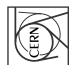
ATLAS Note

EventView

- The Design Behind an Analysis Framework

ATL-SOFT-PUB-2007-008
18 December 2007



K. Cranmer
New York University, 4 Washington Place, New York, NY 10003
(Previously Brookhaven National Lab, Upton, NY, USA)

A. Farbin
University of Texas at Arlington, 502 Yates St, Arlington, TX 76013

A. Shibata
New York University, 4 Washington Place, New York, NY 10003
(Previously Department of Physics, Queen Mary, University of London Mile End Road, London, UK)

Abstract

The development of software used to process petabytes of data per year is an elaborate project. The complexity of the detector means components of very diverse nature are required to process the data and one needs well defined frameworks that are both flexible and maintainable. Modern programming architecture based on object-oriented component design supports desirable features of such frameworks. The principle has been applied in almost all sub-systems of ATLASsoftware and its robustness has benefited the collaboration. An implementation of such framework for physics analysis, however, did not exist before the work presented in this paper. As it turns out the realisation of object-oriented analysis framework is closely related to the design of the event data object.

In this paper, we well review the design behind the analysis framework developed around a data class called "EventView". It is a highly integrated part of the ATLASsoftware framework and is now becoming a standard platform for physics analysis in the collaboration.

The idea is the whole thing. If you stay true to the idea, it tells you everything you need to know, really. – David Lynch

1 Introduction

The ATLAS detector is a complex collection of cutting-edge particle detection technologies, which consists of several sub-detectors of very different nature. Its components ranges from state-of-the-art silicon tracking devices to a complex of muon spectrometer system. Construction and integration of such intricate devices cast a major challenge. Accordingly, development of the computing software required for the experiment faces countless issues including full detector simulation, event reconstruction of the detector output, generation of Monte Carlo events and physics analysis.

To incorporate wide variety of demands and to provide uniform interconnection among the offline software, the ATHENA framework (figure 1) was developed to assemble diverse sub-components and external packages. Software projects within the framework share common interfaces and services, which enables them to communicate between each other. At the same time, the framework is general enough that context-specific sub-systems can be built within ATHENA according to more specific requirements.

Physics analysis is at the end of the computing workflow and it depends on a large portion of the rest of the framework. Therefore, a general analysis package such as ROOT [1] by itself is not sufficient for ATLAS physics analysis[1].In-framework analysis is the only place one can obtain full accessibility to ATHENA reconstruction algorithms and, hence, general and powerful analysis tools must be developed within the ATHENA framework.

Prior to this work, the development of in-framework analysis was based on a traditional approach whereby loosely related sub-routines are instantiated from one or a few tightly related algorithms invoked via ATHENA. Various problems were encountered through this approach. In this papera novel approach to physics analysis based on the concept of an EventView, and an object oriented component model is presented. At the core of the idea is the representative "view" of an event, which defines the contents of event data suitable for event-level physics analysis. This enabled us to develop fully fledged analysis framework, the "EVENTVIEW analysis framework" (or simply, EVENTVIEW[2]), which is highly flexible and modular in nature.

The existing ATLASsoftware infrastructure and the event data model forms the backbone of the EVENTVIEW framework, and are detailed in section 1. Section 2 introduces the core philosophy behind the design of the framework where we apply the ideas of object-oriented component model to designing analysis tools. Eventually this flourished into a fully-fledged analysis toolkit and the same design principle was applied to different purposes that commonly arise in an analysis, as shown in section 3. Finally in section 4, we will show the role of this framework within the ATLAS collaboration and how the framework is used in real physics analyses.

[1] ROOT is an external component of ATHENA but ATHENA is not built on top of ROOT like it is the case in some experiments. Therefore in-framework analysis may use ROOT functionality but is fundamentally different from stand-alone ROOT analysis.

[2] "EventView" is used to refer to the data object while the notation "EVENTVIEW" is used to refer to the whole analysis framework including the data class and the tools built around it

1.1 Relevant Components of Athena

In terms of software development, one of the main aims of EventView is to factorise the complex process of physics analysis into well defined modules that represent a single task or a grouped operation. Such software design allows single parts of the entire physics analysis to be modified or exchanged without disrupting the rest of the analysis. The whole of this structure is embedded in the Athena software framework, which generously supports flexible sub-systems. Many of the components of the EventView are derived from the architecture of Athena. It is therefore appropriate to introduce the relevant components of Athena in this section.

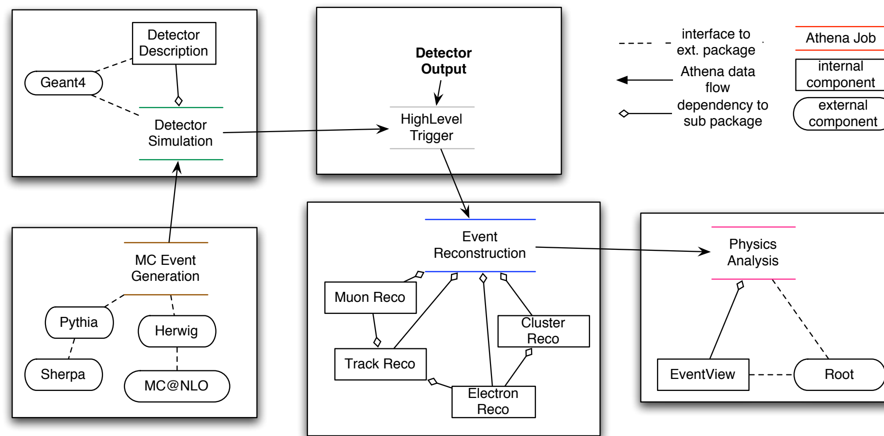


Figure 1: Various computing tasks in Athena components model.

Athena [2] [3] is an enhanced version of the original C++ based software framework Gaudi [5], initially developed by the LHCb collaboration. Component model, employed by the Athena-Gaudi architecture is a common software design of large scale projects where numerous types of internal and external software components needs to be encompassed in a single application. The component library structure allows that modules are loaded as shared libraries at job configuration level, or at run-time. As a result, dependencies between various libraries used in the application is reduced for increased stability of the framework.

Three main basic building blocks can be named, which forms the pillar of the Athena architecture:

- The `Service` class is designed to provide dedicated functionality throughout the execution of the program. One of the important realisation of a `Service` is the transient data store, `StoreGateSvc` (or simply "StoreGate"). The instance of `Service` classes are handled by a central `ExtSvc` manager that regulates initialisation and finalisation and the facility is uniformly provided to all Athena components.
- The `Algorithm` class represent the primary algorithmic part of an Athena application. It is dedicated to actions that are taken exactly one time at every event and classes derived from the `Algorithm` class needs to be registered to the central `ApplicationMgr` that steers initialisation, finalisation and the execution of the `Algorithm` at every event.
- The `AlgTool` class provides more flexible solution for smaller pieces of algorithms that are typically invoked multiple times within different contexts. `AlgTool` instances are called through an `Algorithm` that either owns `AlgTool` (in which case called *private*) instances

or retrieves them through the central ToolSvc where all public tools are registered. This pattern allows AlgTool classes to be instantiated multiple times with different configurations or once with same configuration but used multiple times from different Algorithm objects.

Algorithm and AlgTool are usually written in C++ since it is advantageous in terms of computing efficiency for it produces compiled binary libraries. On the other hand, robust configuration capabilities are provided in Athena by Python scripting language [6]. So-called "Python bindings" enable configuration of C++ Algorithm and AlgTool from the Python interpreter. Being an interpreted language, Python is equipped with a dynamic scripting environment, which favours rapid development and interactivity. In addition, it is a multi-paradigm language with support for high-level dynamic data types and a design concept such as object-orientation.

Generally, the Algorithm is responsible for retrieving input data collections from and writing the output data to the transient event store, StoreGateSvc. On the other hand, modularisation of analysis can be achieved by taking advantage of light-weight AlgTool classes, which are self contained collection of small algorithms that can be dynamically chained together through Algorithm using run-time configuration. EventView fully benefits from lightweight C++ AlgTools and their configurability provided by Python to realise a flexible modular framework. The implication of such a system is significant and highly related to various aspects of physics analysis.

1.2 Developments of Analysis Event Data Model

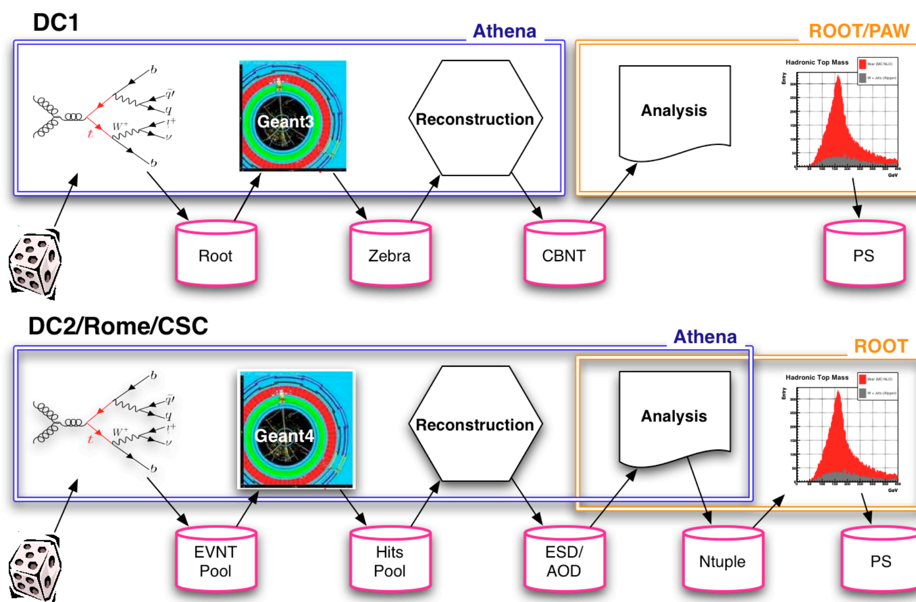


Figure 2: Workflow before and after introduction of structured data. The first workflow refers to the model available in DC1 while the latter is used in the subsequent productions including DC2, Rome and CSC.

The design of Athena framework places a strong emphasis on separation of data classes and algorithmic code. This is a consequence of the data centred architecture employed by Athena, which is referred to as *blackboard architecture style* in [7] and summarised in [8] as follows:

> the `StoreGateSvc` acts as the blackboard to which the clients read from or write to (in Athena, this is represented by the templated `StoreGateSvc::retrieve()` and `StoreGateSvc::record()` interface respectively). The `ApplicationMgr` plays the role of a *controller* (teacher) in this model and organises the reading and writing to and from the blackboard. The result of the blackboard design can be seen as a *pseudo data flow*: in an abstract picture the data objects are handed over from one `Algorithm` to the next one in the sequence, while in reality the data exchange always progresses via the blackboard.

Under blackboard design, the design of data objects has an intrinsic importance to the subsystems, which deals with the data object. It effectively becomes the *language* in which the algorithms are written in for each part of algorithm is defined in terms of its interaction with the data object. Therefore, the Event Data Model (EDM) is an inherent part of Atlas computing model, which defines the analysis model of the experiment.

Tightly coupled to EDM is the ability to write data into files ("persistification"). Before persistification of structured data objects (such as objects of class `Electron`) was introduced, the results of reconstruction were written into "flat" Root ntuples, which only contained integers and float numbers (and arrays of them). Therefore, new persistification technology called Pool[9] was developed to support flexible I/O handling of data in the LHC experiments. This replaced all previously used data format in Athena workflow as shown in figure 2.

At the time of computing exercise called Data Challenge One (DC1, 2002-2003), there was no EDM within Athena for event reconstruction or physics analysis. The natural consequence was that analysis of reconstructed data was performed solely out of the framework using Root. With the arrival of Pool, structured data classes were developed by the time of Data Challenge Two (DC2, 2004-2005). Results of reconstruction can now be written out in a high-level data structures that can be read in Athena again. This opened the possibilities for in-framework physics analysis that is well interfaced with the rest of the framework. It is still possible to do most of physics analysis outside the framework though in-framework analysis has numerous advantages.

By the time of Rome computing production (2005), the output formats of the event data were firmly established. The first output of event reconstruction is saved in Event Summary Data (ESD) [10]. Reconstruction EDM objects are persistified in ESD but due to the large event size (500KB per event), they will only be available at Tier-1 Grid sites[11]. ESD is therefore slimmed down into analysis EDM objects and persistified as Analysis Object Data (AOD) that are small enough to be made available in all Tier-2 sites (a target of 100KB per event). The contents of AOD should provide sufficient information for most physics analysis except detailed study of the detector. While reduction of information is required due to size requirements, separate implementation of reconstruction and analysis EDM is not necessary, or desirable, if EDM classes had ability to dynamically regulate their data contents. This, "ESD/AOD Merger"[12] is being realised in the latest development of EDM classes.

An emerging feature of EDM is Derived Physics Data (DPD), which is actively exercised in Computing Service Commissioning (CSC, 2006-2007). To support a range of analyses, the contents of AOD must be fairly general. Objects in AOD are thus only loosely defined candidates of final analysis objects and overlapping interpretation of the same objects coexist. For instance, a reconstructed electron candidate is almost always reconstructed as a jet candidate; in addition, there are several jet candidates reconstructed using different reconstruction algorithms. In a given analysis, one needs to resolve such ambiguities between their analysis objects via process of preselection and overlap removal and construct a consistent view of an event. It may also

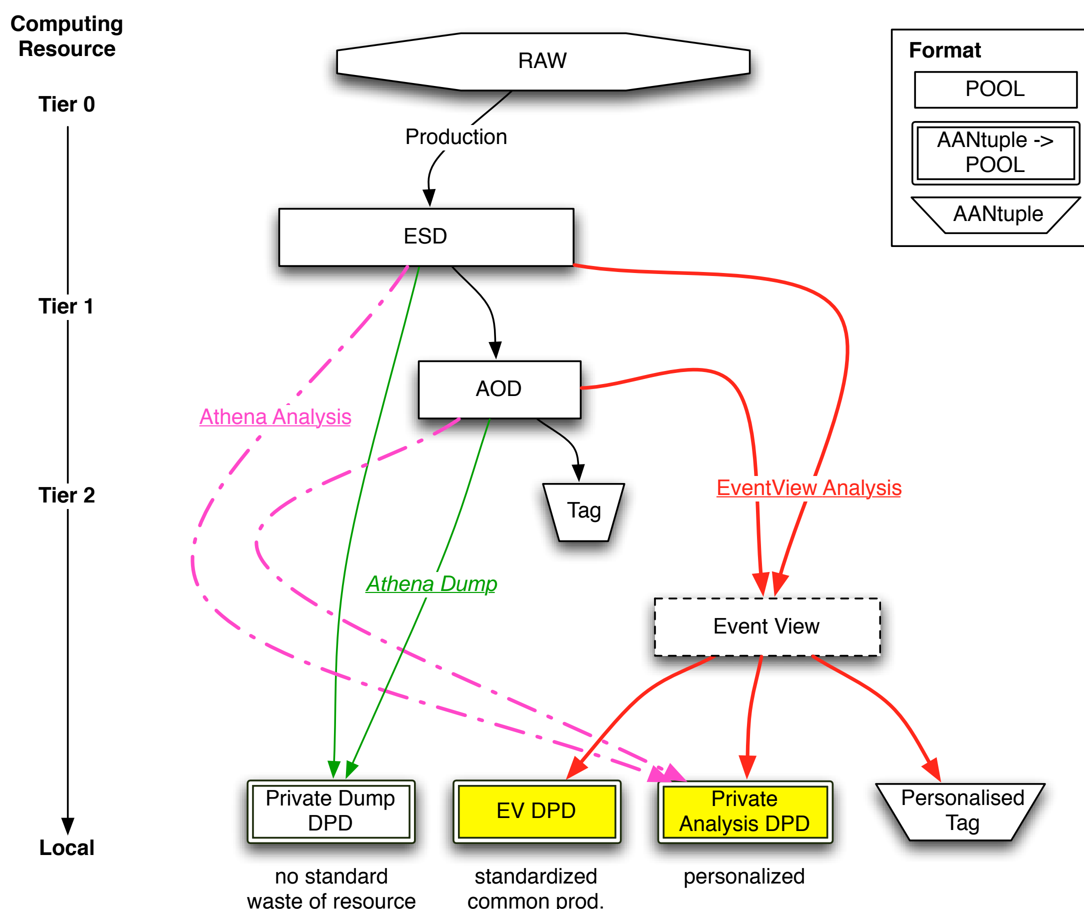


Figure 3: Different types of DPD in relation to the rest of EDM. Athena algorithms are used to produce DPD from ESD and AOD, which may result in a specialised contents (thick solid, pink line). EventView can be used to standardise this process while leaving the possibility for customisation (dashed, red line). This solution is significantly improved compared to copying the whole contents of AOD, which was seen in earlier analyses. TAG is used to quickly search through the datasets to select a relevant set of events for DPD production.

be necessary in this process to apply refinements to these objects by re-calibrating objects and re-running particle identification algorithms [3].

As shown in figure 3, in the early days of AOD analysis, DPD were produced without much organisation and significant amount of redundancy was observed in producing such data. In many cases, a simple algorithm was used to copy the contents of AOD into Root ntuples without any further processing. While highly personalised data production using private Athena algorithm may be beneficial in cases where there is a very specialised purpose, much of the DPD production can be standardised to improve communication between related analyses.

Therefore, DPD production is an activity within the computing model with a fast production cycle (of the order of weeks) adopted to incidental needs of physics analysis. EventView provides a framework to construct such derived data and standard set of tools were developed for building and persistifying such data. It can be used to produce customised DPD shared

[3]An alternative approach, currently under development, is to construct a particle level view, “ParticleView”. Each ParticleView object represents one abstract physical object, which turns into corresponding representation depending on the context. This provides an elegant interface to manipulating multiple representations, existing or new, and replaces the `EventViewTransformation` tools mentioned later.

within a group or more specific contents for a single analysis.

It is important that DPD supports generic ROOT I/O as they bridge in-framework (ATHENA) analysis and out-of-framework (ROOT) analysis. It is also crucial to be able to trace back to the AOD/ESD it was produced from so that one can inspect interesting events in more detail. In release 12 of the ATHENA, the AOD and ESD is written via POOL to a ROOT file, but it is not possible to access the EDM objects in ROOT. Instead, the primary DPD format is a ("flat") ROOT ntuple (`TTree`) with additional ATHENA information to make it "Athena-Aware" (hence the name "Athena-Aware Ntuple"). In release 13, the "flat" DPD format will mostly be replaced by a POOL-based DPD once the ongoing development of "AthenaROOTAccess" technology is completed[13]. In short, this technology uses the persistent-to-transient converters for the ATHENA EDM classes within ROOT and provides mechanisms that function like `ElementLink` and `DataLink` within the context of ROOT.

2 Core Components of the EventView Analysis Framework

2.1 The EventView EDM Class

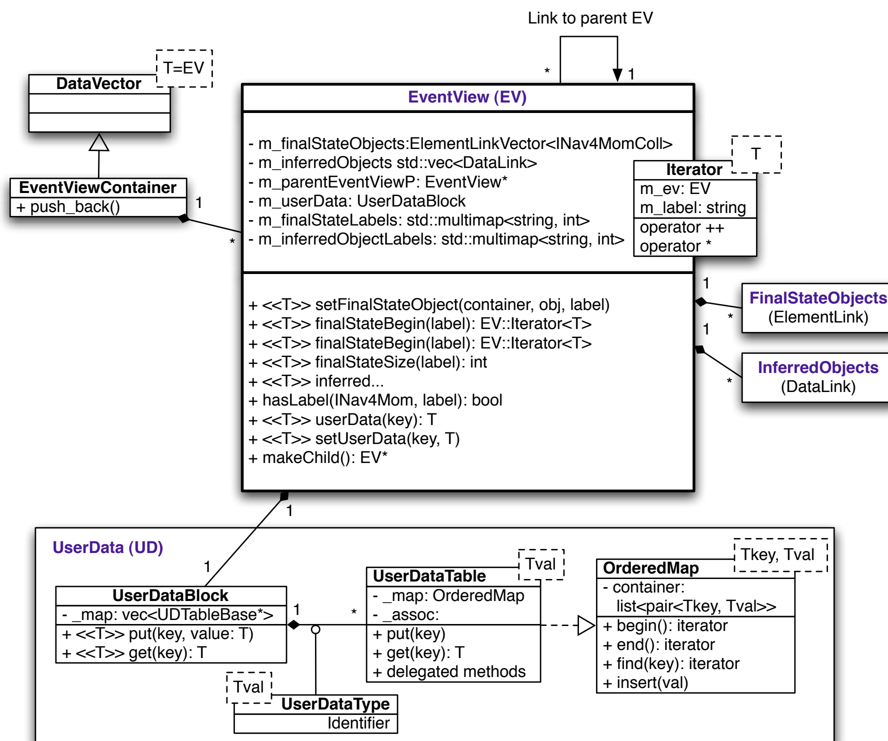


Figure 4: UML diagram of the EventView class and closely related components.

There are two major types of data being managed during analysis: particle-like objects and user-defined variables. Particle-like objects are either read from a file or created during analysis; in both cases one wants a very convenient and flexible way to access and group these objects. User-defined variables are usually simple integers and floats; the challenge here is one of bookkeeping. The EventView EDM class is a specialised EDM class for analysis that has been designed to ease these types of data management, while retaining StoreGate as the fundamental technology for memory management and I/O handling.

In short, the EventView EDM class acts as a proxy to StoreGate for all particle-like objects (including tracks and clusters as well as electrons and jets.) It does not own any of the particles, it only points to them and attaches labels to them. Therefore, memory management, POOL conversion and other I/O related operations are still dealt with by StoreGate. Instead of raw C++ pointers, the links are based on basic ATHENA framework's persistifiable pointers: `ElementLink`, `ElementLinkVector`, and `DataLink`. These links need to specify a type, and so the EventView choose the `INavigable4Momentum` interface as a common base type since for all the particle-like objects[14].

For user-defined variables the EventView provides an elegant solution. Since there may be many views of the event in which the same logical quantity (e.g. the sum of p_T of the jets in the event) may take on different values, it is natural to group these variables with the EventView. The EventView stores these variables in its UserData (see below) with user-defined keys. Because the EventView is in turn recorded in StoreGate the memory management of the UserData is ultimately handled by StoreGate. This approach together with the templated UserData allows users to store arbitrary data with a natural bookkeeping device that is coupled to the particle-like data in the event without having to worry about memory management or create a new data structure and registering a unique Class ID (a requirement for storing something in StoreGate). This design provides the flexibility needed for analysis without recreating the memory management functionality of StoreGate.

In addition to the technical issues of managing data, the EventView interface eases several common operations, such as giving particles user-defined labels for bookkeeping purposes, `dynamic_cast`-ing from the `INavigable4Momentum` interface to a concrete class like `Electron`, iterating over particles that can be cast to a certain type and which satisfy certain labelling requirements. In addition, the EventView design removes the need to create several new "View Containers" and register them in StoreGate – a common practice in non-EventView based analysis code that is error prone, often leads to segmentation faults due to ownership conflicts, and which causes many problems when writing POOL-based DPD.

The conceptual definition of EventView was formulated through discussions in [15] [16] and its crucial idea is summarised as follows:

> An EventView is a collection of physics objects, which are coherent, exhaustive and mutually exclusive. EventViews are not unique; for each event a user may wish to consider the event with multiple different views. From this view, a user may wish to calculate several quantities (thrust, likelihood the event came from a given hypothesis, etc.) and associate it with the view (thus the collection of physics objects may include non-four-momentum like entities).

The realisation of the EventView class consists of three types of sub-containers:

- Final State (FS) Objects : Preselected objects considered in an analysis.
- Inferred Objects (IO): Secondary objects reconstructed out of the final state objects.
- UserData (UD): Variables calculated during the course of an analysis.

Figure 4 illustrates the design of the EventView EDM class and its sub-components.

2.1.1 Final State and Inferred Objects

The first step in a physics analysis is to identify the relevant objects for the analysis, a process called "Analysis Preparation" in [14]. Those objects are preselected out of the loosely defined candidates in AOD. Objects with multiple representations need to be resolved at this stage by removing the overlaps according to the precedence defined for the analysis. Links to these objects (called `ElementLink` ATHENA) are stored as Final State (FS) object in EventView. In the course of analysis, secondary objects are reconstructed, e.g. Z boson from two electrons. To retain coherence of the FS objects and avoid double-counting, links to these objects (`DataLink` in ATHENA) are stored separately in Inferred Objects (IO) [4].

Objects in FS and IO are accessed through templated iterators. Down-casting to the concrete class is factorised in the dereferencing of the iterators. The `begin` and `end` methods returns the iterators for the subset of FS and IO specified by the type and label information; incrementation of an iterator will invoke type identification of the object and label requirement is checked using the `haslabel` method. For example, one can obtain an iterator for objects of type `ParticleJet` with label "Tagged" via: `ev->finalStateBegin<ParticleJet>("Tagged")`.

2.1.2 UserData

Various quantities are calculated in an analysis. For example, sum of p_T of the jets, can be useful for discriminating background. One may also wish to re-calculate the missing transverse energy for the choice FS objects made in the analysis. UserData (UD) is a data store for any such quantities occurring within an analysis. Its concept is similar to `map` in the C++ standard template library but extended to allow dynamic building of multi-type data structure.

One instance of `UserDataBlock` is held by each EventView object. When a variable of a certain type is put into UD, it creates new `UserDataTable` if it is the first instance for a value of that type to be inserted. Otherwise the existing `UserDataTable` is used for this variable. `UserDataTable` is templated for the requested type and it delegates an instance of `OrderedMap` object. `OrderedMap` is like `std::map` but the ordering of contents follows the order of insertion. It only allows sequential access of the contents (i.e. no random access) for performance optimisation and on its own supports very simple operations.

In short, multiple instances of `OrderedMap` objects for each requested template types, delegated through `UserDataTable` with additional methods are managed by `UserDataBlock`. These methods are forwarded to the front end user interface, which exists in the EventView class as appropriate. The user interaction with UD is rather simple and he/she only needs to specify the key and the value to be inserted (e.g. `setUserDataa("key", val)`) from which point run-time type information is used to resolve the whole operation.

2.1.3 Multiple EventViews and `EventViewContainer`

As introduced earlier, support for multiple EventViews is a frequently occurring requirement. Multiple object preselection may be compared by keeping them separately in different EventView instances created in the same event. Another prominent example is when one has to consider multiple combinatorial choices when secondary objects are reconstructed (e.g. make all possible combination of dijets in the event to find W like combination.) Not only that FS and IO and their labels may differ in each view, the calculated quantities in UD would also

[4] If the Z boson was placed in FS, looping over all objects in FS would double-count the electrons.

differ from one view to another. Bookkeeping such a complex situation is rather trivial with multiple EventView as each instance holds independent and completely separated containers for each view. These instances are held together by `EventViewContainer`, or EVContainer, which inherits from Athena `DataVector`[5].

2.2 The EventView Application Manager and Component Interface

Applying the analogy of *blackboard architecture style* to the EventView framework, the EventView EDM class can be seen as the blackboard and the `EventViewToolLooper`, or "EVToolLooper" is the teacher who controls the flow of the application (i.e. an application manager). The main blackboard of Athena still remains as the StoreGate and `ApplicationMgr` is still the primary controller of the whole Athena job, but EV now work as a lightweight secondary blackboard and `EventViewToolLooper`, an Athena `Algorithm`, acts as the controller of the private `AlgTool` instances.

In terms of architecture design, a crucial difference between StoreGate and EV is that EV is more like a notepad passed around between the pupils rather than a heavy blackboard stuck on the wall, which pupils need to come forward to access. In fact EVToolLooper passes around the EventView object (or a `EventViewContainer` object) to each `AlgTool` components whenever they are executed. Since EventView holds all the information necessary for the analysis at hand, it is the only data object algorithms need to interact with in most situation (it is always possible to access StoreGate if needed.) For this additional interface, an interface class called `EventViewBaseTool` is derived from `AlgTool`. All EventView sub-components are derived from this class and are referred to as "EVTools".

Figure 5 illustrates the flow of an EventView analysis. After initialisation of EVTools (done by `ToolSvc` even if the `AlgTool` is private), execution is initiated by the EVToolLooper. An instance of EventView is created at the beginning of the event execution and subsequently passed to each EVTool through the argument of their `execute(EventView* ev)` method. This is repeated until all events have been processed. When multiple views are requested for the analysis, `EventViewMultipleOutputToolLooper` is used instead. This is a generalised version of EventView application manager with extra functionality to organise the lineage of EventView trees. In the execution, views are passed either separately or altogether in a container depending on the construction of the EVTool.

To handle multiple EventView instances, EVTools can be configured to be one of the three types available. The simplest is the single input tool, which receives one instance of EventView. Next level up is the multiple input tool to which an instance of EventView and an empty EVContainer is handed. New EventView objects created in the EVTool are pushed into the container, which are subsequently added to the main container held by the EVToolLooper. Multiple in, multiple out is the last type of EVTool, which receives the main container from the looper and an empty container to which all or subset of the input views and newly created ones can be inserted. This type of tool is useful for operations like sorting of existing EventView objects.

The EventView EDM class, the application manager EVToolLooper, and the `EventViewBaseTool` interface are the foundation of the EventView analysis framework (the design diagram is shown in figure 6). These specialised components significantly reduce the overhead of algorithm development and enables a whole suite of analysis environment to be built on top of them as seen in the next section. In fact, this structure is not unique to EventView: The same design pattern is seen in almost all Athena reconstruction sub-systems that employ object-oriented

[5] `DataVector` is much like `std::vector` but with support for memory management through `StoreGateService`.

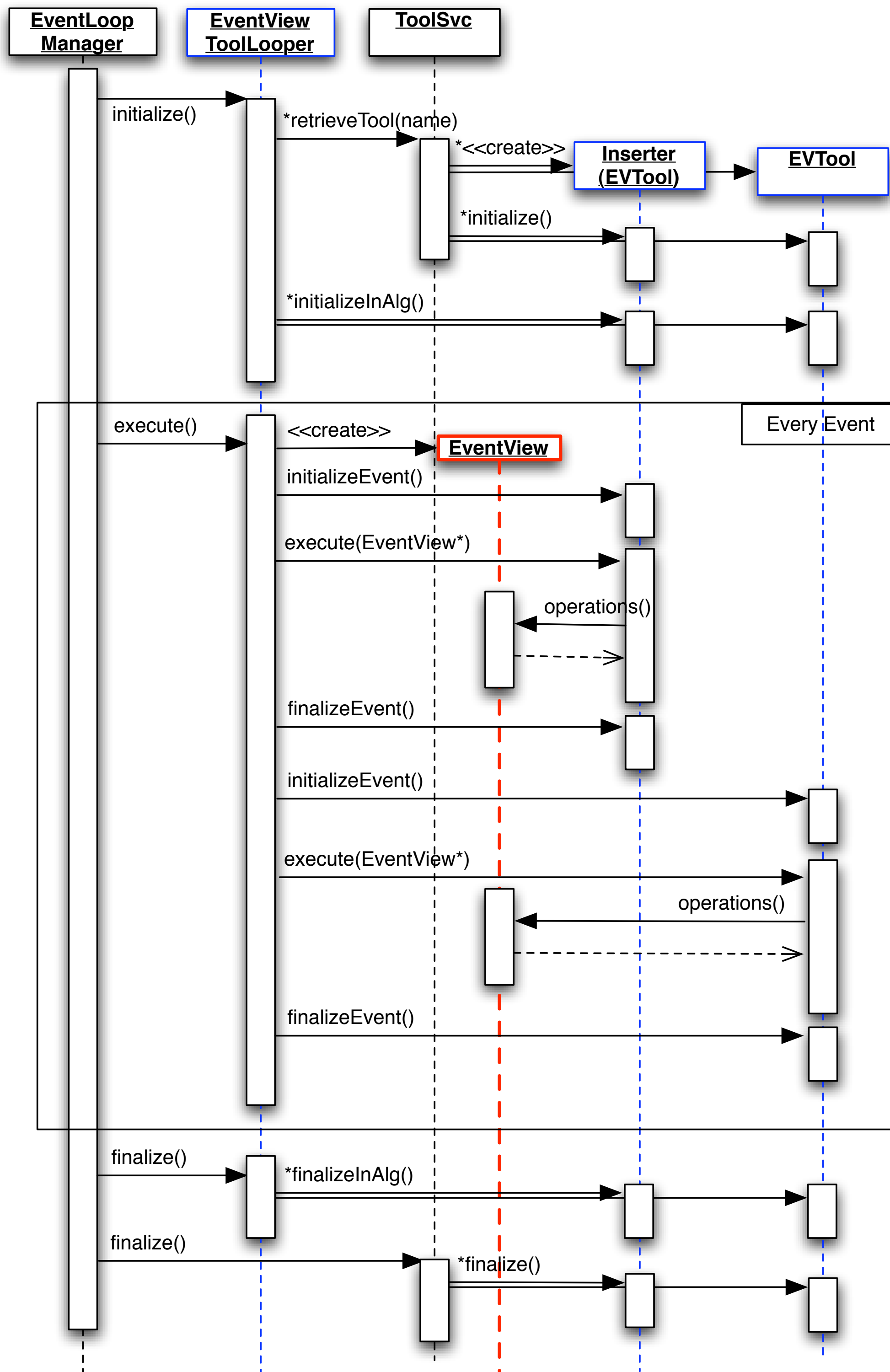


Figure 5: Sequence diagram of execution of EVTools as managed by an EVToolLooper.

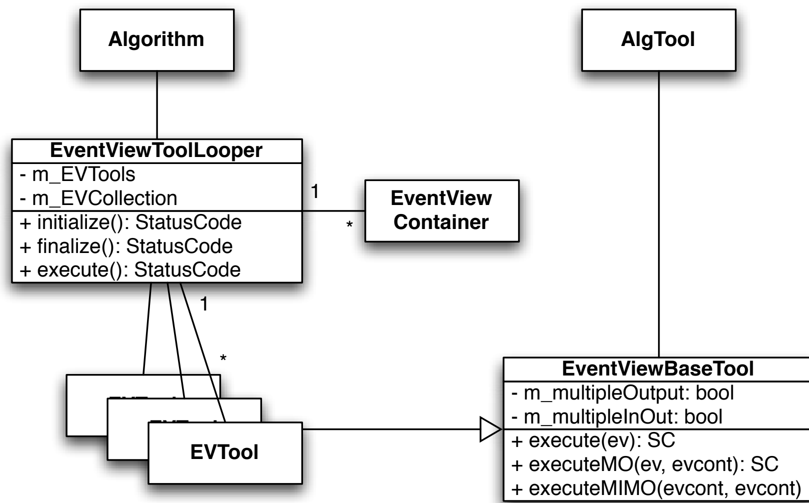


Figure 6: The core components of EVENTVIEW.

component model. In these frameworks, the data class is in the reconstruction EDM such as `TauJet` or `Track` instead of EventView.

3 EventView Analysis Toolkit

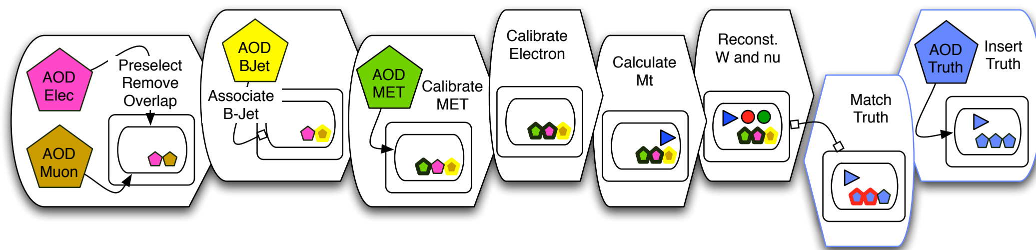


Figure 7: Schematic diagram of a modular analysis linked by the common EventView container.

The main components of the EVENTVIEW has already been fully described in the preceding sections and the main philosophical idea behind the design of the system has been introduced. The major implication of the construction of this basic foundation is that it enables one to factorise small steps of analysis into well defined modules, or EVTools. Analysis based on such modules are called "modular analysis" where a complex analysis is constructed out of small pieces of modules. There are numerous advantages to modular analysis that are specially vital to a large collaborative environment like ATLAS some of which are named below:

- **Organisation** - Different parts of an algorithm dedicated to specific tasks can be separated into different components, which are logically consistent within themselves;
- **Reusability** - Each component can be used multiple times in different contexts by applying suitable configurations. This reduces the duplication of coding required to construct analysis;

- **Uniformity** - Small specialised components can easily be shared by a number of users, which provides a common method for a common task;
- **Reliability** - Shared components will undergo numerous tests under different use cases and problems can be found and fixed effectively.

The implementation of modular analysis environment built around EventView is sketched roughly in figure 15. It is based on several main components. It illustrates the role of EventView as the carrier of analysis information, which flows through the chain of modules: after initial selection of objects from AOD, modification to final state objects is done to calibrate electrons. An event quantity, transverse mass here, is calculated and added to the `UserData` and finally, W boson and neutrino are reconstructed and added to the view as inferred objects. It also shows the interaction between multiple views, in this case a Truth EventView, which has information from the Monte Carlo Truth, which is compared to the objects in the reconstructed view.

The availability of EVTools exceeds well over a hundred covering most of the common operations forming a "toolkit" for in-framework analysis. Such proliferation was possible due to the object-oriented design of the sub-divisions of tools, which enabled efficient development of new EVTools. With this, the extendibility of the EventView is a natural feature that benefits the developers and the users alike.

3.1 Inserter Tools

Insertion of final state objects is the first step in most EventView analyses. This process involves preselection and removal of overlap between the objects considered for further processing (i.e. Analysis Preparation). Essentially, the process defines the view of the event by making decisions as to which objects have significance to the analysis. Such definition is highly dependent on the analysis and therefore flexibility is a main feature required for inserter tools. On the other hand, implementation of these tools can be made significantly simpler by using a base class for all inserter tools.

Certain operations in Athena require type specific handling of object containers, which prohibits one from writing a general non-templated implementation of the base class[6]. Aside from the technical problems, insertion generally depends on type specific information (such as shower shape of electrons). Nonetheless, all the type dependency and complexity involved in handling data containers is factorised into the templated base class, `EVInserterBase`. By inheriting from this class, most inserter tools were developed merely by implementing the virtual methods `preselect`, which defines the preselection procedure and `checkOverlap`, which specifies the logic for removing overlap.

The inserter tools have dependency on the EDM classes, which are subject to rapid development and subsequent changes. Concrete implementation of inserter tools therefore belong to a separate package `EventViewInserters` from its base class, which only depends on components, which undergo developments of much longer time scale. Hence, `EVInserterBase` and other base classes mentioned below are in `EventViewBuilderUtils` package, which only has dependency to the core components of Athena.

[6] Although most data objects have a common base class (`INavigable4Momentum`, which is an abstract representation of four momentum objects with virtual interface, which adds navigability to constituents) the containers they are in do not inherit from the container of the base class i.e. `Electron` class and `Muon` class both inherit from `INavigable4Momentum` but neither `ElectronContainer` or `MuonContainer` inherit from `INavigable4MomentumContainer`. Rather, they are concrete implementation of templated container class `DataVector`. A solution in Athena has been implemented and this is not an issue any longer.

3.2 Calculator Tools

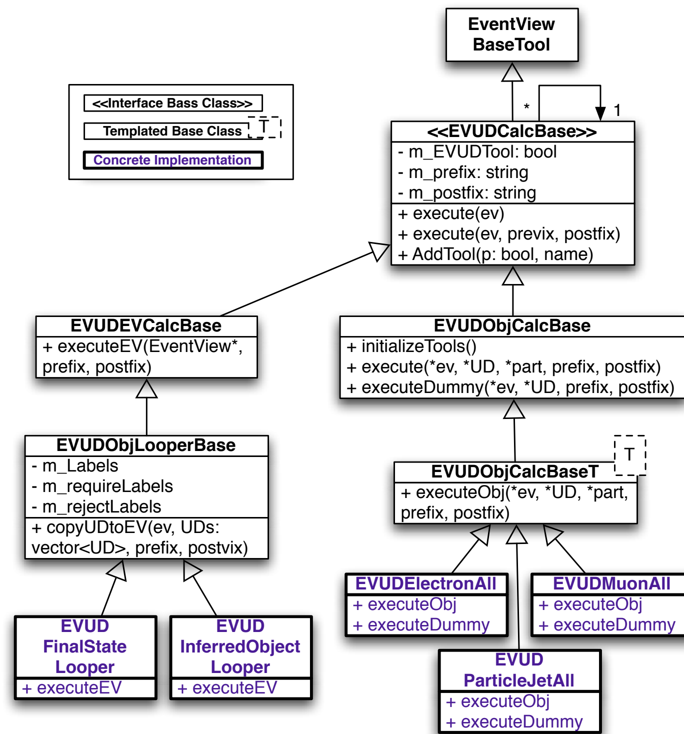


Figure 8: UML diagram of the design of the EVENTVIEW calculator tools.

Once Final State Objects are selected, one will need to calculate various quantities from them. Calculation in this context has two meanings: one is to extract information directly available in the EDM objects such as mass, energy and momentum and the other is to calculate secondary variables that needs algorithmic computation to obtain. The first is necessary primarily due to the unavailability of the access methods to such information from out-of-framework analysis otherwise. The current data persistification technology based on POOL uses ROOT format but its contents cannot directly be accessed using ROOT [7] as previously mentioned.

Reading information from EDM objects and copying to `UserData` enables producing such information in the format of ntuples merely by scheduling an EVTool, `EVAANTupleDumper`, at the end of the analysis as described in a later section. Methods to read information from each object can be generalised greatly thanks to the common base class from which EDM classes are derived. The only thing that requires coding in the concrete class implementation is the specification of access methods needed to obtain the variables specific to concrete EDM classes.

The base classes of calculator tools are equipped with the structure necessary to modularise variable calculation tools. The design UML diagram is shown in figure 8. As with all EVTools, the calculator tools derive from the `EventViewBaseTool`, which provides the most basic interfaces for `EventView`. The top base class of the inheritance tree of the calculator tools is the `EVUDCalcBase` class. This class, a virtual interface class, provides abilities to calculator tools so that sub-tools can be added to them. This is a useful feature in the organisation of variable calculators: a muon track information calculator may have a sub-tool, which calculates inner detector information and another tool, which calculates muon segment information. One may

[7] In other words, structured EDM objects of type such as `Electron` and `Muon` can be persistified using POOL and read back to ATHENA but not directly in a ROOT analysis.

choose to use these tools separately or as a single track information tool, which schedule them both.

The common usage of these tools is to loop over all objects of a certain type and obtain information from each of them, which will subsequently be stored in UD as vector of values. Looping of objects is generalised in the `EVUDObjLooperBase`, which is designed to schedule sub-tools derived from `EVUDObjCalcBase`. Concrete implementation of tools, which derive from `EVUDObjCalcBase` contain the class specific method needed to obtain information from each object passed from the `EVUDFinalStateLooper` or `EVUDInferredObjectLooper`. Therefore, looping of objects is completely de-coupled from the reading of information as it should be. This structure enables dynamic configuration of variable calculation so that one can add or remove calculator tools at configuration time to obtain specific information required in the output. Selection of objects to be looped can be specified by labels with additional logic such that one can require or reject objects with specified labels. Calculator tools are templated for the appropriate level of EDM inheritance. For example calculation of kinematic information, which is common to all vector like objects, can be done by a single tool irrespective of the concrete type of the objects to be dealt with as long as the object is a vector like object.

Naming of the variable is consistently organised through prefix and postfix variables propagated through the tools and appropriate prefix is added to the variable by the object looper tools. All electron variables would have "El_" prefix configured in the looper tool and the ordering of all electron variables are kept in synchronisation by the looper.

The second type of calculator tool, those that calculate secondary information fits into the same framework. A new tool of type `EVUDObjCalcBase` can be created to calculate variables based on arbitrary algorithm and scheduled in the object looper. Calculation of variables, which require more than one objects (such as taking the sum of the p_T of the jets) is not supported by the object-level calculator interface and one has to create a new tool directly deriving from `EventViewBaseTool`.

3.3 Associator Tools

Association between objects is another frequently occurring operation within an analysis. This happens in various context: matching a reconstructed object and a Truth object in AOD; matching a reconstructed object with a trigger object; associating an object with its constituents and associating one object in one EventView to an object in another EventView. Each of these require slightly different interface though appropriate level of modularisation can be achieved by capturing common patterns in the design of the base classes.

Associator tools are designed as object calculator tools with extended methods for association since association is done from a given object. The interface method called `executeMatch` is provided in the immediate derived class called `EVUDObjAssocBase`. This method is implemented in the subsequent derived classes, which define the access method for matching a given object to another. For example, `EVUDToEVAssocBase` implements methods to access another EventView in which a matched object is looked for. The concrete implementations specify the template EDM types required for association (e.g. `Electron` and `TruthParticle` for electron Truth match tool). Associator tools make use of the calculator functionality, which enable them to schedule sub-tools. Once a match is found with a specified method, one needs to calculate the information of that object using calculator tools. A typical use case therefore is for example, loop over all reconstructed electrons, look for the nearest Truth electron in the Truth EventView and calculate the kinematics of the matched Truth object and so on. This will create variables such as "El_Tru_p_T" (i.e. the p_T of a Truth electron that matched a given reconstructed electron)

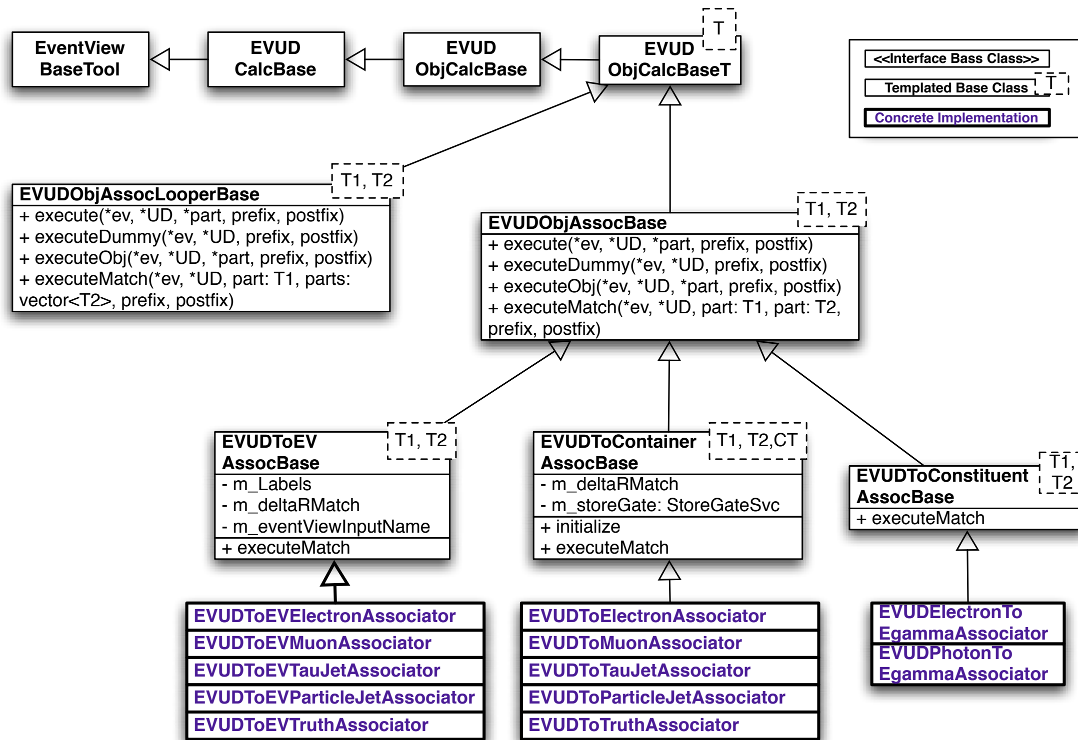


Figure 9: UML diagram of the design of the EventView associator tools.

for each electron found in the event. Additional implementations of associator tools are in `EventViewTrigger` package, which are used for matching trigger objects (e.g. Truth object to trigger object).

A part of the implementation of associator tools is shown in figure 9, which mainly shows the one-to-one matching tools. There is another group of tools, which are used for one-to-many matching. These tools inherit from the other branch, whose base class, `EVUDObjAssocLooperBase` is shown in the figure. The structure of the design is the same as that of one-to-one match tools. An example of such type of tools is constituent associator, which associates a composite object with its constituents.

3.4 Transformation Tools

During the course of analysis, the objects in EventView may need to be modified for various reasons. Calibration of object, merging of two objects into one and boosting of objects into other frames all require the FS objects to be modified. Since AOD objects are of constant type, one cannot modify the existing objects, but rather, one needs to create a new object and replace the existing one. In addition, EventView, being a data class, does not support replacement of objects by itself.

`EventViewTransformation` tools were developed to handle those situations where one needs to "transform" the FS and IO of EventView. A new EventView is created from the existing one without any FS or IO. For each object in initial FS/IO, new objects need be created, which replaces the old ones. As the class has to have an identifier to be stored in StoreGate and identifier is only given to container classes, new objects need to be inserted into containers of appropriate type. These common operations are done in the base classes of the transformation

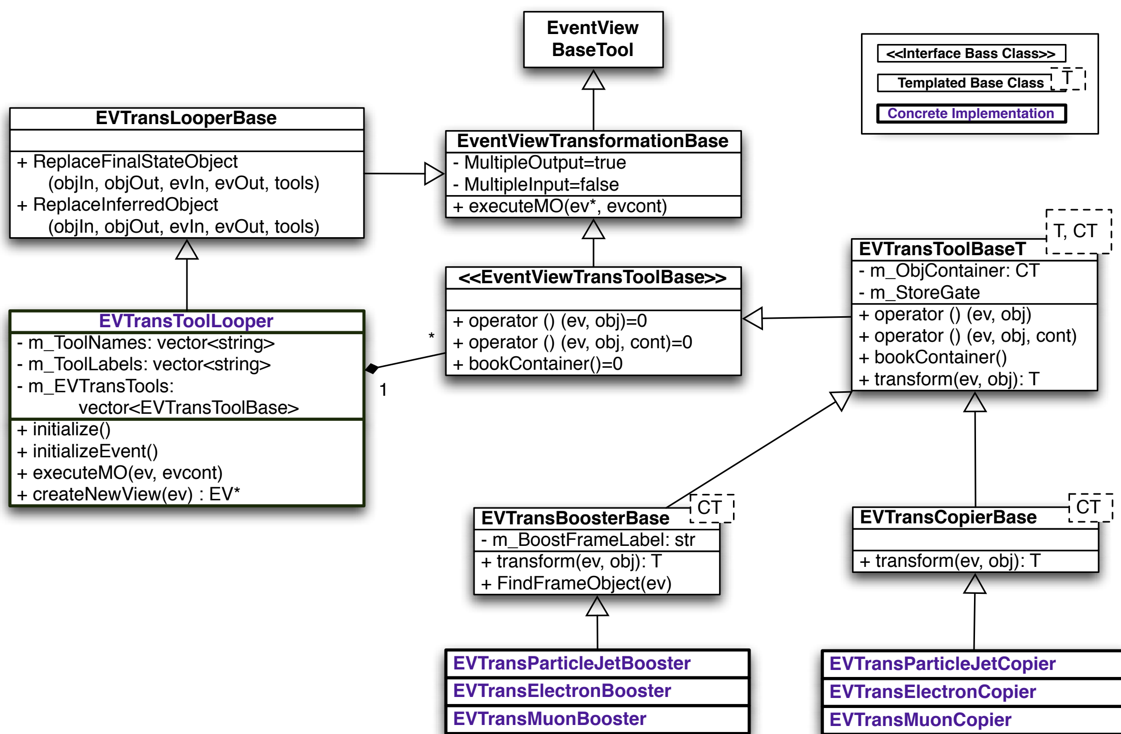


Figure 10: UML diagram of the design of the EventView transformation tools.

tools as shown in figure 10.

The design pattern follows that of calculator tools in which there is a common base class for the object looper and the object tools, in this case called `EventViewTransformationBase`. The object looper, `EVTransToolLooper` handles the retrieval of objects based on labels and scheduling of object tools as specified through run-time configuration. Object tools are functors, which implement the operator, "()". When the operator is called, the class method `transform` is forwarded to the looper, which passes one object to the method at a time. Therefore, the concrete classes of transformation tools merely implements the logic for replacing one object with a new one and the rest is handled by the underlying structure.

Similar to the calculator design, `EventViewTransToolBase` is a non-templated interface class. Since the concrete classes need to deal with a range of EDM classes, the implementations of the interface is templated. `EventViewTransToolBase` identifies the object tools regardless of their concrete types and provides uniform access of the functor interface to `EVTransToolLooper`.

3.5 Dumper Tools

Calculator tools and associator tools are good examples of the common type of tasks required for the analysis. With these one prepares the information needed in further analysis. For example, efficiency and purity of object reconstruction can easily be calculated once association with Truth has been done and p_T, η dependency of such quantities can be plotted as long as these quantities have been calculated.

Dumper tools are used to output the information stored within EventView for inspection or further usage in external analysis. This may be a simple screen dump (figure 12), which prints

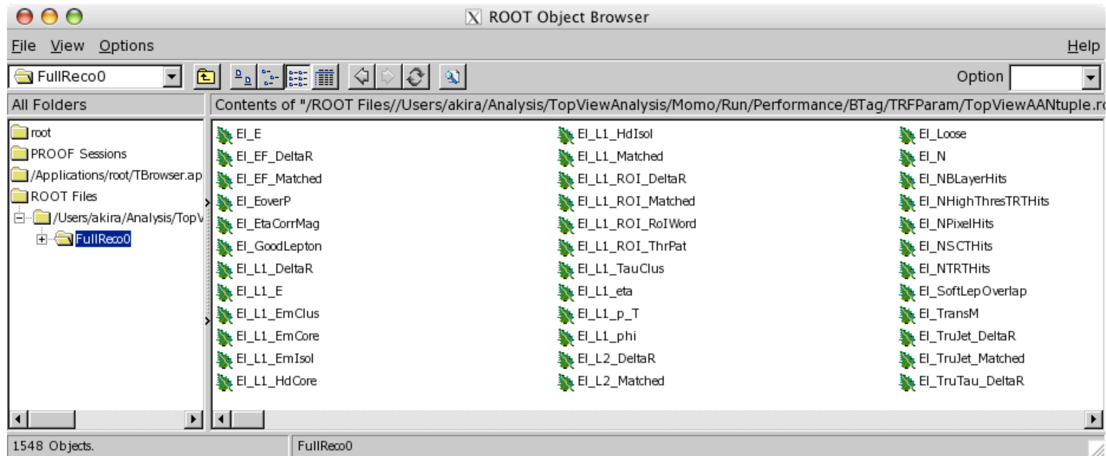


Figure 11: ROOT browser showing the contents of ntuple produced in EVENTVIEW.

out the contents of FS/IO or UD in EventView, XML file that can be used in ATLANTIS [17] event display (figure 13), or ntuple that can be read into stand-alone ROOT analysis (figure 11).

Due to the diversity of the output formats, there is no common interface for dumper tools except that they inherit from `EventViewBaseTool` and the details of implementation of such tools are highly dependent on the technology used for writing out the data. For e.g., ntuple format is written out using `TTree` objects of ROOT. Writing of great number of variables can be a time consuming computing operation. The design of UD is particularly relevant to this and its design has been optimised using run-time profiling.

3.6 Other Tools and Toolkit Development

In addition to calculator, associator, transformation, and dumper tools, numerous other tools have been developed, some of which are:

- selector tools: Apply event selection and print cut flow table summarising the efficiency. One can apply cuts on EventView in case selection fails and terminate the event processing.
- combiner tools: Combine multiple objects into one `CompositeParticle`. In case there are multiple combinations, one can choose to produce multiple EventView objects each containing different combination.
- sort tools: Sort multiple event view according to arbitrary criteria such as mass of combined objects or χ^2 of constrained fit. Comparison logic is contained in separate tools similar to functor approach in transformation tools. Therefore, comparison logic can be replaced through run-time configuration.
- thinning tools: thinning is the process of keeping only selected objects in a container (e.g. good electrons), and is an important step in the creation of POOL-based DPD.

These tools are all based on the core components that have been introduced already. Applicability of core components to wide range of application shows robustness of the fundamental design principle of EVENTVIEW. For very common type of tools such as object calculator tools, scripts have been written, which generate skeleton C++ code and development of EVTools is

```
---------------- Final State Objects ----------------
Object 0:  p_T = 55461.3 phi = -2.18794 eta = 1.04014 type = Analysis::Muon
   Labels: Lepton  Muid  Muon  Tight
Object 1:  p_T = 10807.9 phi = 0.978227 eta = 0.17026 type = Analysis::Muon
   Labels: Lepton  Loose  Muid  Muon  Tight
Object 2:  p_T = 238032 phi = -1.54079 eta = 1.69019 type = ParticleJet
   Labels: CentralJet  Cone4  HardJet  ParticleJet
Object 3:  p_T = 160690 phi = 1.54603 eta = 0.308174 type = ParticleJet
   Labels: CentralJet  Cone4  HadronicTopDaughter  HadronicWDaughter  HardJet  ParticleJet
Object 4:  p_T = 115243 phi = 2.01035 eta = 0.720166 type = ParticleJet
   Labels: CentralJet  Cone4  HadronicTopDaughter  HadronicWDaughter  HardJet  ParticleJet
Object 5:  p_T = 88584.7 phi = 1.00186 eta = 0.0929353 type = ParticleJet
   Labels: BTagged  CentralJet  Cone4  HadronicTopDaughter  HardJet  ParticleJet
---------------- Inferred Objects -------------------
Object 0:  m = 814805 p_T = 67360 phi = 1.81452 eta = 3.22118 type = CompositeParticle
   Labels: AllObjVectSum
Object 1:  m = 168641 p_T = 339570 phi = 1.56295 eta = 0.425561 type = CompositeParticle
   Labels: Top  TopWithHadronicW
Object 2:  m = 91709.3 p_T = 268734 phi = 1.73927 eta = 0.502074 type = CompositeParticle
   Labels: HadronicW  W
```

Figure 12: Screen dump of an EventView.

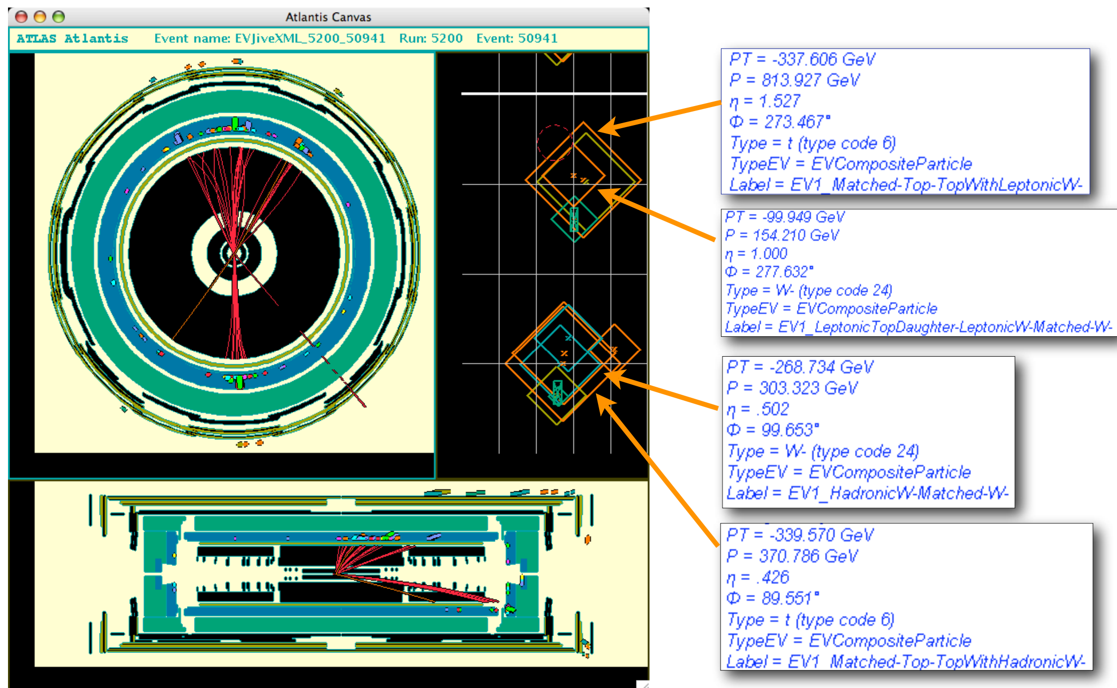


Figure 13: ATLANTIS event display showing the contents of EVENTVIEW.

much simplified to the extent that one only needs to implement one or two class methods to create a new EVTool.

Later evolution of the EventView is largely a matter of adding new tools to the framework. The tools introduced so far are rather general that they can be used independent of the context of the analysis. For example, the combiner tools can be used to reconstruct top quarks, or the Higgs. However, a number of tools more specific to analysis context have also been developed and maintained within physics working groups. Several packages called have been developed for this purpose as shown in section 4.3.

3.7 EventView Configuration and Modules

Development of new EVTools is necessary to perform specific tasks required for specific analysis. However, it often suffices to use general tools like the ones introduced so far and configure them to do specific tasks. It is advantageous to leave the EVTools as general as reasonably possible so that similar tasks are always handled by common tools. In the light of this spirit, the algorithmic part of the analysis should be well separated from the variable parameters of the algorithm as much as algorithms are separated from data within the framework.

Therefore, configuration is a significant aspect of analysis, which stems from the analysis context in which the tools are used. As EventView algorithms became more generalised, development of analysis shifted more towards run-time configuration than compiled algorithms. It became evident that configuration has to be performed in a well defined and well structured manner.

In Athena, both `Algorithm` and `AlgTool` have interface to declare configurable "property". The bridge between the variables in C++ and configuration in Python is the C++ reflection technology, Reflex[18], which enables run-time introspection of C++ objects through automatically generated Python bindings. Since Python is a fully-fledged programming language with support for object-oriented structure, it is fully equipped with the ability to define methods for structured run-time configuration.

Figure 14 shows the design diagram of the EventView configuration classes. All the components shown in this diagram is in the region of run-time configuration written in Python and the configuration set through this mechanism is eventually propagated to the C++ `Algorithms` and `AlgTools` as described above. Such structure is necessary since the run-time configuration is performed based on pre-run-time (or the configuration-time) manipulation of properties on the Python side, which happens before the actual instantiation of C++ objects[8].

The job steering must reflect the instantiation of the C++ objects in structured manner. For each C++ object to be instantiated in the job, one Python representation is created. These objects act as "property proxy", which are place holders of the configured properties of the corresponding tools (shown with bold line in the diagram). Much of the functionality is provided by the "configurable" scheme (shown in green in the diagram) and additional functionality are added or overridden by the EventView configuration package (shown in black), which inherit from the configurable classes.

The base class for all EVTools is `GenericEventViewTool`. It has the ability to add property

[8]This is due to various practical issues: Configuration of components may require re-initialisation depending on the context while such method is not always implemented; Some parts of configuration is order dependent and one cannot guarantee ordering at run-time; Loading of dynamic libraries takes large amount of time; and so on. Gaudi was never meant for real run-time usage and its behaviour is not very well defined unless proper configuration is ensured pre-run-time.

Figure 14: UML diagram of the design of the EVENTVIEW configuration.

proxy objects on-the-fly at the configuration-time[9]. They are instantiated by the alias `anEVTool` with the name of the EVTool to be created and are added to the Python instance of `EVTool-LooperBase` (whose Python representation is treated separately since it is an `Algorithm`) or the object looper. Each declared property can be configured through the `anEVTool` Python instances as if one has obtained the handle to the run-time instance of the objects. The `setup()` method defined in `Configurable`[19] triggers the property proxies to be propagated to the C++ instances and at that point the execution of the application advances to the run-time phase starting with initialisation.

In addition to structured organisation of analysis configuration, the EVENTVIEW configuration layer adds a new entity called `EventViewModule` ("EVModule"). It is a Python class, which holds one or more EVTools with a specific configuration. For example, one can build an EVModule, which represent top quark reconstruction, which in turn consists of some object selection and combiner tools. It may be done with the standard tools in the toolkit though configuration has to be done appropriately. In an EVModule, EVTools and their configuration can be put together to form a context dependent analysis object. The design behind EVENTVIEW configuration ensures such objects can be treated in the same way as EVTools. This means that one can schedule EVTools and EVModules to the EVToolLooper on an equal footing as shown in figure 15. It is also possible to create an EVModule with a collection of object calculator tools, which is schedule to an object looper tool.

EVModules are also good machinery to realise high level logic associated to configuration-time operations. One can write an analysis, which makes use of a large number of EVTools simply

[9]This is a temporary solution before the configurable scheme is in full swing. There is a `genconf` mechanism, which generates corresponding Python configuration class for each C++ EVTool class and list of their properties. In ATHENA release 13, ad-hoc generation of property proxy is no longer necessary as currently done with the `anEVTool` instance. This is why the Python instance of `AlgTool` is marked "old style".

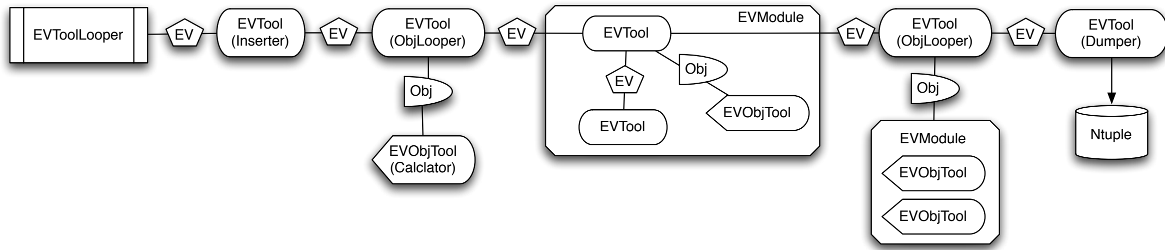


Figure 15: Schematic diagram of EVTools and EVModules as scheduled by EventViewToolLooper in an analysis.

by instantiating a few EVModules, which automatically creates and configures EVTools. For example, an electron information module may perform full set of analysis on electron ranging from inspection of reconstructed track to calculation of trigger efficiency.

4 Role of EventView in Atlas

4.1 Package Management and Organisation

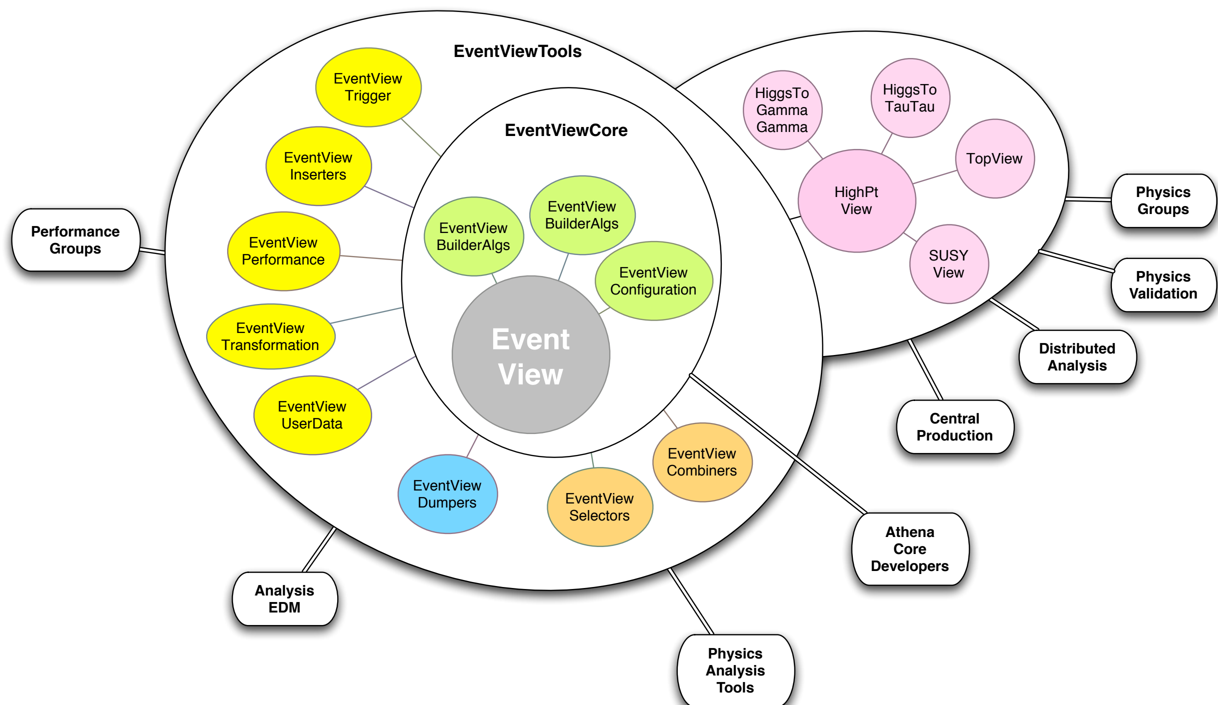


Figure 16: Package organisation of EventView and relationship with working groups.

As shown in the previous section, EventView is a sub-system of Athena with a rich collection of generalised algorithms built around an analysis data object. It has proven its relevance to

various working groups within the collaboration and in many cases it is acting as functional connection between them. Figure 16 illustrates the relationship between various working groups and EventView packages. The main part of the development activities is within the scope of Physics Analysis Tools (PAT) group where requirements to the framework and technical design solutions are discussed. As there is a continuous development of the output DPD data format, the dumper tools have to adopt to the latest persistification technology. Development of core components often needs assistance from the Athena core developers.

Inserter, trigger, transformation, and UserData tools are relevant to performance groups where object reconstruction, calibration and selection are studied. Input from the performance groups is used to develop appropriate algorithms and configurations. These are readily available to physics groups who seek to improve the analysis by making use of the latest performance study. "PhysicsView" packages incorporate such new features into analysis in a well defined manner as described in the next section. These packages put together the baseline analysis specialised to the requirement of each group, which is subsequently sent to distributed analysis for batch processing of AOD datasets. The output of the analysis is used in physics validation where the performance of the reconstruction and analysis software are check regularly.

4.2 EventView and Atlas Analysis Model

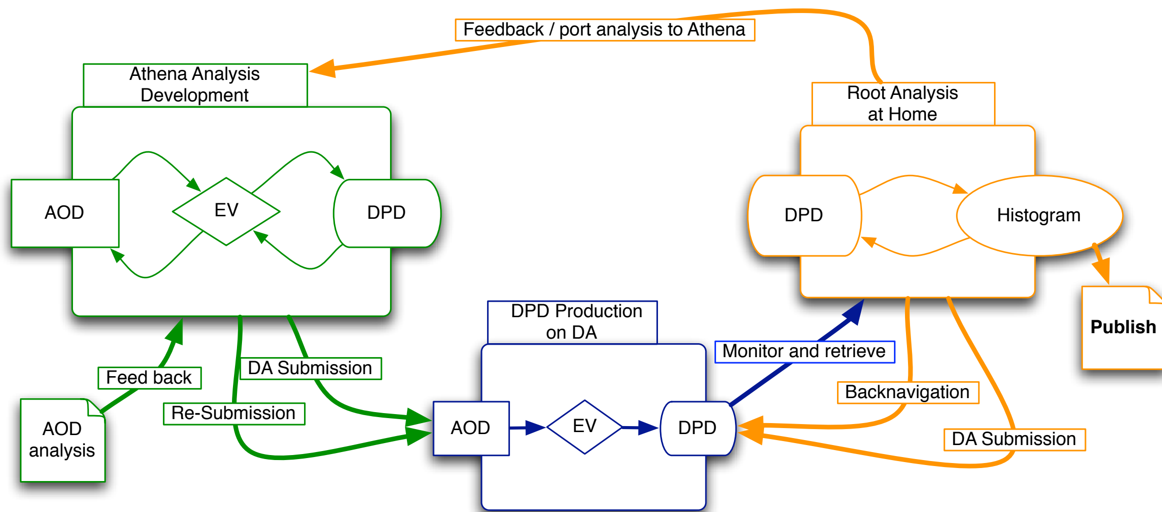


Figure 17: Athena/EventViewanalysis model within two-stage analysis involving Grid processing.

While EventView is a powerful analysis framework and is playing a vital role in the Atlas analysis model, there is a large amount of analysis to be done outside the framework. What is called "analysis" includes various stages of data processing. We make an attempt to make four crude sub-divisions:

1. Reconstruction: Event reconstruction of raw data, which includes object reconstruction (towers, clusters, tracks, jets and so on) and particle identification (electron, b-tagging, τ jet etc). Therefore this step is typically an object-level analysis in contrast to the event-level analysis in the third step. This is performed centrally using Athena and the result is persistified in ESD and AOD.

2. Analysis Preparation: First step of physics analysis. One defines the view of an event by defining final state physics objects from the list of objects created by reconstruction. This

mainly involves preselection and overlap removal of objects and prepares the next, event-level, analysis stage. Some of the pre-physics analysis may be re-done at this point when it is necessary to apply corrections to improve the quality of reconstructed objects. This is necessary since full reconstruction cannot be repeated very frequently and additional refinement on ESD/AOD becomes available between two production cycles.

3. Event-level physics analysis: Once final state objects are defined, one can do further analysis such as taking combinations of reconstructed objects to reconstruct inferred objects. In addition, event and object-level variables (sphericity, H_T, etc) can be calculated at this level.

4. Sample-level analysis: Physics analysis which requires a global view of several samples over a number of events. This includes plotting histogram, fitting templates, study with toy MC and so on and publishable results are produced at this stage.

The distinction above is not necessarily well defined or mutually exclusive though one can see how each step is processed in terms of computing. The first step is a central production (on the Grid computing resource). Baseline analysis is the first step in physics analysis and is one of the major scopes of EventView. EventView has a collection of tools useful for further analysis and some or all of event-level analysis can be covered. This, however, may also be done as out-of-framework analysis based on DPD produced from baseline analysis. Typically parts of event-level analysis is done in EventView and the rest in Root (or any other out-of-framework analysis package). Finally, analysis which requires information over the whole sample is not well performed in Athena which is intrinsically optimised for event-level processing. Therefore, sample-level analysis is typically performed outside the framework.

Analysis which require AOD as an input may not be run locally due to the size of AOD. In view of distribution of DPD produced from a common baseline analysis (and some of event-level analysis) such process should be done on a common resource on the Grid through distributed analysis (DA) services. On the other hand, further processing of DPD usually requires a highly interactive analysis environment which can only be performed locally. Therefore, user level physics analysis is roughly divided into these two sides which are bridged by DPD produced on DA resource. Figure 17 summarises this pattern of data analysis. In practice, one analysis would have to repeat the whole process several times until a satisfactory result is obtained. This forms a kind of feedback loop in the analysis model where the result of local analysis improves the next round of the in-framework analysis.

4.3 PhysicsView Packages

The PhysicsView packages are at the point of interaction between EventView and physics groups; thus they have particular importance. Extendibility is a natural feature of EventView analysis framework and new EVTools and EVModules can readily be produced. Often these developments are specific to certain physics scenario in which case they tend to be closely related. A PhysicsView package is a collection of related tools under a given analysis context. It is a play ground for collaborating physicists to construct one or many common baseline analysis which can also be used to produce common DPD.

EVTools that are specific to analysis context are developed within these packages and specific configuration of general tools are stored in the form of EVModules. This includes object selection for the analysis, definition of output data structure, variable calculators, object reconstruction tools and so on. In particular, object selection and output structure (which depends on the configuration of calculator and associator tools) can be abstracted so that the same interface is

available to all PhysicsView packages. General frameworks for these tasks have been developed in `HighPtView` package together with default set of configuration which forms a reasonable baseline for most high-p_T physics analysis. Each group can start by taking this package as a template and override its settings as appropriate.

4.4 Case Study - TopView

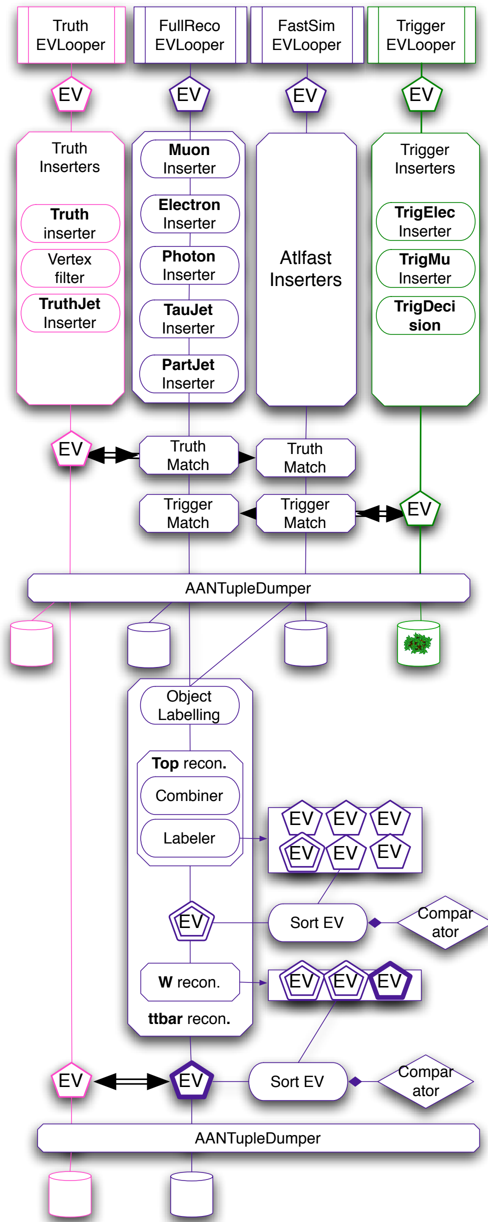


Figure 18: Schematic diagram of default `TopView` analysis which includes baseline analysis (top half) and $t\bar{t}$ reconstruction (bottom half).

A `PhysicsView` package for the top physics working group, `TopView` [20], fully illustrates the ideas of EventView analysis model. Figure 18 shows some of the important parts of the default analysis job which is used to produce the common DPD from AOD for the whole group. The baseline object selection is done by overriding those in `HighPtView` through discussion with the performance group and is based on the common EV-Tools from the default toolkit for the most part. One exception is Truth particle insertion which requires special care to identify relevant objects for top study using vertex filtering[10]. Truth, full reconstruction ("FullReco"), fast simulation ("FastSim") and trigger analysis are run in parallel and matching between them are performed after the insertion of objects have been completed (each type of insertion is done by one module as shown in the figure.) At the same time, information of FS objects and matched objects are calculated to the level of detail needed in the local Root analysis. At this point the data in UD of each EventView is dumped into separate Root trees using the ntuple dumper.

After the baseline analysis, the job proceeds to perform a $t\bar{t}$ analysis known as "Commissioning Analysis" [21] which is widely studied in the top group for the first LHC data. The analysis consist of a simple top reconstruction which combines three jets in an event and select the combination with the highest p_T. From this combination, all dijet combinations of three daughter jets are computed and again, the highest p_T combination is selected.

Since object selection of Commissioning Analysis is tighter than that of the baseline TopView selection, the objects are labelled if they passes the additional cuts ("Object Labelling" in the diagram). Subsequently, the labelled objects are combined and multiple EventViews are produced, each of which has one reconstructed top candidate. The views are sorted according to the p_T of their the candidate and the first one is kept for the next step[11]. From the selected Event-

[10]Vertex filter tool looks for some patterns in the decay chain and inserts those into EventView. In `TopView` the pattern includes $t \to W + b$ and $W \to e + \nu$.

[11]Note, one can chose to keep all combinations and save them to the output ntuple though this is not done in the default job.

View, a W boson is reconstructed as described above, and this time sorting is done on the W candidate's p_T. Finally, the kinematics of the reconstructed candidates is calculated and the selected view is matched to the Truth EventView to determine if the reconstruction was successful. These are finally written out to separate ROOT trees.

Note that the each part of the analysis is confined to separate modules be they EVTools or EVModules. This makes the analysis very flexible. One can easily replace selection cuts or algorithm used for the reconstruction of the object. If one wants to apply calibration to some of the objects, that can also be done without disturbing the rest of the analysis. Each module is reusable and same top analysis is used for both FullReco and FastSim making comparison of the two a trivial task.

The default `TopView` analysis job is sent to the computing Grid through the PANDA [22] distributed analysis service where all specified datasets are processed using the same analysis. Ntuples are produced and made available through Distributed Data Management (DDM) system and further analysis can be done locally based on these. Figure 19 shows the result of the Commissioning top analysis combining the $t\bar{t}$ signal and W+jets background. Sample-level analysis has been performed to measure the level of background using curve fitting and a Gaussian fit to the signal peak is shown as the dashed line. Since top reconstruction is done in the EVENTVIEW analysis, final analysis of this level can be done with little complexity.

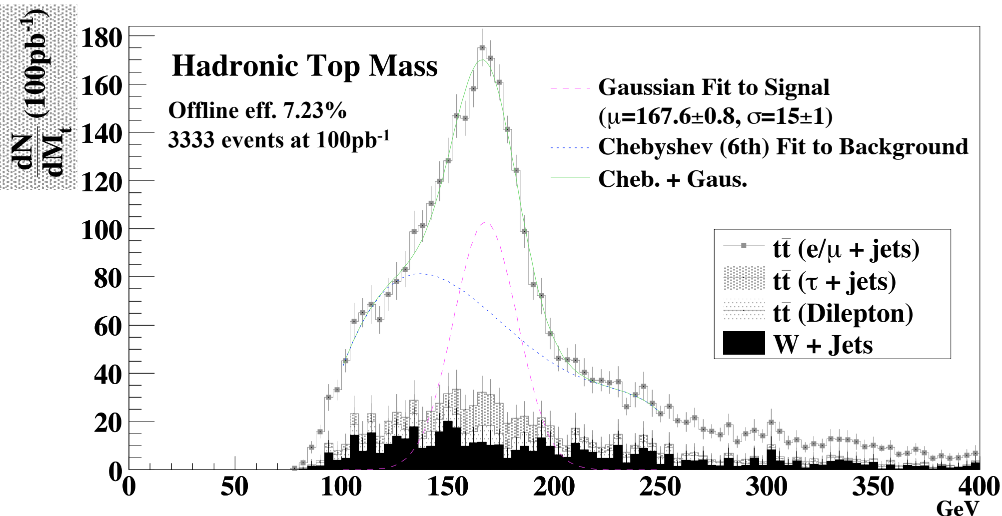


Figure 19: The mass of the top quark as reconstructed by the `TopView` default job using the Commissioning Analysis procedure. The black filled histogram is the contribution from W+Jets background. The coloured lines are the result of curve fitting using a Chebyshev polynomial and a Gaussian.

5 Summary

The ideas and the design behind the EVENTVIEW analysis framework was summarised in this paper. Several key concepts were introduced with respect to the components in EVENTVIEW.

- EventView EDM class: The data class in the EVENTVIEW framework. It is based on three types of sub-containers, Final State Objects, Inferred Objects and UserData and it acts as a *blackboard* within the component model setting the *language* of algorithm writing.

- EVToolLooper: An application manager in EventView analysis. Being an Athena `Algorithm`, it manages the sequence of EVTools scheduled for the analysis and also manages the flow of the EventView object throughout the analysis.

- EVTool: An Athena `AlgTool` with EventView interface derived from `EventViewBaseTool` which implements the interface needed for algorithms written for EventView analyses. Most of the algorithmic part of the analysis is in EVTools. Special care is taken to make the EVTools as general as possible and specific object-oriented structures were developed to achieve this as seen in calculator and associator tools.

- EVModule: A logical entity which consists of one or more the configured EVTools. It sets the variable parameters of generic EVTools and configures their behaviour within the context of the analysis.

- PhysicsView: A package with a collection of EVTools and EVModules which are closely related to a physics analysis context. Full baseline analyses are constructed in these packages which are used for common DPD production.

In summary, EventView is a suite of programs with a robust component model, which forms a general framework for physics analysis in any context. It has successfully identified a paradigm for a collaborative analysis model and its solution has proven to be relevant to functional physics analysis in the Atlas collaboration.

References

[1] R. Burn and F. Rademakers. Root - an object oriented data analysis framework,. In *AIHENP'96 Workshop*, volume A. 389, pages 81–86, Lausanne, Sept 1997. Nucl. Inst. Meth. in Phys. Res.

[2] ATLAS Collaboration. *ATHENA, The ATLAS Common Framework - Developer Guide.*

[3] ATLAS Computing Group. Technical design report. *ATLAS TDR*, 2005.

[4] B. Stroustrup. *C++ Programming Language.* Addison Wesley, 1997.

[5] G. Barrand et al. GAUDI - a software architecture and framework for building LHCb data processing applications. In *CHEP 2000*, 2000.

[6] D. Quarrie, A. Shibata, et al. Taming the beast: Using python to control ATLAS software. In *Europython 2006*, 2006.

[7] L. Tao, X. Fu, and K. Qian. *Software Architecture Design - Methodology and Styles.* Number ISBN 1-58874-621-6. Stipes Publishing L.L.C, 2006.

[8] T. Cornelissen et al. Concepts, design and implementation of the ATLAS New Tracking. *ATLAS Notes*, com-soft(002), 2007.

[9] I. Papadopoulos et al. POOL - the LCG persistency framework [online]. Available from World Wide Web: `http://pool.cern.ch/talksandpubl.html`.

[10] A. K. Assamagan, D. Barberis, D. Contanzo, et al. Final report of the ATLAS AOD/ESD definition task force [online]. 2004. Available from World Wide Web: `http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/domains/Reconstruction/AODESD/AODESDtaskforce.htm`.

[11] D. Adams et al. The ATLAS computing model. *ATLAS Notes*, SOFT(2004-007), 2005.

[12] S. Asai, A. Shibata, et al. Physics analysis tools 2006 workshop summary report. *ATLAS Notes*, ATL-SOFT-PUB(005), 2006.

[13] S. Snyder et al. ATHENA ROOT access [online]. Available from World Wide Web: `https://twiki.cern.ch/twiki/bin/view/Atlas/AthenaROOTAccess`.

[14] S. George et al. Final report of the ATLAS reconstruction task force. *ATLAS Notes*, SOFT(010), 2003.

[15] F. Akesson et al. Outcome of UCL workshop on the architectural aspects of physics analysis in athena. *ATLAS Notes*, 2004.

[16] F. Akesson et al. Physics analysis tools workshop summary report. *ATLAS Notes*, 2005.

[17] ATLANTIS team. ATLANTIS event display for ATLAS [online]. Available from World Wide Web: `http://www.hep.ucl.ac.uk/atlas/atlantis`.

[18] S Roiser. The SEAL C++ reflection system. In *CHEP 2004*, Sept 2004.

[19] W. Liebig et al. Physics-level job configuration. In *CHEP'06*, volume I, pages 446–449, Mumbai, India, Feb 1996. Available from World Wide Web: `https://twiki.cern.ch/twiki/bin/view/Atlas/PropertyRepository`.

[20] A. Shibata. TopView - ATLAS top physics analysis package. *ATLAS Notes*, SOFT-PUB(002), 2007.

[21] S. Bentvelsen and M. Cobal. Top studies for the ATLAS detector commissioning. *ATLAS Notes*, ATL-PHYS-PUB(024), June 2005.

[22] T. Wenaus et al. The PanDA production and distributed analysis system [online]. Available from World Wide Web: `https://twiki.cern.ch/twiki/bin/view/Atlas/PanDA`.