on the detector symmetry axis. However a number of refinements are needed to get the best resolution:

- Take into account high p_{T} muon, by including the best estimate of the muon parameters from the combined muon reconstruction and avoiding double counting of the muon deposit in the calorimeters
- Take into account high p_{T} electron and photons, filtering out the corresponding cells.
- Apply a threshold on the cell energy for which the cell per cell estimate of the noise for the current luminosity is needed.
- Apply weights which might depend on the existence of nearby jets

A more involved approach (usually called “Energy flow”) would attempt to use the tracking information not just for the high p_{T} lepton.

4.4.5 Recommendations for combined reconstruction

The following recommendations are made for the combined reconstruction.

- Combined reconstruction should re-use the tools from the subsystem reconstruction as much as possible.
- The existing combined reconstruction should be examined with a view to mapping on to the modular design described here.

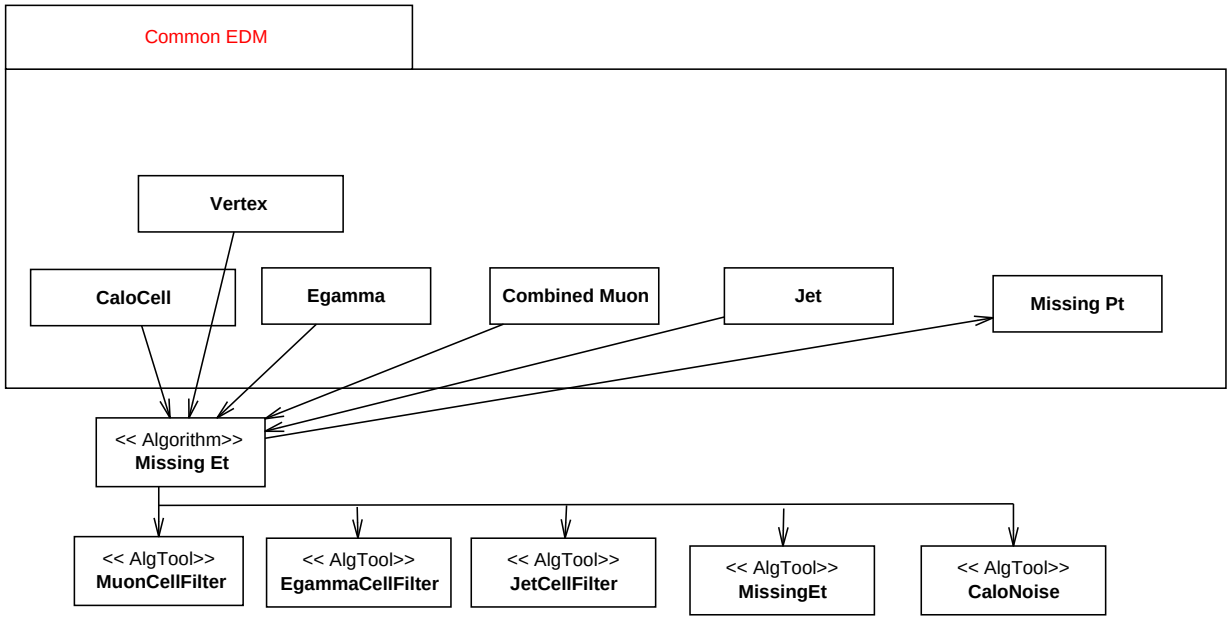

Figure 22: Missing E_{T} reconstruction dataflow.

4.5 Analysis preparation

For the purposes of this report, the process of "analysis" has been defined in the following way: making analysis-specific cuts to existing objects, not making new objects. It is therefore necessary to have a processing step that goes from the Combined Reconstruction objects up to the input objects used by user analysis. This section describes this step, which is best described as "preparation for analysis" and the new objects and algorithms are part of the analysis data domain.

Users will always be free to re-run parts of the reconstruction before doing their analysis. The reconstruction software will not preclude this, but it is a matter for the ATLAS computing model to provide the infrastructure to make it possible.

The analysis data domain[7] is the minimal set of data on which a physicist can perform a meaningful analysis. The data content corresponding to this definition will evolve dynamically with the experiment. In the initial phase of running, with limited understanding of the detector performance, all attempts at analysis will require access to essentially the full output of the combined reconstruction, with navigation tools to allow the recovery of information at the subsystem reconstruction level.

With the evolution of the experiment, when the performance of the different particle identification tools will be well understood, the definition of the analysis data will asymptotically converge to a set of particle objects, roughly consisting of i.e. four-vectors plus additional identification information which will allow the analysis some limited amount of choice about the way different objects should be classified. Global event information, such as $E_{\mathrm{T}}^{\mathrm{miss}}$, $\sum E_{\mathrm{T}}$, circularity, and a list of reconstructed vertexes is also part of this final set of of objects on which the kinematic selection cuts should be applied.

In addition to the bare particle objects, a strong requirement for a meaningful analysis is the presence in the analysis domain of additional information. This is necessary to validate the events at the end of the selection procedure, and to be able to reevaluate at least partially the particle identification information. The definition of the exact nature of this information will be the result of the initial detector studies performed on real data and a compromise between the need for flexibility which is a primary analysis requirement, and practical considerations. The list of tracks in the inner detector, with a reduced set of

[7]NB The analysis data domain is purposely distinguished from Analysis Object Data "AOD", which refers to a stream in the computing model. It's content is defined according to the analysis needs at the time it is produced, so it might very reasonably include data which has been classified into other domains, and data conditional upon certain cuts or conditions.

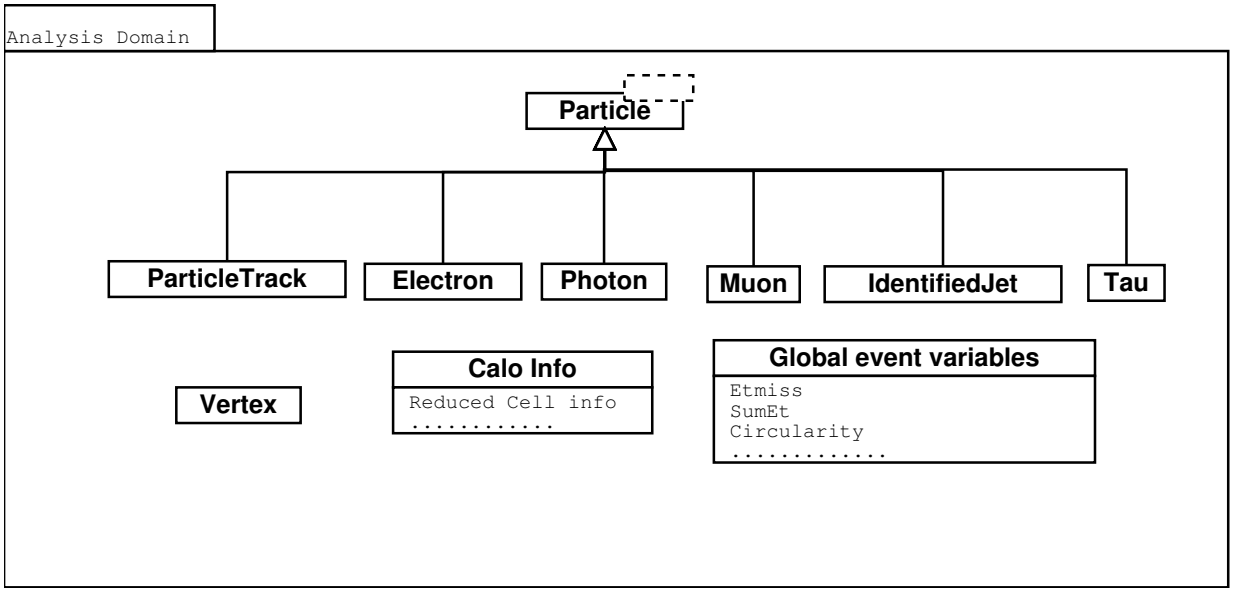

Figure 23: High level view of event data objects belonging to the analysis domain.

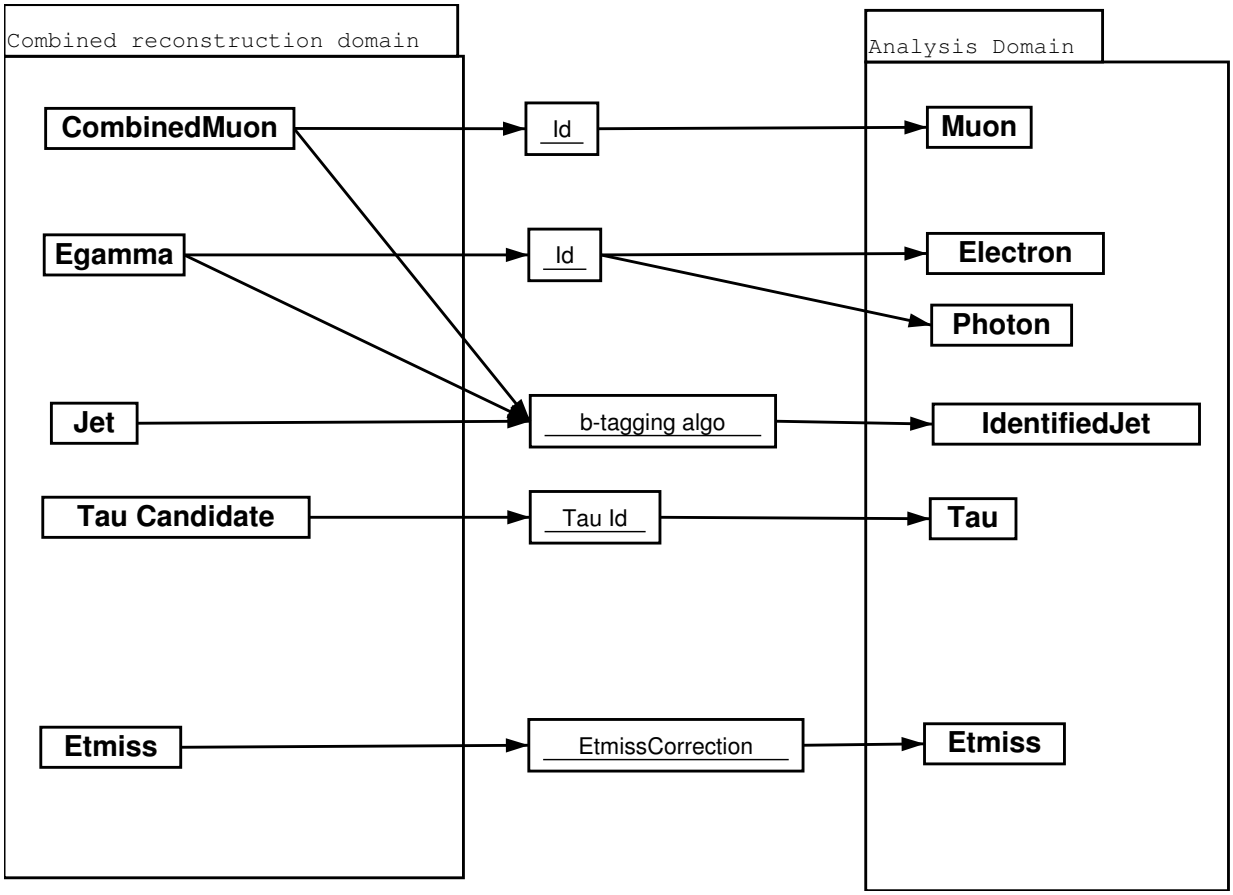

Figure 24: Overview of the data flow in combined analysis. The arrows represent data flow. Boxes in between the EDM domains are algorithms.

attributes, will be part of the analysis domain as the particle objects on which b-physics analysis will be developed, but will also be necessary for most other analysis as a tool to cross-check the validity of complex event topologies after the kinematic selection has been performed. Some compressed cell-level information from the calorimeters is also most likely needed at analysis level.

The flow of data, from the ESD to the analysis objects, consists in a set of identification algorithms which, using information from both the output of the combined reconstruction and from Detector Objects, builds the four-vectors corresponding to basic/average particle identification prescriptions. The algorithms, as well the prescriptions will again be the results of the first stage of detector studies, and may vary depending on the data stream considered.

The result of the particle identification stage will be a set of identification variables which each object will have. To give an example, a jet will have different flags, which will allow to classify it as a light-jet, a b-jet or a τ. In order to achieve maximum flexibility, ambiguities will not be solved at this stage, but it will be the task of the user to decide, using the identification flags, the correct identification of a given object. As an exemplification, a τ candidate can appear in the analysis domain both among the jets and among the τ candidates. In order to avoid double counting, appropriate association tools are needed, which allow, for each object in a list of particle candidates, to identify objects in other lists which are built out of the same objects at subsystem level.

While retaining the flexibility for users to change the identification, a default identification with ambiguity solving[8] should be provided, with two purposes, one to provide a baseline to the user. the other to be used internally for building the final $p_{\mathrm{T}}^{\mathrm{miss}}$ variable. In fact the corrected $p_{\mathrm{T}}^{\mathrm{miss}}$ variable will rely on the 'final' particle identification for all objects, and the $E_{\mathrm{T}}^{\mathrm{miss}}$ correction algorithm will receive input from all the other objects in the analysis data domain.

4.5.1 Recommendations for analysis preparation

The recommendations for combined reconstruction in section 4.4.5 apply equally in this context.

- It is also recommended that work begins on the analysis domain itself, as this is necessary to get user feedback on the tools and data classes described above. It is beyond the scope of the RTF to make further design recommendations about the analysis domain, but it is noted that presentations made at the RTF open meetings did address the software needed for analysis, and could be taken as a starting point. See for example [14].

4.6 Integrating fast simulation and full reconstruction

This section describes how the fast and the full reconstruction can be integrated. It has been written in terms of a general fast simulation and reconstruction package; although this is implicitly ATLFast, the concept of fast simulation and reconstruction has not been limited to what is currently available from ATLFast. This is why some of the examples are not possible with the current software, and the general term "fast simulation" is used throughout the text.

4.6.1 Use cases

Use case 1
To be able to write analysis code that works on the simplest data objects (e.g. from fast simulation) and then run on other source of data objects without changing code. In other words, objects from fast simulation, HLT, and full reconstruction have the same basic interface, but each may extend this when they can provide additional attributes for the data. Note that the reverse use case (to be able to take code that runs on objects from full reconstruction and run it on fast simulation objects) is inherently impossible, because the full reconstruction output contains details that fast simulation is not intended to provide.

[8] See the navigation section for ways to record ambiguities as a mutual exclusion between data.

Use case 2
There is a "baseline offline reconstruction" that works off fully simulated Reconstruction Input Objects (RIO) and another "baseline fast simulation reconstruction" that works off fast simulation input, but the latter can use pieces of the offline code where possible. In other words, the subset of reconstruction algorithms which only need 4-vector information to work, should be reusable in the fast simulation reconstruction.

4.6.2 Problem analysis

First, it is clear that a fast simulation such as ATLFast combines fast simulation of both the detector response and the reconstruction. It makes some short-cuts between the two, since it was never intended as a complete simulation of the detector response. Therefore it is not possible to expect that a fast simulation would simulate data at the RIO level.

Secondly, it is noted that the reconstruction is not hierarchical (see the data flows in section 4). In other words, steps tend not to build solely on the data objects produced by the previous step. Typically, each step in the reconstruction takes the output of the previous step, but then navigates back to the RIO data associated to it, or geometrically near it, and uses this to perform further reconstruction. This makes it impossible for most reconstruction algorithms to work on objects produced by fast simulation, since the RIO level of data cannot be produced by fast simulation. This applies to the both the subsystem and combined reconstruction domains.

Sometimes this may be addressed by splitting algorithms into two parts, one with the more basic reconstruction functionality, and the other with the fine-tuning required for the best reconstruction. This approach maximises the amount of reconstruction software than can be re-used in both fast and full simulation reconstructions.

Examples:

- Primary vertex finding can work from the tracks alone and only needs the clusters in a subsequent step to refine the vertex. Therefore, separate the primary vertex fitter (which works only on tracks) from the code to subsequently refine the vertex by accessing the clusters on the tracks, so one could configure at run-time a reconstruction that does not need clusters by only choosing the first, basic module.
- Track finding can work off clusters, but may need to go one step back to RDO to reexamine pixel clusters in the light of the tracks that have been found; to split V-shape clusters for example. *Note that this example goes beyond the current capabilities of ATLFast.*
- Clustering algorithms, such as KT and cone algorithms, require only the 4-momentum information without the knowledge of the elements it is trying to cluster. This KT tool can be used by reconstruction of both fast and full simulation but algorithms that navigate back to calorimeter cells would be specific to offline.

In the analysis domain, there is more scope for common classes. There is still a problem that some analyses will want to go back to data objects in the subsystem reconstruction domain, but there are also a large number that do not, for example the many analyses being developed with ATLFast at the moment. Those analyses which need full reconstruction can be written with this constraint, but those which do not can be written using a reduced subset of data classes that can be produced by either fast simulation or full reconstruction. The classes that fall within this subset will have to be made clear to developers of analysis and reconstruction software.

Many objects will contain links to other objects not produced in fast simulation. This can be handled by empty link containers, as described in section 5 where navigation is discussed. The algorithms which are intended to work in both fast and full simulation must handle this in a common way. Differences should be accommodated in separate algorithms, external to the common algorithms.

In conclusion, it does not seem very useful for fast simulation and the full reconstruction to share common to interfaces to *all* data objects in the subsystem and combined reconstruction domains. There

are however some places where this could and should be done. The level in the data model at which fast simulation and full reconstruction should generally provide data with a common interface is the analysis domain.

For the longer term, it is understood that a "medium speed" simulation, using parameterised Geant4, is being developed. This is somewhere between fast simulation and full reconstruction, simulating the detector response and producing the RIO data in full, but without the rich complexity and characteristics of full Geant simulation or the real data. It would allow a mix and match approach between fast and full simulation for different detectors. As such it is a very useful tool which complements the current too approaches. A great advantage of this approach is that it would automatically share the reconstruction EDM and algorithms with the full simulation and real data.

4.6.3 Recommendations for integrating fast simulation and full reconstruction

- Fast simulation and full reconstruction should have common interfaces to data classes in the analysis domain.
- Where possible, separate algorithms into two parts: the first part is the code that requires only basic information which is available even in reconstruction from fast simulation; the second part localises the code which needs to access details only available in the full reconstruction. This will maximise code sharing between the reconstruction of both fast and full simulation.
- Simulation based on fast parameterisations inside Geant4 should work with the full reconstruction just as well as full simulation does.

5 Data Object Navigation

Data object navigation is central to the event reconstruction - one may even argue that establishing object associations and navigating them is what reconstruction is about. C++ provides pointers (and references) to implement associations. For example a track object may have a list of pointers to the hits it was reconstructed from

```
class Track {
   list<const Hit*> m_hits;
   ...
};
```

In ATLAS, the Track author will probably replace the list of hit pointers with an ElementLinkVector to allow the associations to be persistified [4][5][6]

```
   ElementLinkVector<HitCollection> m_hits;
```

As customary, in this section, we will discuss and provide examples of both unidirectional and bidirectional associations. We will also distinguish between causal associations that can be implemented using simple pointers or ElementLink and non-causal associations that we propose to implement using separate association objects.

5.1 Causal Associations

The Track⟶Hit association above follows the natural reconstruction data flow: hits needs to be decoded before any track can be reconstructed[9]. In this sense the Track⟶Hit association is a *causal* association. Notice also that often causal associations are, in UML terminology [12][13], Composition relationships,

[9] Notice how this causality relationship is context dependent: in event simulation, tracking is performed before digitisation

and by definition a Composition relationship is causal. In other words, a Reconstruction Track can not exist without the Hits it was made from.

Causal associations are, by definition, natural to establish in the reconstruction process: in a tracking algorithm the hits associated to a candidate track are added to it while the Track object is being constructed and before the Track is `record`-ed to the Event Store.

5.2 Reverse Navigation and Association Objects

By contrast consider the situation in which the second iteration of hit decoding is using the track association information, e.g. to improve the drift time resolution. In this case, each hit is associated to the track it was used to form. We call the non-causal association from an object created earlier in the reconstruction process (the Hit) to an object created later (the Track) a *reverse* association.

A reverse association requires in general a separate *association object* to implement it. Coming back to our example, when we refine a hit using its associated track info, we can not simply put the associated Track pointer in the original Hit object, for a number of reasons:

- A hit does not necessarily belong to any track. In certain cases the event may even contain no track at all and still have decoded hits.
- When the hit is originally recorded in the event store, no tracks have yet been reconstructed. Only after the second hit decoding iteration the associated track pointer can be set. Although it is technically possible, modifying a data object already stored into the event store is strongly discouraged by our EDM design philosophy.
- A hit may be associated to several track objects, either because it is shared among them, or because the various tracking iterations change the hit assigned to each track, or because several tracking algorithms have been run on the same hit data for comparison.

To implement reverse associations like Track⟵Hit we propose to create separate association objects. For our example a class HitTrackAssoc will contain pointers to the original Hit and the "backward" associated Track, plus, when relevant, any property of the association:

```
class HitTrackAssoc {
   Hit* m_rawHit;      //or ElementLink<HitCollection>
   Track* m_track;  //or ElementLink<TrackCollection>
   double m_dca;       //distance of closest approach (or chisq, or ...)
   ....
};
```

since the HitTrackAssoc objects are created during the second tracking iteration both pointers above implement causal associations. If an Hit is not used to form a Track no corresponding HitTrackAssoc object will be created. Finally should a Hit object be associated to multiple tracks the second iteration decoding will simply create multiple instances of HitTrackAssoc pointing to the same "raw" Hit.

5.3 Bi-directional associations

Bidirectional associations in the simplest cases can be implemented using an associative container that adds "reverse navigation" capabilities to an existing causal relationship. In the Track⟷Hit case, a HitTrackMap defined as

```
typedef std::multimap<const Hit*, const Track*> HitTrackMap;
```

allows to find the tracks that contain a given hit, while of course the hit belonging to a given track can be directly found following the hit pointers in the track. If the associations have any property than it is

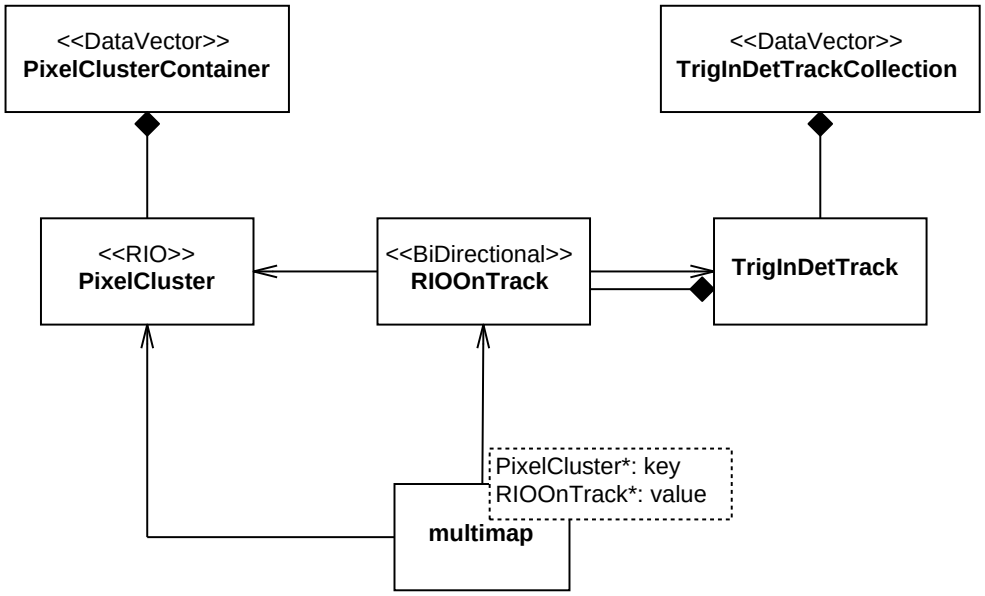

Figure 25: The bi-directional association example

convenient to express them in terms of associations objects containing back-to-back pointers to the two associated objects and the properties of the association, like the `HitTrackAssoc` object of the previous section. In this case the "reverse navigation" container should index the association objects rather than the original ones:

```
typedef std::multimap<const Hit*, const HitTrackAssoc*> HitTrackMap;
```

5.3.1 `BiDirectionalPrototype`: a Complete Example

The package `Reconstruction/RecExample/BiDirectionalPrototype` contains a complete and hopefully realistic example of how to establish and use bi-directional associations in Algorithms. The prototype example is illustrated in Figure 25. The class `FillingAlg` creates some `PixelCluster`s and makes some `TrigInDetTrack`s using them. It then creates the association objects `RIO_OnTrack` which are added both to the Tracks and to a

```
std::multimap<const PixelCluster*, const RIO_OnTrack*>
```

The second algorithm in the package, `NavigatingAlg`, shows how to use the RIO_OnTrack to navigate from the Track to the RIO and vice versa.

5.3.2 Truth Navigation and Bi-directional associations

Truth navigation is perhaps the most important application of bi-directional associations at the time of this writing: for simulation studies it is vital to be able to associate a given simulation object (e.g. a simulated RDO) to the HepMC particle(s) that produced its signal, and vice versa to know all data objects created while simulating the propagation of a given particle.

Currently, for the Inner Detector, there is a map between the RDO identifier and an InDetSimData object, part of the MCTruth domain. The InDetSimData object contains a vector of pair of the particle bar code and the energy associated with this particle contributing to this RDO. The InDetSimData object also contains a coded word corresponding to various flags associated with the simulated RDO (noise, belowThreshold, etc.). There are subdetector helpers used to filter the coded word.

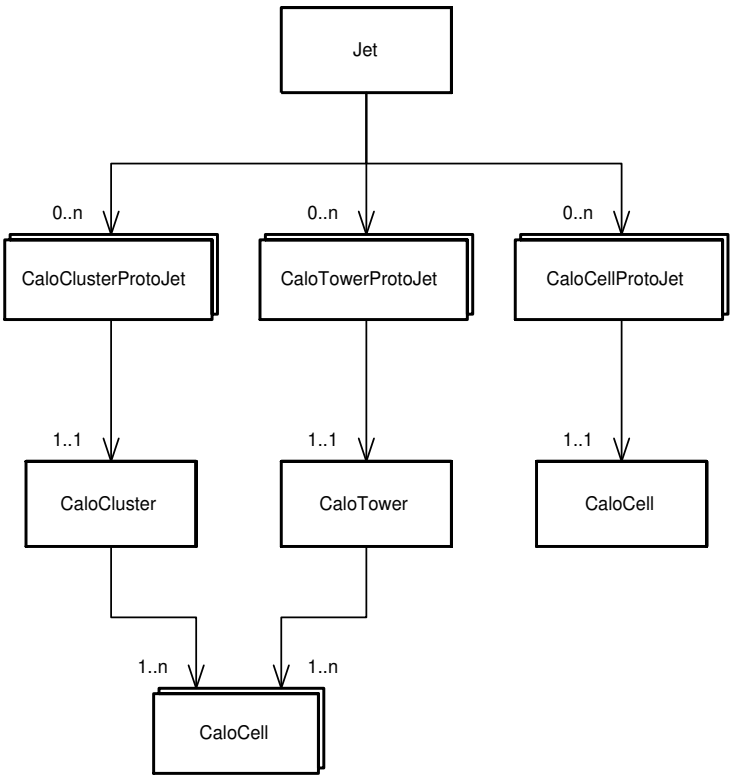

Figure 26: Calorimetry navigation object diagram[10]

Following the above discussion, there also needs to be a map between a given particle and which simulated RDOs it contributed. There should be an associative object that provides these two maps.

In the case of the Inner Detector, the RDOs are part of the Raw EDM domain and typically are not navigated to very often. The RIO are the beginning of the reconstruction chain. There needs to be a similar MCTruth data object for the RIO that will also contain the information related with the different bar codes and their contributed energy to this RIO. The MCTruth domain algorithm which would build this object would make use of the InDetSimData object to create this object and would also build an association object between the RIO identifier and the created RIO MCTruth data object. Similarly other objects in the reconstruction chain, like Track, need their associated MCTruth data object and association object connecting them to the track. There might also be the need to have the reverse mapping: knowing that a particle contributed to which tracks, this reverse map is to be contained in the associative object as well.

5.4 Event Navigation

The causal associations among event objects form a directed graph. Navigating the graph, for example to obtain the list of cells in a jet is greatly simplified if the associations are implemented using the composite navigation scheme already used by the calorimeter reconstruction [10]. In a nutshell, a `NavigationToken` is sent down the graph to gather associations to objects of a given type (e.g. CaloCell). Each node implements the `INavigable` interface through which it can add associations to the token or pass it down to its child nodes. The composite graph is illustrated in Figure 26 where a query is made to the `Jet` about the list of CaloCells by sending down a `NavigationToken<CaloCell>`. Note that the Jet class itself has no knowledge about CaloCells, but knows only its immediate children: one of the many possible `ProtoJet`.

At the end of this process the token contains a list of all the required associations.

The present calorimeter navigation scheme[10] requires each data class to implement the INavigable interface. Noting that this can be burdensome to the data object developers, the RTF recommends that the present calorimeter navigation scheme be generalised. The Navigation must provide a generic implementation of the INavigable interface to ease the job of the data object developers. It must also deal with just not weights (type double as implemented in the current navigation) but arbitrary types attached to each association. Finally, it must allow for conditional queries in a generic way. The RTF has developed prototype models to ensure that the INavigable interface can be implemented in a generic way. It is now recommended that this prototype be quickly extended to provide a generic scheme for composite navigation for any ATLAS reconstruction data classes.

5.5 Recommendations on navigation

- Use C++ pointers to implement causal object associations.
- When required by persistency, replace the C++ pointer with a DataLink or ElementLink [6]. In any case the client interface should be in terms of plain pointers.
- Use association objects such as HitTrackAssoc to implement backward associations and/or to "attach" properties to the association itself.
- When the use cases point to the need to navigate through multiple layers of the tree of relationships (Jet to Cells), use the composite Navigation scheme instead of plain pointers to reduce physical couplings and enforce a coherent interface.

6 Steering

6.1 Introduction

This section is concerned with the concept of "steering", which is the control of the flow of the reconstruction. In the HLT this also involves taking decisions to continue or reject the event at key points in its reconstruction and classifying events. The aim is to decide how this can be done in a standard and flexible way for offline reconstruction and HLT.

A detailed analysis of the steering problem was given in the second interim report of the RTF. Since then the RTF has investigated some designs and made a prototype to tackle the problem of seeding, but was unable to conclude this work in time for the final report. This aspect of the steering design has therefore been deferred to a later report, outside the scope of the RTF.

Some general recommendations from the second interim report on the structuring of software are retained below. A following section deals with how to chain algorithms together via their input and output collections.

Note that the recommendation to structure the reconstruction software using AlgTools raises the following issue. AlgTools and other services which can be initialised on demand cause a potential problem for the LVL2 trigger. In the LVL2 environment, it is only possible to access configuration databases at the start of run, not during event processing. It is also not possible to perform lengthy initialisation procedures during event processing because this will cause LVL2 to time out the event. These requirements are potentially at odds with the "on-demand" model of Gaudi tools and services. The architecture team should consider how to reconcile these issues.

6.1.1 Recommendations on steering

- The top level Algorithms will be ordered by use of the Sequencer or the ApplicationManager's TopAlg list, or an alternative steering scheme for the HLT. Top algorithms must be provided for both types of

[10] P. Loch et al, document in preparation.

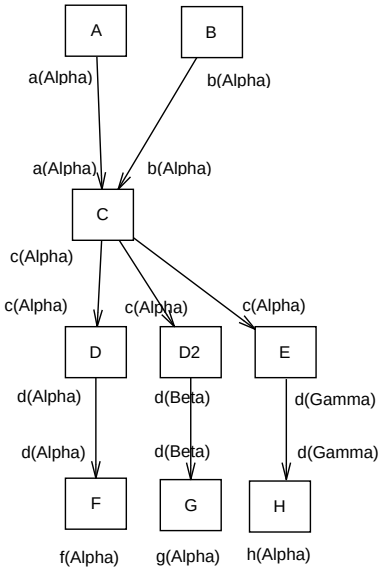

Figure 27: Schematic representation of a network of algorithms. The different algorithms are indicated with capital letters, with the different instances indicated with a number appended. The different input/output objects are indicated with lower case letter, with the key in parenthesis.

steering. To avoid duplication of code, their content should be limited to code that must be different for HLT and offline use of the algorithm.

- Reconstruction software should be structured as follows. At the top level, an Algorithm should provide the reconstruction for each major stage in a domain. This Algorithm should contain no algorithmic code, as explained in the first recommendation, but will accomplish its task by collaborating with specified modules, in the form of AlgTools or sub-Algorithms, which are configurable properties of the algorithm. AlgTools/subAlgorithms may of course call other AlgTools if necessary. In the simplest case this might be no more than a sequencer for AlgTools - if this proves to be a common pattern then it should be provided by Athena. But in many cases the execute methods of the AlgTools may well require arguments, so the output from one AlgTool can be the input to another. This can best be achieved by using AlgTools.
- Different instances of AlgTools should be used when they have different properties, rather than embedding the different configurations inside the tool and letting it decide how to react to the data. This allows the user to define unforeseen configurations, and as many different configurations as she likes for use in different places, so it is more flexible.
- At an even lower level in the structure, there will be basic utility classes which are not specific to ATLAS and do not interface to any of the framework services. These typically provide mathematical functions, such as matrix algebra, fitting, minimisation. Some will be provided by external class libraries, others will be internal to the ATLAS reconstruction suite. It is recommended that classes at this level are not made into AlgTools, in order to keep them generic.
- The architecture team should consider how to reconcile the issues of "on-demand" services and tools versus deterministic requirements of the HLT.
- Steering design(s) that encompass seeding, reject/continue decisions and classification for both offline and HLT use cases should be completed.

6.2 Specification of input/output data keys

The specification of the reconstruction involves specifying a network of algorithms instances, each instance retrieving (possible multiple) input data store object specified by their type and a key, writing out often

one (but not necessarily) output data store object. Figure 27 shows a schematic example which will allow to illustrate the discussion. To build the network it is necessary to specify for each algorithm the keys of their inputs and outputs [11], which keys have to match at the end. Past experience shows that this turns out to be often difficult, because it requires coordination between different packages; it is also difficult for end user analysis which need to specify the types and keys of many objects. There is clearly a potential scalability problem as the reconstruction network will grow in complexity.

6.2.1 Current situation

Before discussing this key synchronisation problem, let's review the situation.

- StoreGate allows:
  - single object writing with a key;
  - single object writing with no key (an internal key is created so that this can be done many times);
  - single object reading with a key;
  - single object reading without a key (if there are several objects of the same type the last one written is returned);
  - multiple object reading without a key through DataHandle iterator.
- Single keyless retrieve or write are hardly ever used, to allow for multiple algorithm instances (see later).
- Input and output keys are specified in most cases (not always) through jobOption properties in the jobOption fragment of the package the algorithm belongs to, to allow multiple algorithm instances to be run and read/write different objects.
- There are no conventions for the property name, nor for the key name.

The first recommendations at the end of this section address some of these issues.

6.2.2 Key synchronisation use cases

To discuss the problem of synchronisation of input and output keys, a number of use cases have to be considered. Let's suppose the network of reconstruction algorithm has been specified as exemplified on Figure 27.

Use Case 1: a new algorithm *I* (performing a new task) needs to be run downstream of the existing network, and needs various inputs, for example *d(Beta)* and *h(Alpha)*.

- **1a** There is no ambiguity on the object input type (for example *h*). It is desirable not to have to specify the key.
- **1b** There is ambiguity (for example *d*), then it is obviously mandatory to specify a key (here *2*)

Use Case 2: a new algorithm *K* (or a differently configured new instance of *C*) needs to be run instead of an existing algorithm *C* with the same inputs/outputs, so that downstream algorithms can be run unchanged with the same inputs/outputs. This is required both for the immediate downstream algorithms *D*, *D2* and *E*, and algorithms further downstream *F*, *G* and *H*.

- **2a** The data has not been processed yet.

[11] The types are by definition hardwired in the code.

- **2b** This is a partial reprocessing of existing data, output collection *c(Alpha)* already exists, as well as the output of further downstream algorithms. The newly made objects should be used as inputs of all the further downstream algorithms instead of the old one.

Use Case 3 : same as Use Case 2 but the new algorithm needs to be run in parallel (rather than instead) to the existing one in order to compare the performances in the same job.

- **3a** If there is no need to connect the new algorithm to the downstream tree, the output key is different from the existing one. For example, an algorithm K (in parallel to C) which will output object *c(Beta)*.
- **3b** If performances of down stream algorithms also need to be studied, new instances of the downstream algorithm (here *D1*, *D2*, E, F, G and H) with new inputs/outputs keys need be created. This would be tedious, and the usefulness of it will have to be weighted with the possibility to run a the job a second time after replacement of the algorithm by another one (use case 2a) and the comparison will be done at ntuple level. Still, this should be possible and not too difficult, as for example if it was just algorithm E which would be replaced and a new instance *H2* created taking *d(Delta)* as input.
- **3c** This new algorithm K now becomes a standard part of the reconstruction network to feed algorithm E (E now takes *c(Beta)* as input instead of *c(Alpha)*). (A real-life example could be that C is a tracking algorithm, and K a new one dedicated to find tracks for conversion finding algorithm E).

6.2.3 Location of key specification

If all the inputs and outputs have keys specified in the jobOptions (which correspond to the existing situation) all the use cases listed above can be satisfied but there are maintenance problems:

- As is mostly the case now, the keys are specified in the jobOption fragments maintained in each package: then this is the synchronisation problem, and the person writing a new algorithm should know about all the keys used.
- All the keys are specified in a common place, then this violates encapsulation and this common place is difficult to maintain.
- The proposed solution for the medium term is to have a few places where these keys are maintained, for example one per subsystem reconstruction algorithm and one per combined reconstruction algorithm.

In use cases 1a, 1b, 2a the keys of all inputs need be specified. Use cases 2b, 3b and 3c are difficult because all the algorithms input and output keys downstream of C need to be changed in the jobOptions.

6.2.4 Generalising keyless reading

It is proposed that keyless read is used whenever possible, i.e. when objects are unambiguous. So for the example in Figure 27, these are objects a, b and c.

- Use cases 1a, 1b and 2a are easier because only keys of ambiguous input objects need be specified.
- Use case 2b is difficult: all input and output keys of downstream algorithms need to be modified so that the output objects do not collide with existing one.
- Use case 3b is even more difficult than use case 2b because new instances of algorithms need be created instead of modifying existing ones.

- Use case 3a : suddenly *c* becomes ambiguous, so either *K* is run after *C* (which may not be possible in complex cases), or the input keys for *D*,*D2* and *E* need now to be specified, because there are now two instances for *c* (all this can be done in private jobOption)
- Use case 3c : if *K* is integrated in the standard reconstruction, the modification of the input keys for *D*, *D2* and *E* need to be propagated wherever the official keys are specified.

A better way to solve use case 2b (partial reprocessing, which is fairly common) and 3b is still needed.

6.2.5 Key versioning

The easiest way is to solve use case 2b is to arrange that attempts to overwrite an object of the same type and key as one that already exists, simply deletes the existing one (in transient memory and with sufficient safe-guard so that this is not done inadvertently). Then, for use case 2b, no modification of output keys nor input keys of downstream algorithms is needed. The other use cases are not simplified. However this would cause dangling pointers from other objects with no guarantee that these objects will be overwritten in turn (if not all algorithms are rerun).

The possibility to define a "default" key has been explored but was rejected because several problems remained with it: when different instances of objects are made for a specific purpose (for example: tracks for b tagging, electron ID, V0 finding), or corresponding to different part of the detector (cell collections from LArg em, FCAL, Tile, HEC ...), a default could not be defined. Also there should be a mechanism to define the "default": is it the algorithm writing the default collection? a separate algorithm? what if there is an attempt to define two defaults ?

Finally, it was felt that the best way to solve use case 2b is to introduce the concept of version of keys. The version would be an integer number automatically generated, so that writing an object of an already existing type and key will create a key of a higher version (with sufficient safe-guard so that it is not done inadvertently). All retrieve statements will retrieve only the object of the highest version, so the downstream algorithms would transparently (no change of input nor output keys) use the most up to date objects. Only for debugging purposes would it be possible to retrieve/write specifically lower version numbers, but this will be disallowed in production. For example, use case 3a and 3b would also be solved by having *K* write on purpose a lower version of *c(Alpha)* and the new downstream algorithm instances access it, so that the new branch does not interfere with the existing one. However, the keys will need to be properly redefined before moving *K* into production as in use case 3c.

Key versioning requires a non trivial change to StoreGate and the implications for persistency need to be thought fully so it is suggested that the overwriting mechanism be provided as a first stage.

Finally it would certainly be very useful to have an automatic tool which would be able to dump the list of all the input and output collections and keys of all algorithms in a given job. One could even imagine using a graphical tool à la cmtgrapher to display the result.

6.2.6 Recommendations for specification of keys for input and output data

- For any algorithm, the keys for all input objects and all output objects should correspond to properties of the algorithm.
- The properties name and variable name should follow some convention (see next section).
- The keys themselves should follow some convention (see next section).
- A migration strategy to the new conventions needs be defined. In any case, all new algorithms should abide to the convention.
- The definition of output keys and input keys (if ambiguous) of algorithms need be grouped in a few logical places.

- There should be a debugging tool within Athena listing all the input and output objects of all algorithms with their type and keys. The output of which should be viewable graphically, à la cmtgrapher.
- The methods for single keyless reading and writing should be removed from StoreGate and replaced by single reading and writing with an empty key, which will be equivalent except for the changes below [12]. (In the following read/write with an empty key will still be called keyless reading and writing.)
- It has already been realised that single keyless retrieve returning the last object written in Data Store was ill-defined, in particular when reading from persistence. So it is proposed that single keyless retrieve simply fails if there are multiple objects of the same type, because they are ambiguous (except in the case of multiple versions, see below).
- Keyless writing is disallowed (except the very special case of using automatically generated keys from StoreGate).
- Keyless reading is encouraged whenever there is no ambiguity, e.g. there is no need to specify the EventInfo object has key "EventInfo".
- A new concept is introduced, the concept of versioning, which would allow objects of the same type and key, but different versions to coexist in the event store.
  - This would simplify the specific use case of partial reprocessing, where a part of the network of algorithms is redone.
  - Writing an object of existing type and key will cause the object be recorded with incremented version number and cause a warning to be issued.
  - Normal retrieve will always access the latest version (even DataHandle iterators on all collections), irrespective of their lock state.
  - Special writing with a specified version number will be possible for debugging purposes (for example by specifying the versions as in *Alpha;1*).
  - Special retrieve methods to access older version will be provided for debugging purposes.
  - Writing/reading with a specified version number should not be part of the normal processing (otherwise one would soon need versions of versions).
  - A debugging tool dumping a list of all existing versions of a given object should be available.
  - Connection with persistency needs be thought out fully.
  - Before this is available in StoreGate an intermediate step allowing overwriting of existing object should be provided.

6.2.7 Convention for key specifications

The properties name, variable name and the keys themselves should follow well defined convention. It is all the more important given that properties names in jobOptions are seen and used by many more users than the code itself, so the fact that we have a coding convention for the code but not for jobOption properties nor keys is a clear gap.

The convention below should be adopted by all new algorithms. A migration strategy for existing algorithms needs be defined.

The property name and the keys should adhere to Atlas coding convention: leading upper case, use of upper case at the start of each word when they are run together, no underscore.

The property name should be derived as follows.

[12] this makes it easy to use keyless reading and writing while having the possibility to switch to a key through the jobOption

- For input objects: < **generic object type**> **Name** or if a list of input object is to be given (so the property is a vector of string instead of just a string): < **generic object type**> **Names**.

  The < **generic object type**> is the object type simplified with the following convention:

  - the exact type does not need to be used to ease future migration of the Event Data Model, so instead of SimpleTrack or IpatTrack or XKtrack, just use Track;
  - for the same reason and for conciseness, Container or Collection are noted by adding an s to the data object type, so instead of TrackContainerName or TrackCollectionName, just write TracksName.

- For output objects:< **generic object type**> **OutputName**

- The property variable itself inside the code would simply be: **m_** <**property name**> except that the property name has now a leading lowercase letter, because that's ATLAS coding rule.

- The key itself should specify in what sense this object would differ from other object of the same type, for example:

  - it could be the algorithm which produced it (e.g. currently "xKalman" or "iPatRec" for Tracks);
  - it could be the part of the detector it refers to (e.g. "LArEM" or "LArFCal" for CaloCellContainers);
  - it should not be redundant with the property name or object type: no need to have "TrackCollection" as a key for a TrackCollection;
  - it should not contain anything like "key" or "loc" or "name" and only be made of alphabetical or numerical characters (no special characters like "/" "+" except possibly in some specific cases where keys are to be decoded inside the code).

A few examples taken from real code in in release 6.3.0:

- `CombinedJetAlg.cxx`

  ```
  declareProperty("outputCombinedJetContainer",m_outputCombinedJetLoc);
  ```

  would become

  ```
  declareProperty("CombinedJetsOutputName", m_combinedJetsOutputName)
  ```

- `MissingETCellBuilder.cxx`

  ```
  declareProperty("MissingETObjName",m_MissingETObjName);
  ```

  would become

  ```
  declareProperty("MissingETOutputName",m_missingETOutputName);
  ```

- `egammaBuilder.cxx`

  ```
  declareProperty("egammaContainer",m_egammaContainerName );
  ```

  would become

  ```
  declareProperty("EgammasOutputName",m_egammasOutputName );
  ```

References

[1] S. Albrand, *Tests to investigate the overhead of a virtual function call* June 2000
`http://a.home.cern.ch/a/albrand/www/TestCode.pdf`

[2] Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison Wesley 1995, ISBN 0201633612

[3] J. Boudreau, D. Quarrie, R.D. Schaffer, *Detector Description - document in preparation.*

[4] *Athena Developer Guide 2.0.0 Draft*
`http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/architecture/General/index.html`

[5] P. Calafiura et al., *StoreGate: A data model for the ATLAS software architecture*, presented at CHEP 2001
`http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/architecture/EventDataModel/Tech.Doc/StoreGate/CHEP01.pdf`

[6] P. Calafiura, H. Ma, S. Rajagopalan, D. Rousseau, *The ATLAS Data Model User's Guide*
`http://atlas-sw.cern.ch/cgi-bin/cvsweb.cgi/offline/AtlasDoc/doc/DataModel/DataModel.pdf`

[7] The ATLAS EDM Group, *The Raw Data Flow in ATLAS*
`http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/architecture/EventDataModel/RawDataFlow.pdf`

[8] S. Armstrong (editor), M. Elsing, D. Froidevaux, I. Gavrilenko, R. Hawkings, N. Konstantinidis, A. Poppleton, W. Wiedenmann, *Requirements for an Inner Detector Event Data Model*, ATL-INDET-2002-014

[9] CLHEP, a Class Library for High Energy Physics `http://wwwinfo.cern.ch/asd/lhc++/clhep/`

[10] Peter Loch et al., *Calorimeter reconstruction navigation*, document in preparation.

[11] M. Elsing et al, *Analysis and Conceptual Design of the HLT Selection Software*, ATL-DAQ-2002-013, Jun 2002

[12] Unified modelling language `http://www.omg.org/uml`

[13] Booch, Rumbaugh, Jacobson, *The unified modelling language user guide*, Addison Wesley 1999, ISBN 0201571684.

[14] C. Padilla, Analysis Tools in a Unified Framework, presntation at the RTF open meeting, 18-Feb-2003
`http://agenda.cern.ch/askArchive.php?base=agenda&categ=a03403&id=a03403s1t7/transparencies`

Appendix

A RTF mandate

> The mandate of the RTF is to perform a design iteration on the reconstruction, revisiting the granularity of the algorithms and the event data model in the light of experience gained with the existing design and feedback from additional requirements such as those from the HLT. The redesign is to take a "top down" approach that should take advantage of the existing design, but not be constrained by it. The deliverable will be a design document containing conceptual class diagrams for the event data model and dataflow diagrams to show the main algorithm building blocks and how they fit together with the EDM.

B RTF task list

This is the task list that forms the work plan of the RTF. The aim is to cover all high priority items, as many medium items as possible, and if time permits some of the low priority items to. This includes working through use cases to illustrate these points and provide worked examples of the design.

B.1 High Priority

- Event Data Model. Data flow diagram showing high level EDM and algorithms/tools, classify EDM into domains (e.g. raw, subsystem reco, combined reco, analysis). Show how feedback loops will work.
- Common core API to data classes produced from fast simulation, full reconstruction, HLT.
- Data class design. Reference frames (different origins for track reco vs. analysis), coordinate systems, common units and provision for transformations. Optimisation for reconstruction. Separation of event and non-event data. Produce recommendations - "style guide".
- Navigation. Composition, use relationships, reverse navigation, exclusivity, HLT steering, association to MC truth (see below). Relationships must be persistable.
- Algorithms. Steering: design to accommodate both full scan and regional reconstruction. Granularity and flow: tools vs. sub-algorithms. Produce recommendations - "style guide".

B.2 Medium Priority

- MC truth - not the particle classes but relationships to them from reconstruction data. Some issues are: AOD to truth relationships that don't go via ESD; different needs of reconstruction developers (who need detailed truth) and analysis developers (who just want a simple association for their AOD objects). As a starting point, take current work in inner detector and calorimeter.
- Compact data definition to allow use of same objects at AOD/ESD.
- Particle ID framework - how to handle cuts, vetoes, PDFs. Take current work as a starting point.
- How to compare two different algorithms to see if they are doing the same thing, using ntuples.
- Calibration - work through use case of how to produce a new calibration or set of alignments.

B.3 Low Priority

- Package structure, aka physical design. Note that this is a wider problem than just the reconstruction. Also note that there are already sound recommendations but they are not universally applied. Try to actually do the physical design for the reconstruction or a representative part of it.
- History - a mechanism to record which algorithm with which properties created a data object, for future reference. (Can assume everything is persistable.)
- JobOptions - dissatisfaction reported with the current system, seem to be due to their maintenance and/or complexity. Think about how this could be improved.