ATL-SOFT-PUB-2006-004
23 July 2006

ATLAS Tracking Event Data Model

Version for AtlasOffline release 12.0.0

F. Akesson[1], T. Atkinson[2], M.J. Costa[1], M. Elsing[1], S. Fleischmann[3], A. Gaponenko[4], W. Liebig[5], E. Moyse[6], A. Salzburger[1,7], M. Siebel[1]

July 20, 2006

[1]CERN, PH department, Geneva, Switzerland
[2]University of Melbourne, Australia
[3]FB C, University of Wuppertal, Germany
[4]Lawrence Berkeley National Laboratory, Berkeley, USA
[5]NIKHEF, Amsterdam, the Netherlands
[6]Dept. of Physics and Astronomy, University of Massachusetts, Amherst, USA
[7]Leopold-Franzens-Universitaet Innsbruck, Austria

Abstract

In this report the event data model (EDM) relevant for tracking in the ATLAS experiment is presented. The core component of the tracking EDM is a common track object which is suited to describe tracks in the innermost tracking sub-detectors and in the muon detectors in offline as well as online reconstruction. The design of the EDM was driven by a demand for modularity and extensibility while taking into account the different requirements of the clients. The structure of the track object and the representation of the tracking-relevant information are described in detail.

1 Introduction

For large collaborations like the ATLAS experiment common interfaces and data objects are a necessity to ensure easy maintenance and coherence of the experiment's software platform over a long period of time. The ATLAS Event Data Model (EDM) improves commonality across the detector subsystems and subgroups such as trigger, test beam reconstruction, combined event reconstruction, and physics analysis. The object-oriented approach in the description of the detector data allows the possibility to have one common raw data flow. Furthermore the EDM encourages the use of common software between online data processing and offline reconstruction. The need for these fundamental features has been recognised by the ATLAS reconstruction task force who has proposed a new prototype design for the entire ATLAS reconstruction software [1].

One important component of the ATLAS EDM is a common track class which is used for combined track reconstruction across the innermost tracking subdetectors and in the muon detectors. This document explains the structure of the track object and its contents.

1.1 Data Flow in the Track Reconstruction Software

The ATLAS software framework ATHENA [2] makes a clear distinction between event data classes and algorithmic classes. This is used to make the access and handling of event data totally independent of the reconstruction algorithms. At the level of the tracking EDM this implies that a track class would not contain code to perform track fits, but a fitting tool would use track data objects as input and output. Common interfaces to objects representing detector data at different stages of the reconstruction help structuring the reconstruction process as a highly modular data flow. As a result the clients have minimal mutual dependencies and a low interfacing load. In addition the possibility is given to store the event information on disk at different stage of the reconstruction, i.e. at the level of raw data, calibrated hits, particle trajectories and finally light-weight but concise physics objects.

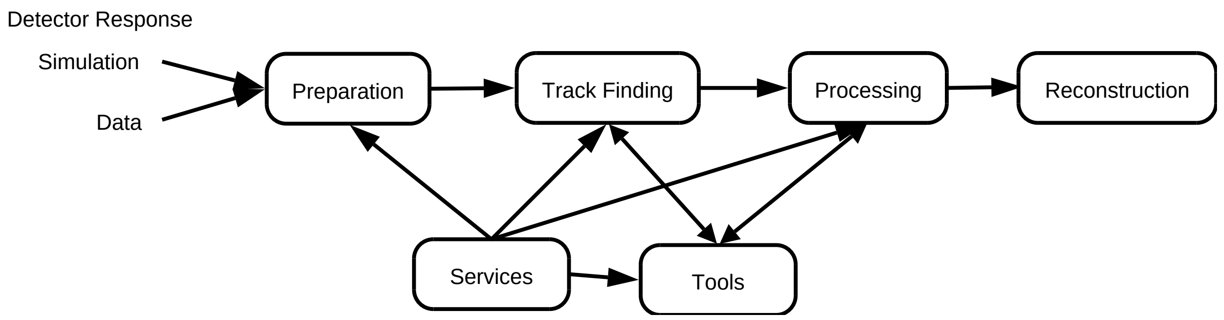


Figure 1: *The simplified stages involved in track reconstruction. The detector information coming from either simulation or real data is prepared for the reconstruction using common classes. The track finding and the subsequent handling of the tracks can then use common services and tools due to the mutual interfaces.*

Figure 1 shows a simplified view of the data-flow and various processes involved in track reconstruction. The data originating from either simulation or real detector response is prepared by applying measurement and geometric calibrations, depending on which part of the detector the information is coming from. The silicon pixel detectors and the semiconducting tracker (SCT), as part of the *Inner Detector*, and the cathode strip chambers (CSC), the resistive plate chambers (RPC) and the thin gap chambers (TGC), as part of the *Muon Spectrometer*, are using *clusters* and *space points* (with different concrete implementations). The transition radiation tracker (TRT) and the monitored drift tubes (MDT) provide *drift circles*.

All different types of clusters and drift circles are combined under a common base class, `PrepRawData`, which forms the input objects to the reconstruction, that is for finding and fitting tracks. This base class structure and its equivalents at later stages during the track reconstruction allow the definition of a common container class, the track, which is the output of the high-level pattern recognition algorithms. Those tracks are then passed through various processing stages: for example, ambiguity resolution for shared hits can be used to improve the quality of the final track collections; while primary and secondary vertex finding is performed to improve the overall quality of reconstruction of the event. For the physics reconstruction particle objects are created from the tracks. Due to the common interfaces a variety of tools, (e.g. track extrapolation from one detector surface to another) and services (e.g. the material service, giving information about detector material in the path of the track), can be employed throughout the reconstruction like drafted in Fig. 1.

1.2 Track Model in ATLAS

The concept of a common track object implies that the basic track must contain information describing the path of the track in the detector. Depending on the algorithm creating the tracks, the information content of the tracks can differ. The track can be filled with one or multiple sets of parameters, describing the track trajectory and position at different surfaces. This is particularly important for track representation in ATLAS because of the very inhomogeneous field when going from the Inner Detector solenoid to and through the toroid magnet system of the Muon Spectrometer. A simple helical track model would not have been sufficient. The track must also be flexible enough to handle both local and global position coordinates. Further it must be possible to store information regarding the quality of the overall track fit. This concept will be expanded in Section 2.

Frame Definitions In this document as well as in the source code the global frame (also called tracking frame) will denote the three-dimensional orthogonal frame, (x, y, z), characterised by the detector symmetry and the solenoid magnetic field orientation. Positions of detector elements are in general given in this frame and will be called global positions. Directions given in the global frame such as a momentum expressed in respect to the global frame will be called global momentum for consistency. The global frame is slightly shifted to the so-called physics frame (x_p, y_p, z_p), which is characterised by the fact that the z_p-axis points along the beam direction and which will be the frame to be used for physics analysis.

The intrinsic frame on detector elements or surfaces will be called local frame. Positions given with respect to the local frames will be called local positions. In general local positions are two dimensional as the third free parameter is determined by the constraint of the position to be on the surface.

2 Implementation

This chapter will explain the specific implementation of the ATLAS Tracking Event Data Model, starting with the track representation. A minimal version of a track can be created with one set of track parameters. A more useful track could contain the measurement from which the parameters were derived, the information about which algorithm created the track (its 'author') and quality information of the fit. Moreover, as a track passes through the ATLAS detector, it crosses many 'surfaces' (such as detector elements, dead material etc.), and the quantities which make up a track are usually defined on a surface: that is, measurements are taken on a detector element, and track parameters are always expressed with respect to one. It is therefore useful to group information defined on the same surface, by that surface. For example, on an inner detector planar surface (representing a silicon wafer) we could have information from a measurement, the track parameters at that point, information about the scattering caused by material interactions and the fit qualities for that particular surface.

The structure of the ATLAS track object (henceforth `Track`) reflects exactly the desired grouping of available information by the surfaces it is defined on. In practice `Track` contains a collection of `TrackStateOnSurface` objects, which in turn can contain pointers to objects representing:

- local coordinates and momentum of the track (i.e. the track parameters) — see Section 2.2;
- the measurement found on that surface — see Section 2.4;
- an outlier measurement — see Section 2.5;
- an interaction with material on that surface, (i.e. the change of direction caused by scattering or bremsstrahlung plus errors on these angles) — see Section 2.6;
- the fit quality of the measurement — see Section 2.7;

In addition to defining the structure of the track information, the surfaces provide the geometric information needed during tracking and thus form an integral part of the Tracking EDM. Their implementation is described in Section 3. The `Track` and `TrackStateOn-Surface` classes are located in the package TrkTrack [3].

The mathematical objects, like position, momentum and corresponding error matrices, are described by data objects which are derived from CLHEP classes [4] and which are used throughout the whole tracking and sub-detector EDMs. Therefore these data objects are grouped in a common package, TrkEventPrimitives [5], and are described in further detail in Section 4.

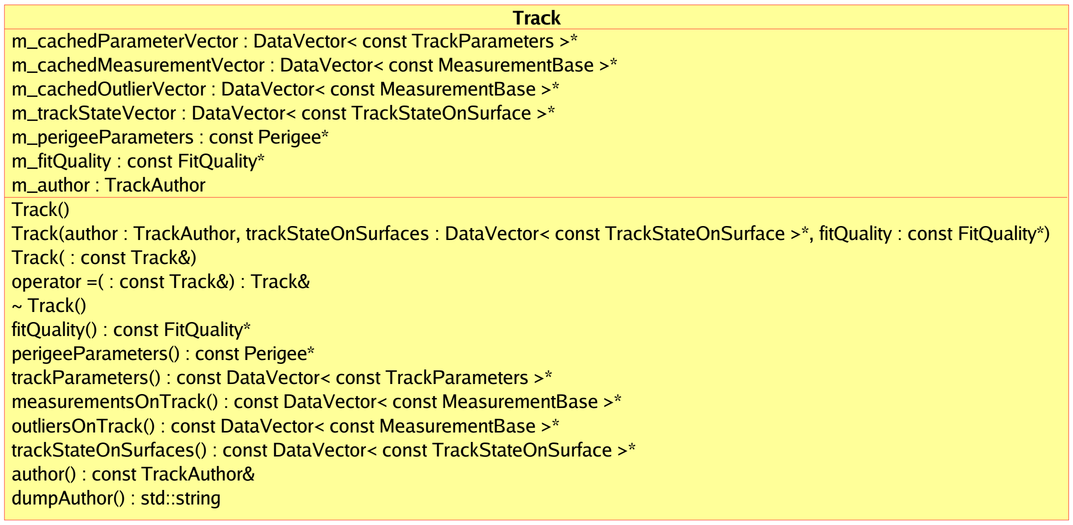
2.1 Organisation of the TrackStateOnSurface Content

It is not necessary to have all these objects in each `TrackStateOnSurface`. For example, it is possible to have a prediction for a track, without a measurement (i.e. the `TrackStateOnSurface` has track parameters, but nothing else). Also the pattern recognition would typically produce only measurements as output, since precisely fitted track parameters and fit quality information are not yet available at that stage. This flexibility

bit #	type	TP object	MB object	SA object	FQ object
0	Measurement	O	**M**	–	O
1	InertMaterial	?	?	?	?
2	BremPoint	**M**	O	**M**	O
3	Scatterer	**M**	O	**M**	O
4	Perigee	**M**	–	–	–
5	Outlier	O	**M**	–	O
6	Hole	O	–	–	–
7	Unknown	O	O	O	O

Table 1: *Binary encryption of the type flag in the `TrackStateOnSurface` class. Depending on the type, it may (O), should (**M**) or must not (–) contain a measurement (MB), track parameter (TP), scattering angle (SA) or fit quality (FQ) object. InertMaterial is reserved for future versions of the Tracking EDM while Unknown can be chosen if a type does not fall into any of the above categories.*

allows ATLAS to handle different kinds of trajectory information with the same class. Not all omissions make sense though: a scatterer, for example, should always have a matching set of track parameters. For fast distinction between different kinds of trajectory information, a set of type flags is stored with each `TrackStateOnSurface`. The flags are encoded as a bit-set explained in Table 1.



```
Track
m_cachedParameterVector : DataVector< const TrackParameters >*
m_cachedMeasurementVector : DataVector< const MeasurementBase >*
m_cachedOutlierVector : DataVector< const MeasurementBase >*
m_trackStateVector : DataVector< const TrackStateOnSurface >*
m_perigeeParameters : const Perigee*
m_fitQuality : const FitQuality*
m_author : TrackAuthor
Track()
Track(author : TrackAuthor, trackStateOnSurfaces : DataVector< const TrackStateOnSurface >*, fitQuality : const FitQuality*)
Track( : const Track&)
operator =( : const Track&) : Track&
~ Track()
fitQuality() : const FitQuality*
perigeeParameters() : const Perigee*
trackParameters() : const DataVector< const TrackParameters >*
measurementsOnTrack() : const DataVector< const MeasurementBase >*
outliersOnTrack() : const DataVector< const MeasurementBase >*
trackStateOnSurfaces() : const DataVector< const TrackStateOnSurface >*
author() : const TrackAuthor&
dumpAuthor() : std::string
```

Figure 2: *UML diagram showing the members and methods of the `Track` class. Trajectory information is entered and stored in the class as a vector of `TrackStateOnSurface`, but can be retrieved also as vectors of solely the measurements, outliers and parameters. Those vectors are created at the time when the partial information is asked for and kept as "cached" vectors until the `Track` object is deleted.*

Figure 2 shows the structure of the `Track` object. The information about the trajectory is passed through as a vector of `TrackStateOnSurface` objects which will be owned by the so-created track. Easy access to only the measurements or reconstructed parameters is provided by separate return methods. An important point is that, for performance reasons, `TrackStateOnSurface` does not check that the information stored is self–consistent (in fact, it does no checking at all). For example, it would be possible to create a `TrackStateOnSurface` which contains track parameters and measurements which are actually from different surfaces. It is also possible to create several `TrackStateOnSurfaces` representing the same surface (for example, one containing the parameter, one containing a scatterer and so on). This must not be done: algorithms using `Track` should not have to loop over the entire `TrackStateOnSurface` collection to be sure they have retrieved all information relevant to one particular surface.

The list of `TrackStateOnSurfaces` does not require strictly to be sorted along the track's path. The track class itself and algorithms using it will, however, be optimised to work with a `Track` whose states are ordered along the particle's direction of flight. For particles from collisions inside ATLAS this means that the Perigee parameters should be given first and the measurements at the outermost detector layers last. As an example, the `Track` class will always look for the Perigee parameters as the first `TrackStateOnSurface` and a track re-fit will try to sort the vector of track states in case they are not yet ordered.

2.1.1 Extended representations of trajectory – the MultiComponentState class

The trajectory of a track can in most cases be adequately described by a single `TrackParameters` object for a given surface, see Section 2.2. This representation is used by track fitters to provide the optimal description of the trajectory in the case where all measurement errors and process noise are Gaussian distributed. In the case of non-Gaussian effects, a more general usage of `TrackParameters` objects must be employed. A technique commonly used by non-linear track fitters to approximate non-Gaussian distributions is to use a weighted Gaussian-sum [6]. The `MultiComponentState` class is used to describe such Gaussian mixtures. It is a list of `TrackParameters`objects, each with a corresponding weight.

Since it is necessary to describe the trajectory of a track on surfaces within the detector, an extension to the `TrackStateOnSurface` class is also required. The class `MultiComponentStateOnSurface` is derived from `TrackStateOnSurface`. In addition to the information held by the base class, it contains a pointer to the `MultiComponentState` object describing the track trajectory at the surface. This inheritance structure is transparently incorporated into the common track class. The according classes for a multi component description can be found in the TrkMultiComponentStateOnSurface package [7].

Gaussian-mixture models facilitate a more precise treatment of: tails in multiple-scattering models; radiative energy loss processes such as bremsstrahlung; and outlier tails in measurement-error distributions. Non-linear track fitters such as the Gaussian-sum filter (a generalisation of the Kalman filter) use Gaussian-mixture approximations extensively.

2.2 Track Parameters

As mentioned above, each `TrackStateOnSurface` can contain an object representing the track through a set of parameters such as space coordinates and the momentum at that point. The objects are called track parameters and are located in the TrkParameters package [8]. A key requirement of the `TrackParameters` objects is that they must provide all the necessary functionality, whilst being lightweight and fast. The features of their design and the different representations are described in the following.

Global Representation:
Global or free track parameters can be assigned to a track in any point in the detector. They are characterised by a set of variables,

$$G_i = (\vec{x}, \vec{p}, q) \quad , \tag{1}$$

where $\vec{x}$ denotes the global position, $\vec{p}$ the global momentum and q the charge of the track. This representation is independent from the underlying track model such as helicoidal or straight line track. As the expression of global parameters is common for all derived types of parameter expressions they are bundled in the base class `TrackParameters`. However, for consistency with track finding and fitting which operate in local coordinate frames, it has been chosen that all track parameters require in addition a local representation. For this reason the `TrackParameters` class is realised as an abstract base class that can not be instantiated during the program flow.

Local Representation:
The local track representation is given by a set of five parameters

$$L_i = (l_1, l_2, \phi, \theta, q/p) \tag{2}$$

where (l_1, l_2) denote the two coordinates in the intrinsic frame of the surface and $(\phi, \theta, q/p)$ a representation of the momentum in the global frame[1]. Evidently the meaning of the local parameters l_1, l_2 vary by the different surface types. To guarantee type safety, individual classes extending the `TrackParameters` base class has been implemented for every surface type. Most flavours of local track parameter representations were predefined by the different surface types that can be found inside the ATLAS detector, like wire lines and sensor planes. The collection of surface types provided by the ATLAS Tracking EDM (see Section 3) exceeds the actual surfaces associated to detector or readout elements, and includes concepts like virtual tracking surfaces or trigger surfaces (e.g. cylindrical barrel layers, end-cap discs). Specific classes for local representations of `TrackParameters` exist for every surface type in the Tracking EDM:

- `AtaCylinder` : contains the parameters on a cylinder, defined as $(locR\phi, locZ, \phi, \theta, q/p)$;
- `AtaDisc` : contains the parameters on a disc, defined as $(locR, locPhi, \phi, \theta, q/p)$;
- `AtaPlane` : contains the parameters on an arbitrarily oriented plane, defined as $(locX, locY, \phi, \theta, q/p)$;
- `AtaStraightLine` : contains the parameters for a representation at a straight line, defined as $(locR, locZ, \phi, \theta, q/p)$. The radius $locR$ is signed in this scope, a detailed description of the calculation of this sign can be found in the Appendix 8.2;
- `Perigee` : contains the parameters for a representation at a Perigee, defined as $(d_0, z_0, \phi, \theta, q/p)$. The transverse impact parameter d_0 is signed, the calculation of the sign is consistent with the calculation of the sign of the radius for the `AtaStraightLine` representation.

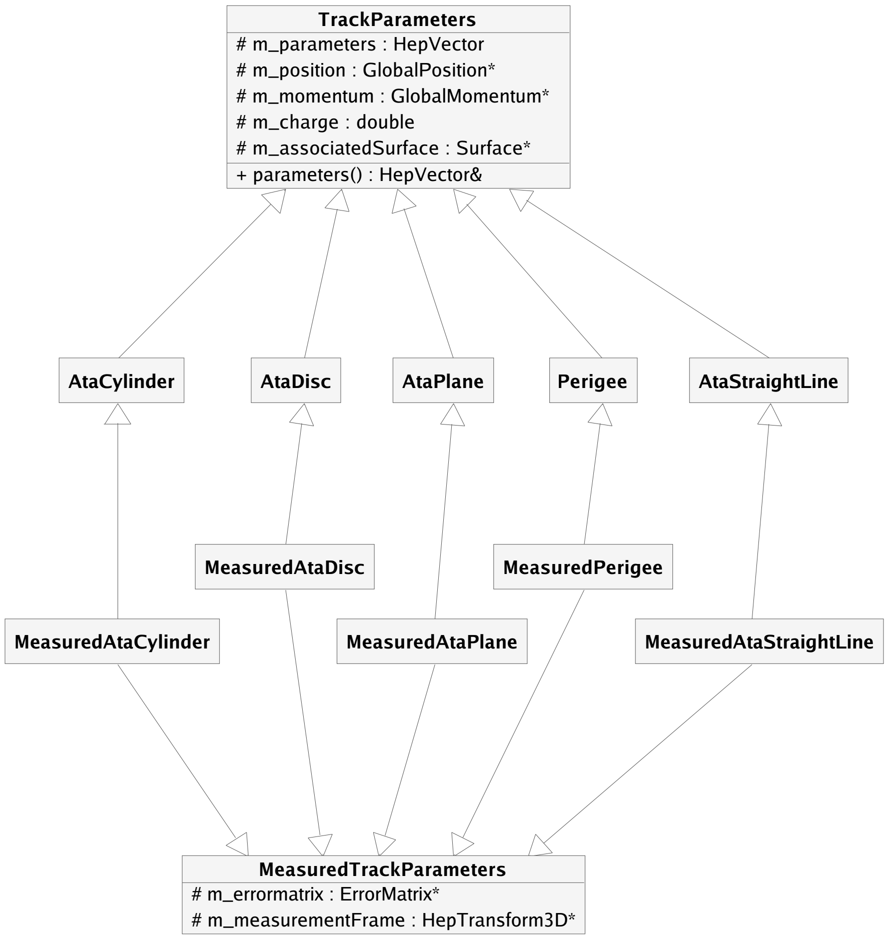

TrackParameters
# m_parameters : HepVector
# m_position : GlobalPosition*
# m_momentum : GlobalMomentum*
# m_charge : double
# m_associatedSurface : Surface*
+ parameters() : HepVector&
AtaCylinder
AtaDisc
AtaPlane
Perigee
AtaStraightLine
MeasuredAtaDisc
MeasuredPerigee
MeasuredAtaCylinder
MeasuredAtaPlane
MeasuredAtaStraightLine
MeasuredTrackParameters
# m_errormatrix : ErrorMatrix*
# m_measurementFrame : HepTransform3D*

Figure 3: *UML diagram showing the inheritance structure of* `TrackParameters` *data classes.* `TrackParameters` *exist in both, an unmeasured and a measured flavour. The measured track parameter classes follow a double inheritance structure, inheriting from the unmeasured class they represent and a common base class for measured track parameters, holding the error matrix description and the measurement frame definition.*

The choice of taking $(\phi, \theta, q/p)$ for the momentum representation and not $(\phi, \cot\theta, q/p_T)$, which has frequently been used for helix parameterisations in ATLAS, was motivated by the following requirements:

- The design of the local parameterisation should not be restricted to a specific detector part either, but represent track states at any point in the detector. Since in a non-homogeneous magnetic field the transverse momentum in general is not a constant of motion, the choice of q/p is to be more appropriate to fulfil this requirement.

- Using $(\phi, \theta, q/p)$ results in a better computing performance for material corrections in the tracking algorithms. Energy loss corrections have to be applied to the current value of the momentum. Introducing multiple scattering by correcting the angular direction and its uncertainty favours the angle-based parameterisation (ϕ, θ) for the momentum direction and also requires the knowledge of the momentum magnitude. Frequent transformations to and from the helical representation should therefore be avoided.

Error Representation:
Each of these classes exists in a second version describing measured track parameters, i.e. the parameters and their uncertainties. The measured track parameters follow a double inheritance structure, inheriting from one of the above classes and from the `MeasuredTrackParameters` base class, which comes with an `ErrorMatrix` and a measurement frame. The entire inheritance structure is displayed in Fig. 3. The additional concept of a measurement frame is needed for parameters at a Perigee, at a line or at a cylinder. As Fig. 4 illustrates, it defines the frame in which the error on the radius and the local z coordinate (resp. the error on d_0 and z_0 for Perigee representations) are given. For a cylinder surface the measurement frame is by convention the tangential Cartesian frame for performance reasons. For a plane surface the measurement frame is identical to the three dimensional helper frame of the surface.

Pseudo constructors:
The `TrackParameters` base class has pseudo-constructors which return a pointer to a new object of the concrete type. They make it possible to copy the object or update the parameter and error values while retaining the type of local representation and the presence of measurement errors. The pseudo-constructors are realised through clone methods as listed in Table 2.

2.3 Measurements Entering the Track Reconstruction

Every sub-detector in ATLAS processes its raw data output to form a local position measurement with calibrated error, which is associated to an alignable measurement surface from the ATLAS detector geometry. The types of measurements present in ATLAS are either clusters from the silicon modules and muon chambers or drift circles from the drift

[1]Following the ATLAS system of units convention, the parameter ranges for the azimuthal respectively polar angles are given as $\phi \in [-\pi, \pi), \theta \in [0, \pi]$. When being compiled in debug mode, the parameters ϕ and θ are actually checked for range consistency and throw a warning message when the range is exceeded.

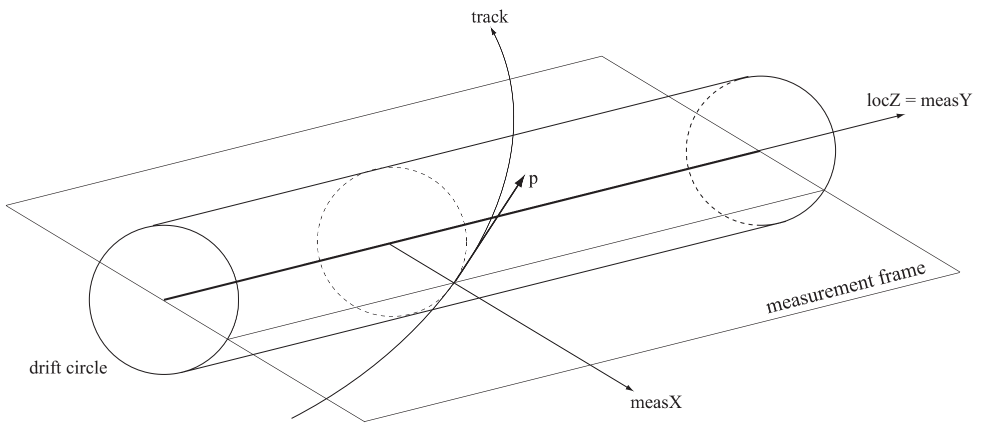


Figure 4: *Illustration of the measurement frame for measured track parameter representation at a Perigee or at a straight line. The plane of the measurement frame is defined by the cylinder/wire's local z direction and the vector of closest approach of the track to the cylinder axis or wire.*

`clone()`	copies the object
`cloneWithoutError()`	turns the copy into an unmeasured flavour
`cloneWithError(ErrorMatrix*)`	turns the copy into a measured flavour, or updates existing error
`cloneToNew(HepVector&, ErrorMatrix*)`	updates parameter values and errors in the copy while keeping the actual child flavour

Table 2: *List of pseudo-constructors for the* `TrackParameters` *class*

tubes in the inner detector and muon spectrometer. They serve as input to the pattern recognition algorithms, which usually start with a very detector-specific local search. Nevertheless the clusters and drift circles from all ATLAS tracking detectors are designed to inherit from a common base class: `PrepRawData` [9]. As its class diagram in Fig. 5 shows, it implements the members and functions which are common to all kinds of tracking measurements, but it is also needed to create the common interfaces of the downstream measurement classes and of the common tracking tools.

2.4 Measurement Objects on the Track

The measurement objects stored on a `Track` are *not* the `PrepRawData`, but specific objects used inside the track reconstruction, which allow the output of the pattern recognition to be different from the input for several reasons.

- In all cases **additional calibration corrections** performed at hit association time are allowed for in the measurement objects (see Section 2.4.1).
- A general **flexibility** is needed because the input to the pattern recognition can be either clusters and drift circles or space points (usually a combination of two one-dimensional measurements into a two-dimensional one by means of a stereo angle between adjacent sensor planes). The output of the pattern recognition can be those

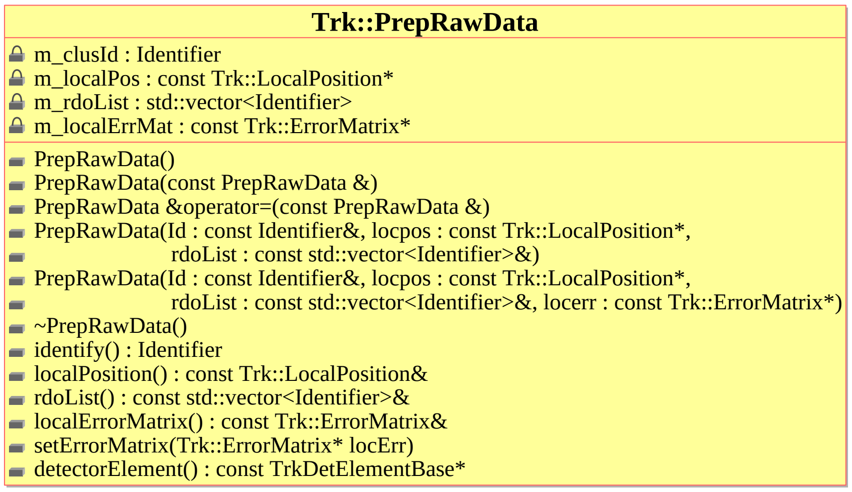

Trk::PrepRawData
m_clusId : Identifier
m_localPos : const Trk::LocalPosition*
m_rdoList : std::vector<Identifier>
m_localErrMat : const Trk::ErrorMatrix*
PrepRawData()
PrepRawData(const PrepRawData &)
PrepRawData &operator=(const PrepRawData &)
PrepRawData(Id : const Identifier&, locpos : const Trk::LocalPosition*, rdoList : const std::vector<Identifier>&)
PrepRawData(Id : const Identifier&, locpos : const Trk::LocalPosition*, rdoList : const std::vector<Identifier>&, locerr : const Trk::ErrorMatrix*)
~PrepRawData()
identify() : Identifier
localPosition() : const Trk::LocalPosition&
rdoList() : const std::vector<Identifier>&
localErrorMatrix() : const Trk::ErrorMatrix&
setErrorMatrix(Trk::ErrorMatrix* locErr)
detectorElement() : const TrkDetElementBase*

Figure 5: *UML diagram for `PrepRawData`, the base class for clusters and drift circles.*

hits associated to a track, but also a whole segment of several hits within a detector region or a group of "competing" measurements which are associated to the track in a fuzzy way.

- Once created by the data preparation, the clusters and drift circles are protected by design against any further modification. New objects which hold the calibrated or combined information are therefore a necessity. This guarantees that subsequent stages of the pattern recognition are not biased by each other and allows, for example, calibration and alignment tasks to be run on top of the full reconstruction.

Ideally, the `TrackStateOnSurface` class should be able to store all of the above measurement concepts, and clients such as track fitters should be able to deal seamlessly with the different concepts. This is achieved by making the different measurement objects inherit from a common base class, `MeasurementBase`, provided by a package with the same name [10]. This class holds the minimum information necessary to make it fittable to a track:

- `LocalParameters`, the measurement in local coordinates of dimension 1 to 5 depending on the specific implementation;
- `ErrorMatrix`, the local error representation;
- `GlobalPosition`, the measurement in global coordinates;
- an associated `Surface` to guarantee a consistent local/global transformation

The information provided by the base class is sufficient to allow track extrapolation, fitting, display and object handling without identifying the concrete type of measurement. The current list of concrete measurement types derived from `MeasurementBase` is

- `RIO_OnTrack` : the clusters and drift circles after applying additional calibration corrections — see Section 2.4.1;
- `CompetingRIOsOnTrack` : a group of `RIO_OnTrack`s on the same detector plane. They represent measurements and unresolved hit ambiguities compatible with the estimated track position — see Section 2.4.2;

- `Segment` : a set of fitted `RIO_OnTracks` — see Section 2.4.3;
- `SpacePoint` : the expression of a detector hit (combination) in global coordinates. They are used to start the search for track patterns — see Section 2.4.4.

Figure 6 shows the inheritance structure of the `MeasurementBase` class and its child classes `RIO_OnTrack`, `CompetingRIOsOnTrack` and `Segment`. These child classes are extended in turn inside the sub-detector repositories by relevant information which is not accessible in the general Tracking realm. This structure allows to integrate new concepts of measurement representation and thus extend the Tracking EDM easily in the future.

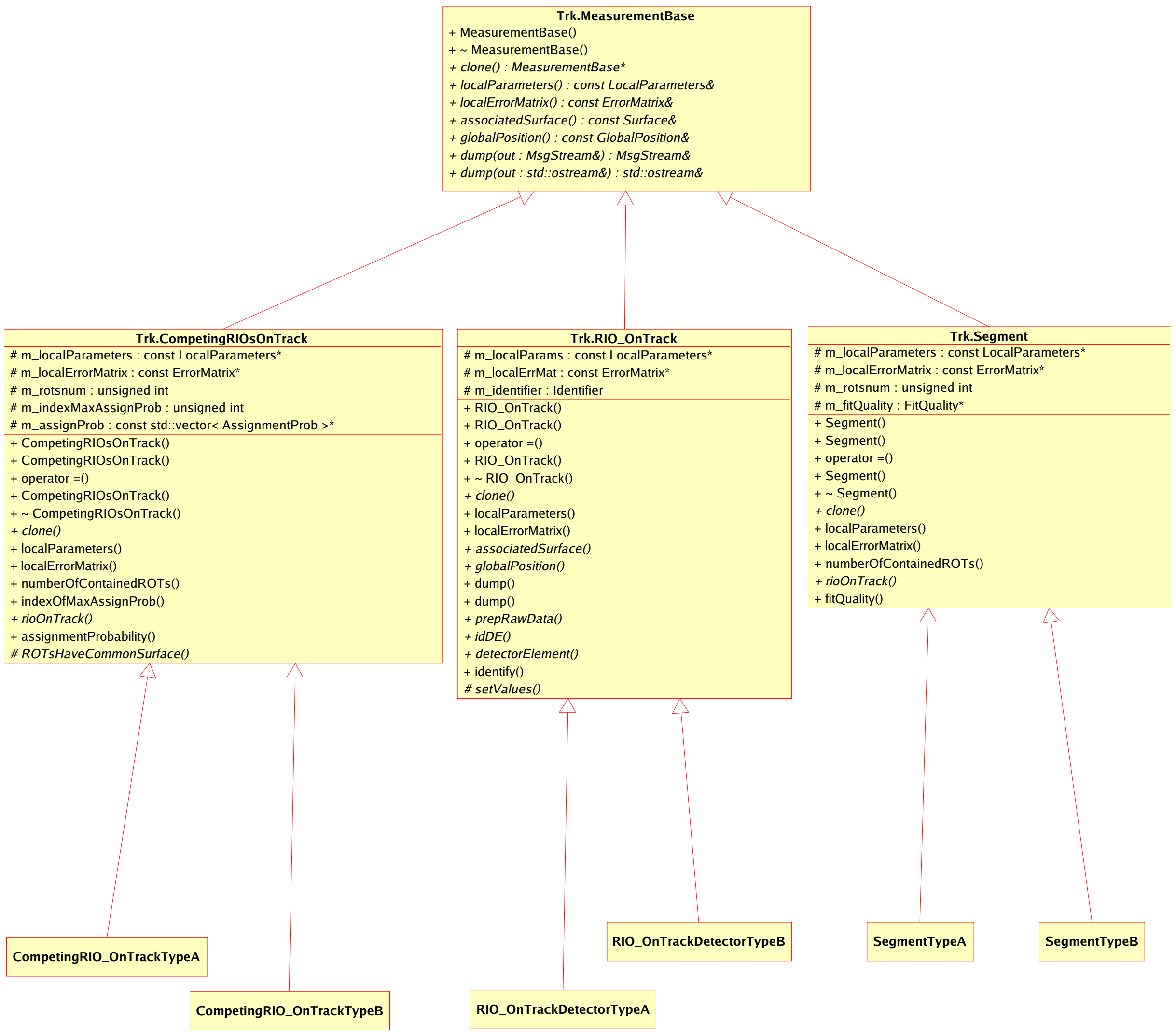

Trk.MeasurementBase
+ MeasurementBase()
+ ~ MeasurementBase()
+ clone() : MeasurementBase*
+ localParameters() : const LocalParameters&
+ localErrorMatrix() : const ErrorMatrix&
+ associatedSurface() : const Surface&
+ globalPosition() : const GlobalPosition&
+ dump(out : MsgStream&) : MsgStream&
+ dump(out : std::ostream&) : std::ostream&
Trk.CompetingRIOsOnTrack
# m_localParameters : const LocalParameters*
# m_localErrorMatrix : const ErrorMatrix*
# m_rotsnum : unsigned int
# m_indexMaxAssignProb : unsigned int
# m_assignProb : const std::vector< AssignmentProb >*
+ CompetingRIOsOnTrack()
+ CompetingRIOsOnTrack()
+ operator =()
+ CompetingRIOsOnTrack()
+ ~ CompetingRIOsOnTrack()
+ clone()
+ localParameters()
+ localErrorMatrix()
+ numberOfContainedROTs()
+ indexOfMaxAssignProb()
+ rioOnTrack()
+ assignmentProbability()
# ROTsHaveCommonSurface()
Trk.RIO_OnTrack
# m_localParams : const LocalParameters*
# m_localErrMat : const ErrorMatrix*
# m_identifier : Identifier
+ RIO_OnTrack()
+ RIO_OnTrack()
+ operator =()
+ RIO_OnTrack()
+ ~ RIO_OnTrack()
+ clone()
+ localParameters()
+ localErrorMatrix()
+ associatedSurface()
+ globalPosition()
+ dump()
+ dump()
+ prepRawData()
+ idDE()
+ detectorElement()
+ identify()
# setValues()
Trk.Segment
# m_localParameters : const LocalParameters*
# m_localErrorMatrix : const ErrorMatrix*
# m_rotsnum : unsigned int
# m_fitQuality : FitQuality*
+ Segment()
+ Segment()
+ operator =()
+ Segment()
+ ~ Segment()
+ clone()
+ localParameters()
+ localErrorMatrix()
+ numberOfContainedROTs()
+ rioOnTrack()
+ fitQuality()
CompetingRIO_OnTrackTypeA
CompetingRIO_OnTrackTypeB
RIO_OnTrackDetectorTypeB
RIO_OnTrackDetectorTypeA
SegmentTypeA
SegmentTypeB

Figure 6: *UML diagram showing the inheritance structure of all fittable measurements in the new Tracking EDM. These classes are purely abstract while the concrete classes (indicated here by "`-TypeA`", "`-TypeB`") extend the structure further into the sub-detectors. The `SpacePoint` class also extends the common measurement base class, but is omitted in this diagram.*

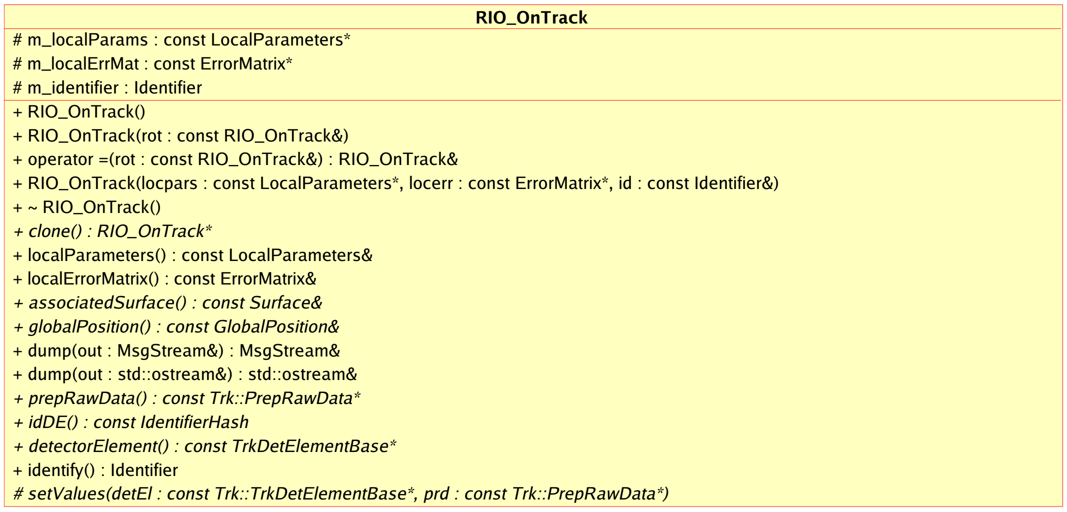
The EDM base classes also have a pseudo-constructor, realised through a `clone()` method, which is overloaded in the child classes, and which returns a pointer to a new object of the concrete type. For example this allows you to copy a vector of

MeasurementBases and still retain their measurement types as well as their concrete detector-specific implementations.

2.4.1 The RIO_OnTrack objects

Inside Tracking an abstract base class is provided, RIO_OnTrack, to represent clusters and drift circles (the PrepRawData objects) after additional calibration corrections have been applied. This class is then extended by the child classes (i.e. concrete classes that provide the InnerDetector-specific and MuonSpectrometer-specific data). The idea behind this concept is to allow common algorithms for pattern recognition and track fitting which work on base class objects (i.e. do not depend on the detector-specific implementation of the measurement corrections), whilst the Inner Detector and Muon Spectrometer maintain full control over the corrections and calibrations via the child classes. The base class RIO_OnTrack is contained in the package TrkRIO_OnTrack [11].

A good example for the additional calibration corrections would be a drift circle from the MDT, where the position has been modified to allow for the sag of the wire[2]. This sagging correction can only be calculated at hit association time, when the exact position along the sagging wire is obtained from the track prediction.



```
RIO_OnTrack
# m_localParams : const LocalParameters*
# m_localErrMat : const ErrorMatrix*
# m_identifier : Identifier
+ RIO_OnTrack()
+ RIO_OnTrack(rot : const RIO_OnTrack&)
+ operator =(rot : const RIO_OnTrack&) : RIO_OnTrack&
+ RIO_OnTrack(locpars : const LocalParameters*, locerr : const ErrorMatrix*, id : const Identifier&)
+ ~ RIO_OnTrack()
+ clone() : RIO_OnTrack*
+ localParameters() : const LocalParameters&
+ localErrorMatrix() : const ErrorMatrix&
+ associatedSurface() : const Surface&
+ globalPosition() : const GlobalPosition&
+ dump(out : MsgStream&) : MsgStream&
+ dump(out : std::ostream&) : std::ostream&
+ prepRawData() : const Trk::PrepRawData*
+ idDE() : const IdentifierHash
+ detectorElement() : const TrkDetElementBase*
+ identify() : Identifier
# setValues(detEl : const Trk::TrkDetElementBase*, prd : const Trk::PrepRawData*)
```

Figure 7: *UML diagram showing the interface of* RIO_OnTrack.

Such additional and often detector-specific information about the hit is completed at the level of tracking, for two reasons. One is computing performance: a time-consuming calibration would only be done for hits entering a precise refit of the final set of tracks to be kept for analysis. The other is that only from the track prediction a missing coordinate or incident angle is supplied, or a left-right ambiguity can be solved. This additional information, however, would not contribute to the track fitting result. As shown in Fig. 7, the RIO_OnTrack class provides additional information to what is already implemented by MeasurementBase:

- a pointer to the common object describing clusters and drift circles: PrepRawData;
- the link (i.e. pointer) to the relevant detector element;

[2]For further details of the particular example, see the relevant Muon Spectrometer documentation.

- the `Identifier`, and `HashIdentifier` which e.g. allow to re-create missing information on persistified data.

Neither the pointer to the prepared raw data, nor the pointer to the detector element are persistified (see the appendices for a more detailed discussion of persistency). It is for this reason that the `Identifier` is needed: once persistified tracks are read in, the `Identifier` allows the pointer to the detector element to be re-created (and in principle would allow the link to the `PrepRawData` to be re-created as well). Of course, it would be possible simply to use `Identifier`s and scrap the pointers altogether, however, this would be unacceptably slow because it would require a search operation every time the associated detector element is accessed during program flow.

2.4.2 Competing RIO_OnTrack objects

This class is used to represent a group of mutually exclusive hits which are nevertheless all included on a track using a fuzzy assignment model based on assignment probabilities. The main algorithm working with these data objects is the Deterministic Annealing Filter [6], a combined local pattern recognition and track fitting algorithm which is implemented in the ATLAS reconstruction software. A `CompetingRIOsOnTrack` contains one or more `RIO_OnTrack`s of the same concrete type in one "effective detector layer" (i.e. possibly from more than one detector surface) which compete against each other in being assigned to a track. The way this "effective detector layer" is defined depends strongly on the sub-detector type and even different definitions may exist for special reconstruction purposes. The single local parameters and local error representations enforced by the base class describe the mean values $\widetilde{\mathbf{m}}$ and $\widetilde{\mathbf{V}}$ according to the weights or assignment probabilities p associated with each `RIO_OnTrack`.

$$\widetilde{\mathbf{V}}^{-1} = \sum_i p_i \mathbf{V}_i^{-1} \tag{3}$$

$$\widetilde{\mathbf{m}} = \widetilde{\mathbf{V}} \left(\sum_i p_i \mathbf{V}_i^{-1} \mathbf{m}_i \right) \tag{4}$$

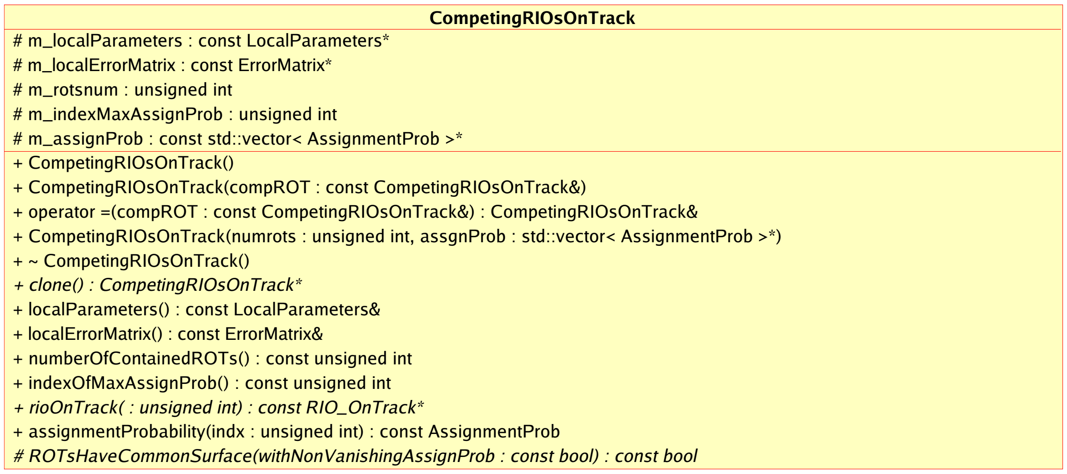
Here m_i and V_i are the coordinate vector and covariance matrix of the ith measurement of the set of competing measurements.

The `CompetingRIOsOnTrack` object is contained in a package of it own [12] and is the abstract base class for derived classes in each detector sub-system. As Fig. 8 details, the base class contains only the vector of assignment probabilities and some basic information to handle a given number of hits while the associated `Surface` and the vector of `RIO_OnTrack` objects are implemented in the concrete classes. This separation allows the derived classes to give back also the specific type of `Surface` and `RIO_OnTrack`, avoiding extensive type identification at run-time (RTTI).

Any information about the contained hits such as their `PrepRawData` or `Identifier` has to be retrieved from the `RIO_OnTrack`s itself.

2.4.3 Track segments

A track segment contains a vector of `RIO_OnTrack` objects and represents the fitted information of these objects using an implicit track model. Track `Segment` objects are also integrated in the `MeasurementBase` scheme since they provide a measurement of several



```
CompetingRIOsOnTrack
# m_localParameters : const LocalParameters*
# m_localErrorMatrix : const ErrorMatrix*
# m_rotsnum : unsigned int
# m_indexMaxAssignProb : unsigned int
# m_assignProb : const std::vector< AssignmentProb >*
+ CompetingRIOsOnTrack()
+ CompetingRIOsOnTrack(compROT : const CompetingRIOsOnTrack&)
+ operator =(compROT : const CompetingRIOsOnTrack&) : CompetingRIOsOnTrack&
+ CompetingRIOsOnTrack(numrots : unsigned int, assgnProb : std::vector< AssignmentProb >*)
+ ~ CompetingRIOsOnTrack()
+ clone() : CompetingRIOsOnTrack*
+ localParameters() : const LocalParameters&
+ localErrorMatrix() : const ErrorMatrix&
+ numberOfContainedROTs() : const unsigned int
+ indexOfMaxAssignProb() : const unsigned int
+ rioOnTrack( : unsigned int) : const RIO_OnTrack*
+ assignmentProbability(indx : unsigned int) : const AssignmentProb
# ROTsHaveCommonSurface(withNonVanishingAssignProb : const bool) : const bool
```

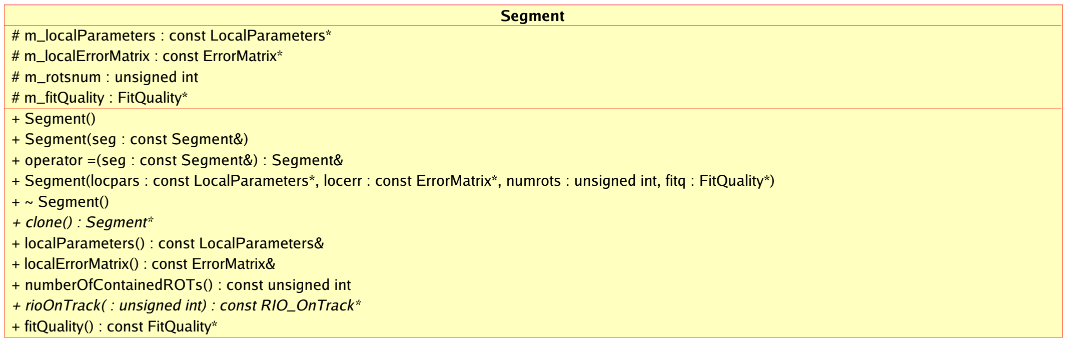
Figure 8: *UML diagram showing the interface of `CompetingRIOsOnTrack` objects.*

local track coordinates and act similar to `RIO_OnTrack` objects in various fitting techniques. Contrary to the `RIO_OnTrack` class representing a single hit, the track segments need access to the initial calibrated measurements and their underlying prepared raw data, so that they can be resolved at any stage. This access is provided by additional methods returning

- the number n of contained `RIO_OnTrack`s on the `Segment`;
- a pointer to `RIO_OnTrack`s describing the i-th hit on the `Segment`, where $i = 0, ..., n-1$. Like with the `CompetingRIOsOnTrack` class, this method is implemented in the classes derived from `Segment` to avoid unnecessary RTTI;
- a pointer to a `FitQuality` object which describes the the quality of a possible local fit. If this information is not filled (e.g. a segment has been constructed using simpler methods) it returns a null pointer.

Detail about the data members and methods are listed in Fig. 9.

The Tracking EDM also provides a general `TrackSegment` as a concrete implementation of the above base class. This `TrackSegment` can be used to describe segments from any part of the detector and can hold `RIO_OnTrack` objects of different type. Both segment classes are provided by the package TrkSegment [13].



```
Segment
# m_localParameters : const LocalParameters*
# m_localErrorMatrix : const ErrorMatrix*
# m_rotsnum : unsigned int
# m_fitQuality : FitQuality*
+ Segment()
+ Segment(seg : const Segment&)
+ operator =(seg : const Segment&) : Segment&
+ Segment(locpars : const LocalParameters*, locerr : const ErrorMatrix*, numrots : unsigned int, fitq : FitQuality*)
+ ~ Segment()
+ clone() : Segment*
+ localParameters() : const LocalParameters&
+ localErrorMatrix() : const ErrorMatrix&
+ numberOfContainedROTs() : const unsigned int
+ rioOnTrack( : unsigned int) : const RIO_OnTrack*
+ fitQuality() : const FitQuality*
```

Figure 9: *UML diagram showing the interface of track `Segment`s.*

2.4.4 Space Points

A space point contains three-dimensional spatial information of a measurement and its errors. Space points are used in the pattern recognition stage of the track finding. The usage of space points in the track fitting within the trigger is being studied.

Inside Tracking under the package `TrkSpacePoint` [14], the abstract base class `SpacePoint` is defined, which is derived from `MeasurementBase` to allow for the integration of the `SpacePoint` objects into the `TrackStateOnSurface` class. By now two concrete classes are derived from the `SpacePoint` base class, the `PixelSpacePoint` and the `SCT_SpacePoint` class, both representing measurements of Inner Detector subsystems.

Space points can be instantiated using local coordinates, as e.g. in the case of `PixelSpacePoints`. The global position has to be set up in the derived class' constructor using the `localToGlobal` method of the associated surface. It is also possible to instantiate a space point using a global position. This is required when a space point is a combination of more than one measurement, as e.g. in the case of the SCT detector, where the information of the two strips of an SCT module is combined by requiring that the assumed hit point on the second module is in elongation of the connection between the primary vertex and the assumed hit point on the first module. In this case the local position on a specified surface has to be set up in the derived class' constructor using the `globalToLocal` method of this surface.

The method to initialise the local error matrix of a space point has to be provided by the derived classes. The local error matrix of space points instantiated with a global position has to be initialised with care, to avoid unwanted dependencies of other packages. In the case of the `SCT_SpacePoint`, the local errors are initialised with a constant instead of a full error propagation. However, this estimation is sufficient to fulfill the requirements of present `SpacePoint` clients. Global errors are then calculated from the local errors by a method provided by the `SpacePoint` base-class For performance reasons all error matrices are only set up or calculated on demand and cached for further reference.

2.4.5 Navigation back to the Raw Data

For a variety of tasks it is important to analyse a given data object at different stages of the track reconstruction, i.e. to navigate back to the original measurement. Such tasks are, for example, the treatment of hits shared between tracks, hardware monitoring, detector performance studies as well as truth matching. The structure of tracking base classes makes backward navigation possible within the tracking realm and without switching to detector-specific code: the general `Track` class can be asked for a list of `MeasurementBase` objects which can be turned into a `RIO_OnTrack`, possibly via `Segments` or `CompetingRIO_OnTracks`. Either from their Identifiers or from their `PrepRawData` links the original hits can be accessed.

2.5 Outlier Measurements

It is often needed to retain information about measurements which have been selected as not compatible with the track hypothesis. Such outliers can appear, for example, on misaligned detector layers, after a point of significant energy loss (mostly electrons) or on drift tube measurements where a second track has interfered. They still need to be kept for evaluating the track's quality and possible recuperation later, but need to be flagged against inclusion into further fit iterations or into calibration procedures. A `TrackStateOnSurface` represents an outlier only through setting the associated type flag, its content is otherwise identical to a measurement.

2.6 Scatterers and Brem Points

When a track passes through a dense region, e.g. sensor material or a detector structure not instrumented at all to measure hits, there is obviously the possibility that its trajectory will change[3]. For further analysis it can be useful to store on the track the information about an identified point on the trajectory where multiple scattering or bremsstrahlung occurred. To this end, `TrackStateOnSurface` supports the types "Scatterer" and "Brem-Point" and objects called `ScatteringAngleOnSurface` which store the change in direction, and the error on this change.

2.7 Fit Quality

The fit quality object (`FitQualityOnSurface`) simply contains the χ^2 and the number of degrees of freedom. As with the `Track`'s overall `FitQuality`, it is envisioned that this class could be inherited from, if a more complex representation of the fit quality is needed. Both classes are a part of TrkEventPrimitives.

2.8 Holes on the Track

A hole on the track describes a missing measurement *between* the first and last actual measurements. For example, if your track starts in the SCT, you should not consider a missing pixel b-layer hit as a hole. The missing measurement is usually expected at an active, efficient and non-masked sensor plane or drift tube layer where the pattern recognition has been unable to find any hit within a given search radius. A high number of holes on a track is usually a strong criterion for a bad track candidate and may lead the pattern recognition to rejecting the candidate. The information about the number of remaining holes may still be used in later reconstruction stages to improve the overall event quality. The `TrackStateOnSurface` must be flagged as hole and, if in the current version of the Tracking EDM the detector surface with missing measurements needs to be specified, it should contain the prediction on the depleted detector layer in form of `TrackParameters`.

3 Common Tracking Geometry

Geometry information is vital for the correct spatial interpretation of the detector measurements and for many reconstruction processes like track search and calculation of material effects. In the part of the ATLAS offline software which is relevant for common tracking there are three concepts providing such geometric information: the *detector elements*, the *tracking surfaces* and the *tracking geometry*. The detector readout elements describe the sensitive planes of the detector modules in the Inner Detector and Muon

[3]Keeping the information about the material on a track is mainly designed for global fitting methods, whereas iterative fitting techniques incorporate the material effects into the track extrapolation process.

Spectrometer. In addition to detector-specific geometry parameters, they also carry comprehensive information about the detector hardware.

With the tracking surfaces, a much more lightweight geometry description has been introduced which provides only the functionality necessary for tracking like position, orientation and boundaries. These surfaces are constructed from the detector elements and therefore take over automatically the current geometry parameters from the full ATLAS detector description, including the alignment corrections. The collection of different surface types is shown in Fig. 10. As already mentioned for the different versions of track parameter classes, the surface types cover more than needed to describe sensitive detector elements (sensor modules, TRT straws). This allows to introduce virtual surfaces, like barrel layers, endcap discs or cylindrical surfaces. The tracking surfaces are fully integrated with the tracking EDM: for example the `MeasurementBase` class and the `TrackParameters` class contain a pointer to the associated tracking surface. Only in the case of virtual surfaces does this object belong to the EDM class; for performance reasons the real detector surfaces have been re-integrated into the detector elements such that their objects are created only once during a reconstruction job. Most of the surface describing sensitive detector elements are actually proxies for the associated elements and simply forward all necessary geometrical information used in tracking.

The different kinds of surface class inherit from a common base class, `Surface`, carrying the interface structure. This structure is detailed in Fig. 11. Each surface is characterised by its individual local coordinate frame. As a consequence, a trapezoidal or rectangular planar surface are both described through one class implementation (`PlaneSurface`) but differ by their type of bounds. A planar disc surface, however, has its own class representation, since its local coordinate frame is not a Cartesian, but a polar one. Fig. 12 illustrates the different kinds of surface bounds. This concept is slightly broken through the existence of the so-called `PerigeeSurface`, which is a specific type of a `StraightLineSurface`. This separation has been done for speed reasons, since the Perigee representation is extensively used as track reference. Each surface carries the necessary information (or pointers to this information) to provide the transformation between local and global frame as well as the ability to check whether a global respectively local point is on the surface and inside the surface's boundaries.

Finally, a common geometry description inside the tracking realm has been introduced to give fast access to detector layers and passive material layers along a trajectory and to guarantee a consistent extrapolation and navigation. This geometry description has been designed to be as lightweight as possible and to be directly built from the common ATLAS detector description GeoModel. The purely geometrical classes can be found in the TrkSurfaces resp. TrkVolumes packages. They are extended with physical quantities such as material or magnetic field properties in the TrkGeometry package [15]. TrkVolumes and TrkGeometry are not relevant for the description of the Tracking Event Data model and will therefore be omitted in this document.

4 Data Primitives in the Tracking EDM

Common data primitives are used within all classes of the ATLAS Tracking EDM. They are mainly extensions or C++ type definitions of objects contained by the used maths library CLHEP. The choice of either extending an old or integrating a new naming schema has been taken to ease a possible future transition to a different underlying math library.

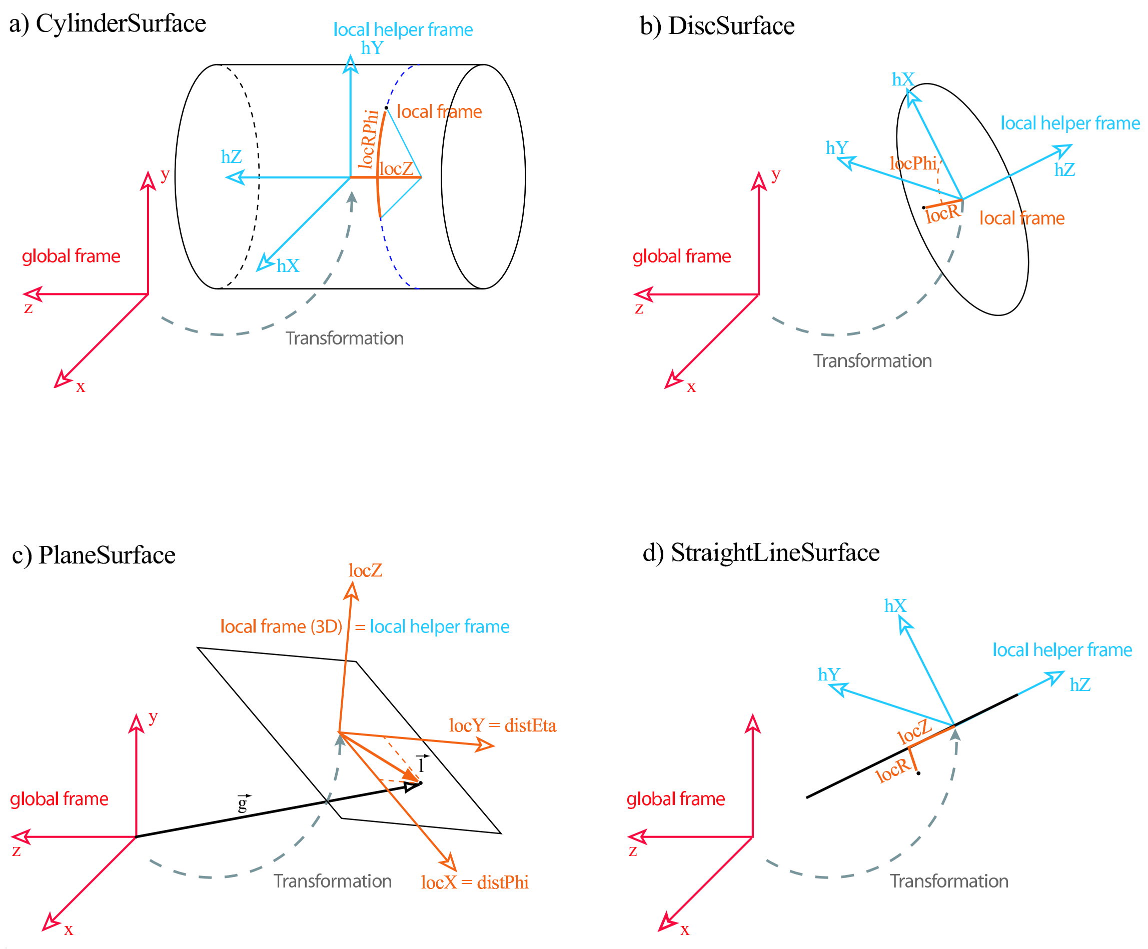


Figure 10: *The different surface types and their global to local transformations, as they are found in the* TrkSurfaces *package: (a) CylinderSurface, (b) DiscSurface, (c) PlaneSurface, (d) StraightLineSurface. The PerigeeSurface is omitted in this illustration, as it is a special case of a StraightLineSurface.*

4.1 Type definitions and simple extensions

Standard C++ `typedef` definitions have been used to identify frequently used classes in the ATLAS Tracking EDM, Table 3 presents an overview of the common primitive classes and their naming system, Table 4 lists additional EDM primitives classes that do not extend or `typedef` existing CLHEP classes.

4.2 The Parameter Identification Schema

Since the two local parameters of the full set of track parameters differ depending on the surface they are related to, an identification schema has been introduced also to increase code readability. A set of `ParamDefs` enumerators are given and should be used for retrieving respectively defining parameters of common Tracking EDM primitives.

A special helper class `DefinedParameter` taking a double and one of the enumerators as construction input can be used to specify explicitly the meaning of a single parameter in an ambiguous context.

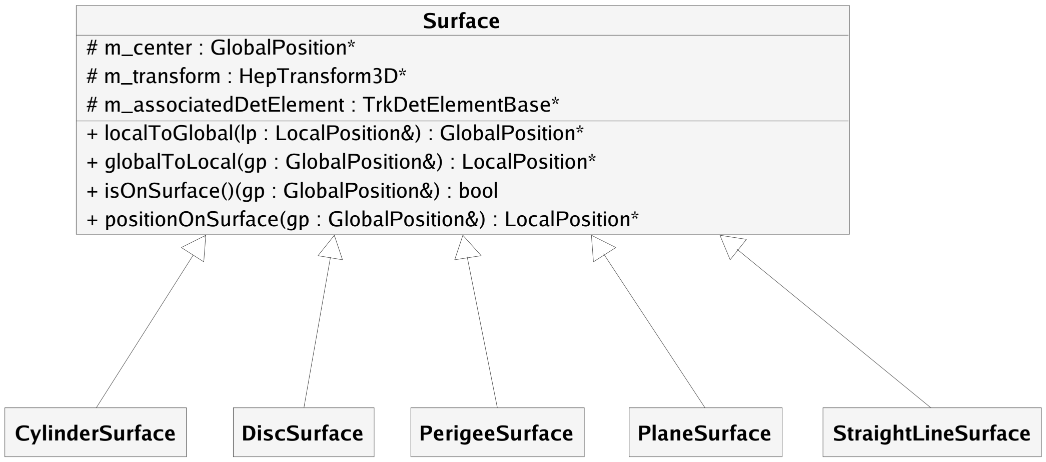

Surface
# m_center : GlobalPosition*
# m_transform : HepTransform3D*
# m_associatedDetElement : TrkDetElementBase*
+ localToGlobal(lp : LocalPosition&) : GlobalPosition*
+ globalToLocal(gp : GlobalPosition&) : LocalPosition*
+ isOnSurface()(gp : GlobalPosition&) : bool
+ positionOnSurface(gp : GlobalPosition&) : LocalPosition*
CylinderSurface
DiscSurface
PerigeeSurface
PlaneSurface
StraightLineSurface

Figure 11: *UML diagram showing the inheritance structure for tracking surfaces. All surfaces inherit from a common base class,* `Surface` *and overwrite transformation methods from local to global coordinates according to the intrinsic natural frame.*

CLHEP class	**EDM class**	**relation**	**description**
`HepPoint3D`	`GlobalPosition`	typedef	3D point in the tracking frame
`HepVector3D`	`GlobalMomentum`	typedef	3D momentum in the tracking frame
`HepVector3D`	`GlobalDirection`	typedef	3D direction in the tracking frame
`HepVector2D`	`LocalPosition`	typedef	2D position in a surface frame
`HepSymMatrix`	`CovarianceMatrix`	extension	matrix representation for covariances
`HepSymMatrix`	`WeightMatrix`	extension	matrix representation for weights
`HepVector`	`LocalParameters`	extension	flexible set of identified parameters

Table 3: *Standard C++ type definitions and simple extensions used in the TrkEvent-Primitives package.*

EDM primitives class	**description**
`LocalDirection`	representation of a 3D direction vector w.r.t. a plane
`ErrorMatrix`	holder class for `CovarianceMatrix` and `WeightMatrix`

Table 4: *Additional EDM primitives objects with no relationship to CLHEP objects.*

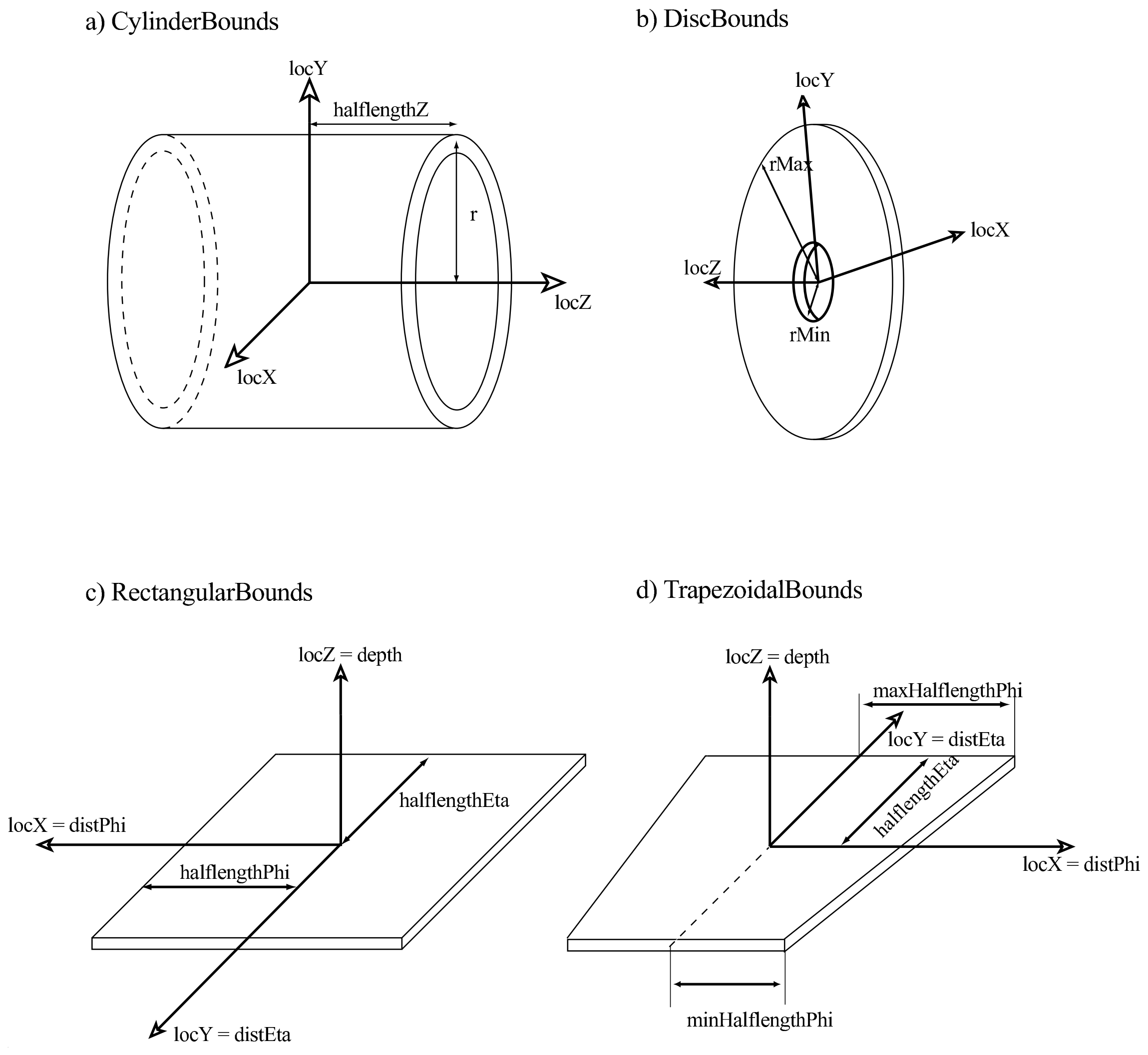


Figure 12: *The different bound types for surfaces in the* TrkSurfaces *package: (a) CylinderBounds for a CylinderSurface and a StraightLineSurface, (b) DiscBounds for a DiscSurface, (c) and (d) RectangleBounds and TrapezoidBounds for a PlaneSurface. The PerigeeSurface does not have any bounds by definition. By providing an opening angle for CylinderBounds and DiscBounds a cylindrical window respectively a disc segment can be constructed.*

Parameter key	binary	l_1	l_2	ϕ	θ	q/p	comment
31	11111	1	1	1	1	1	full set of track parameters
15	01111	1	1	1	1	0	set without q/p information
2	00010	0	1	0	0	0	only second local coordinate
1	00001	1	0	0	0	0	only first local coordinate

Table 5: *Examples for sets of local parameters, the associated key and its meaning.*

4.3 Concept of the LocalParameters Class

The concept of `LocalParameters` had to be introduced with the establishment of the track segments. As track segments combine information of multiple calibrated measurements, they represent in general more than just local measurement information without necessarily providing the full mathematical track information such as the `TrackParameters` class. `LocalParameters` can expand from a representation of a single parameter (one of the two local measurement parameters, such as one of the three global momentum representations) up to the full set of five parameters, see Section 2.2. Therefore an internal scheme to identify the contained parameters is needed for consistent writing and reading from the `LocalParameters` class.

The identification of the contained parameters in the `LocalParameters` class is done by the `DefinedParameter` class and a binary encryption representing the inserted values. A fully measured set of local measurement parameters plus global momentum representation can be characterised by the binary number 11111. Each digit of value 1 means that this parameter is contained in the `LocalParameters`, whereas the digit of value 0 stands for the absence of this parameter. This binary number is then translated into an integer (which leads in this example to the number 31) and is then stored as a private member in the `LocalParameters` class. Within this scheme all 32 possibilities of different combinations of measured parameters out of the full parameter set $[loc1, loc2, \phi, \theta, \frac{q}{p}]$ can be represented. For convenience, the order of the parameter set is swapped when computing the binary number, such that the measurement of the first local parameter is not represented by the binary number 10000 but 00001. Table 5 shows some examples of different possible combinations and their binary/integer representation.

The `LocalParameters` class can be asked whether it contains a parameter or not: a Boolean is returned when a `ParamDefs` enum is given. Static data members of 32 expansion and 32 reduction matrices, respectively, expand the actual dimension to a full five dimensional representation, and vice-versa. These matrices allow to use the `LocalParameters` class and its associated local error matrix directly inside comparisons or calculations with `TrackParameters` objects. The according expansion or reduction matrix for a given set of defined parameters can be directly retrieved by an interface method, picking out the right matrix is – again – internally handled by the parameter key.

4.4 Jacobians

A set of different Jacobian matrices are provided within TrkEventPrimitives package that can be used to transform covariances or errors from one representation to another. Currently Jacobians for the following transformations and their inverse applications can be found :

- polar to cartesian on a planar surface frame
- $(\theta, q/p)$ to $(\cot\theta, q/p_T)$ for legacy transformations
- $(\phi, \theta, q/p)$ for a global error representation of the momentum

5 Track Summary Information

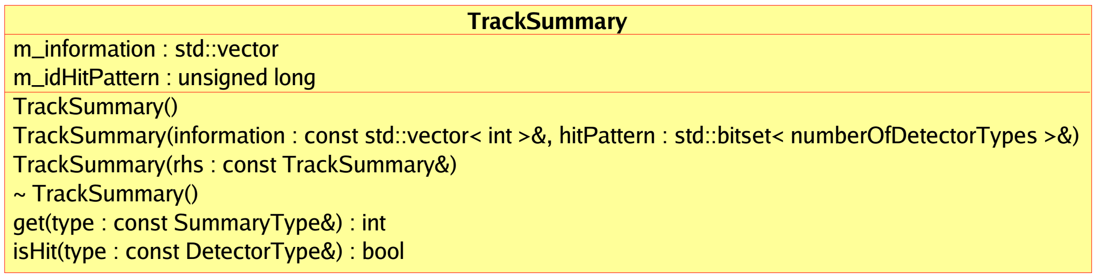

TrackSummary
m_information : std::vector
m_idHitPattern : unsigned long
TrackSummary()
TrackSummary(information : const std::vector< int >&, hitPattern : std::bitset< numberOfDetectorTypes >&)
TrackSummary(rhs : const TrackSummary&)
~ TrackSummary()
get(type : const SummaryType&) : int
isHit(type : const DetectorType&) : bool

Figure 13: *UML diagram showing the members and methods of the* `TrackSummary` *class.*

Tracks carrying the full trajectory information, that is measurements from different detector types and layers included in the track fit, in principle enable subsequent algorithms to evaluate precisely the track quality and hit patterns. To avoid that such algorithms need to provide sophisticated track analysis code with detailed knowledge about the subdetector classes, a `TrackSummary` class is provided with the Tracking EDM [16]. This class holds a set of counters about the number of hits in certain detector types together with a bit-set describing hit patterns within a detector. An external tool fills the counters and bit-set by analysing the input track and possibly using detector-specific knowledge from the Inner and Muon Detectors.

Furthermore, large-size `Track` objects cannot be kept as part of the strongly reduced data format intended as basis for the physics analysis. Instead the track summary information is stored there together with the reconstructed Perigee parameters.

6 Monte Carlo Truth

One of design decisions was to keep objects used by the track reconstruction software free of any references to simulation, thus guaranteeing that no hidden dependency of reconstruction on information that is not available for real data can exist. However to quantify performance of the software in terms of efficiency and fake rate, one needs to compare results of reconstruction with “Monte Carlo truth” information about simulated events. The possibility of such a comparison is implemented through a set of special association classes, which establish correspondence between reconstructed objects and MC truth data about them. These classes and relevant algorithms are described here, along with some information about MC truth itself.

A “truth” particle can be produced by either an event generator (such as Pythia), or later by GEANT, which propagates generated particles through the detector and may introduce new particles that are decay or interaction products. Information from both steps for the primary and any pile-up events is combined into a single McEventCollection. Not every simulated particle is stored, see [18] for details. The digitization uses GEANT hits to form Raw Data Objects (RDOs), which contain information similar to what can be read out from the real detector and serve as inputs to the reconstruction, and Simulated Data Objects (SDOs), which contain MC truth data.

No SDO should be created for noise RDO. So each SDO corresponds to an RDO from a real particle. Note that the reverse is not true. A real particle RDO may lack the corresponding SDO. Also, even when the SDO is present, the particle link in it may be missing. This happens when the producing particle is below e.g. a kinetic energy threshold, so it is not saved in GEANT. Particles from pile-up events have different thresholds than those from the signal event, and the thresholds may vary depending on the detector type (See the Monte Carlo Truth Task Force Report for details [18]). All in all, only two cases can be unambiguously distinguished:

- One or more valid HepMcParticleLinks correspond to an RDO. This RDO was produced by the particle(s).
- An RDO has no corresponding SDO, or the SDO has no valid HepMcParticleLinks. This RDO is produced by a low energy particle (from the signal or a pile-up event), or it is pure noise.

From RDOs the reconstruction forms `PrepRawData` objects, and eventually tracks. Truth information for those higher-level reconstruction objects is also created, from the corresponding SDOs. There are two paths of truth data handling: "classical truth" (historically the earlier implementation), and "detailed truth". Either or both of the paths can be activated at run time using job options.

6.0.1 Classical Truth

The implementation uses `std::map` based containers (`PrepRawDataTruthCollection`, `SpacePointTruthCollection`, and `TrackTruthCollection`) to allow for fast lookup of a truth object associated to a given reconstructed object. The use of `std::map` imposes the restriction of having no more than one truth object associated to a single reconstructed object. In ambiguous cases, e.g. when several `GenParticle`-s contribute to a cluster, the particle with the largest energy deposit is chosen. A similar weighting logic was used for track selection, but later it evolved to use a metric that is closer to a simple hit count.

6.0.2 Detailed Truth

Detailed Truth handling was developed to avoid premature loss of information in the truth matching process. To accomplish this, the PrepRawData level truth structure, `PRD_MultiTruthCollection`, stores information about all GenParticles contributing to a cluster or a drift circle. `PRD_MultiTruthCollection` is a `std::multimap<Identifier,HepMcParticleLink>`[4] , which is both simple and memory efficient. Only in a small fraction of cases there is more than one particle corresponding

[4]The decision to use as a key an Identifier (returned by the identify() method of a PrepRawData object) was to simplify persistency. A cleaner design would be to use a pointer (ElementLink) to PrepRawData instead.

to a cluster, so using instead a simple map to some class containing a collection of HepMcParticleLinks would almost double required memory size. (A collection class would need to either store the number of entries, or an end marker, in addition to the particle links.)

Another issue addressed by Detailed Truth is interaction recovery. Every time a particle undergoes an interaction causing GEANT to record an interaction vertex, it changes its "barcode" and is represented by a new C++ object, although it may still correspond to the same "physical" particle (e.g. in the case of bremsstrahlung). To handle such situations, the concept of `TruthTrajectory` is used. A `TruthTrajectory` is defined as a sequence of charged `GenParticle`-s connected through the mother-daughter relationship.

The central data class in this truth implementation is `DetailedTrackTruthCollection`. This is a `std::multimap` of reconstructed tracks into `DetailedTrackTruth`; the latter class containing a `TruthTrajectory` and per subdetector information about the number of hits shared between the track and the truth trajectory. It is straightforward to use `DetailedTrackTruthCollection` to define "the best match" for a track, with different definitions of what match is the best.

The `TrackTruthSelector` algorithm provides a definition of the "best match". It converts `DetailedTrackTruthCollection` into the "classic truth" format `TrackTruthCollection`. This makes details of the truth handling transparent to the downstream clients, which only use a `TrackTruthCollection` as their input.

7 Conclusions

The common event data model for tracking in the ATLAS experiment has been presented. It is integrated into the ATLAS reconstruction chain by providing object representations of calibrated measurements and detecting surfaces which can be used directly inside track search and fit algorithms. On the output side it offers several objects: `TrackParameters`, `TrackSegment` and `Track` to store the results from the tracking expressed on a single surface, detector region or any part of the whole experiment, respectively. Additional concepts of multiple track states or multiple measurements per surface, which are needed by novel track reconstruction techniques, have been been integrated seamlessly into the existing design.

The data model strictly follows an open design of base classes to provide all information needed for track fitting and handling in a way which is entirely independent of detector technology. Only the child classes are implemented in the sub-detector realms of Inner Detector and Muon Spectrometer, allowing them to store extra information, to apply specific calibrations and to decide about their persistent data model without affecting the interfaces to the tracking algorithms. A similar design is employed for the geometric part, in particular the detector surfaces, where many geometry-specific calculations are encapsulated by the child classes.

This document describes the tracking data model as it is part of the ATLAS offline software 12.0.0 release. In the course of previous releases, when reconstruction algorithms gradually worked more and more with the data model objects, its design has been refined in detail, but the general concept of making the basic objects detector-independent has proved to be stable and successful. It is now being widely used by reconstruction algorithms working in different parts of the ATLAS detector and by the software commissioning with test-beam and cosmic data.

Other parts of the ATLAS data model, such as the detector-dependent or the physics analysis EDMs, have been developed in parallel to the Tracking EDM. They are described in separate documents.

8 Appendix

8.1 Nomenclature

Expressions that represent actual classes or software implementations in C++ are written in `Courier face`. Packages grouping several classes in the ATLAS software repository are written in sans-serif face.

8.2 Definition of the sign of d_0 and the drift radius

For the representation of track parameters at a Perigee and at a straight line, the first local coordinate is a signed parameter. This is the impact parameter, d_0, for the Perigee and the drift radius, r, for the straight line. In the following the definition of the sign for a measurement in respect of the general straight line is shown. By setting the local z axis to be the global one, this transforms into the Perigee case.

May $\vec{m}$ denote the global position of the point of closest approach to the line and $\vec{p}$ the global momentum at this stage. The line is characterised by a local three dimensional frame, where the local z axis points along the line's direction, $\vec{l_z}$ is the unit vector into this direction. Denoting $\vec{c}$ to be the vector to the line centre coordinates and $\vec{d} = \vec{m} - \vec{c}$ the vector pointing from the line centre to $\vec{m}$, the sign of the drift radius, $sign(d)$, is then defined as

$$sign(d) = sign\left((\vec{p} \times \vec{l_z}) \cdot \vec{d}\right) \quad . \tag{5}$$

The corresponding quantities are illustrated in Fig. 14. The $\vec{l_z}$ vector points in the direction of the local z axis.

This is consistent with the following convention for the sign of d_0 in the Perigee representation: May ϕ denote the azimuthal angle to the Perigee position and ϕ_0 the azimuthal angle of the momentum in the Perigee. The sign of d_0 is then defined as positive, if $\phi - \phi_0 = \frac{\pi}{2} + n \cdot 2 \cdot \pi$, where $n \in Z_0$.

8.3 Persistency

An important consideration for the design of the Tracking EDM is that it must be possible to save ('persistify' in ATLAS parlance). This has many implications, some purely technical (it is not possible to store pointers to objects without a virtual table) and some which are fundamental design issues (e.g. how to persistify inter-object links). The solution first follows closely the concept of ownership between the EDM classes: the `Track` class, for example, has links to the `TrackParameters` and `MeasurementBase` objects which are only accessible through the `Track`, i.e. the `Track` *owns* those objects. In this case each concrete class of `TrackParameters` and `MeasurementBase` is saved together with the `Track`. With each persistent EDM package comes therefore a selection file determining

- which members need to be made persistent: data and objects with ownership;
- which members must be transient, i.e. not saved, but the link re-created when reading it in: objects owned by other entities;

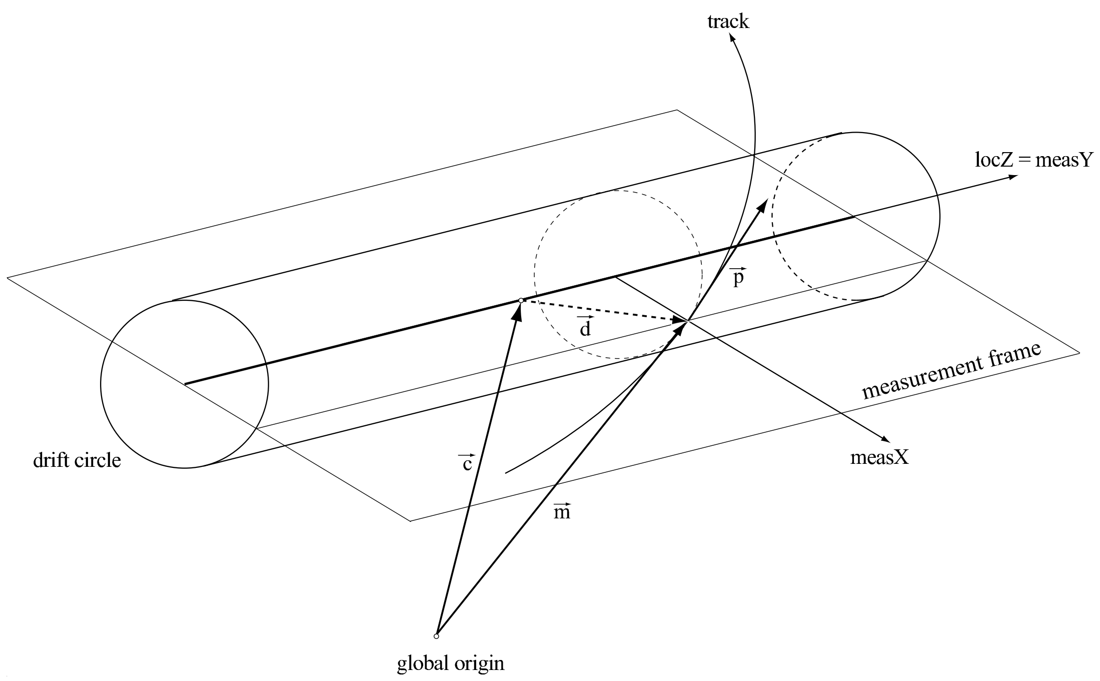


Figure 14: *Illustration of the necessary quantities used for the calculation of the sign of d_0 in a Perigee, respectively of the radius in e.g. a wire drift chamber.*

size	**in memory**	**compressed**
per event	2010 kB	593 kB
per track	34 kB	10 kB

Table 6: *Example for the size of track objects in memory and in a root file on disk (i.e. compressed). The numbers are obtained from reconstructed Inner Detector tracks on a sample of 25 simulated $t\bar{t}$ signal events.*

- which members must be transient and need not be re-created later: cached or otherwise economised information.

Since the restauration of inter-object links (e.g. by analysing an identifier) involves algorithmic code, tools are employed to convert the persistifed objects back to event data objects. Another serious constraint is disk-space: for example, it will not in general be possible to store the `PrepRawData` collections, and so the links from the `RIO_OnTracks` to `PrepRawData` must be transient.

Taking tracks reconstructed in the Inner Detector as an example, Table 6 shows the sizes of the `Track` container and objects in memory and on disk. They were obtained with release 12.0.0 on $t\bar{t}$ events (without pile-up and underlying event) which lead to about 60 tracks per event with an average of 35 hits per track in the Inner Detector. The compressed data sizes reflect the amount stored in the Event Summary Data (ESD), a format allowing parts of the reconstruction (e.g. re-calibrations and re-fits) to be redone. The Analysis Object Data (AOD), i.e. inputs for the physics analyses, employ a different data model which strongly reduces the event size. In there `Track` objects are discarded and only the `TrackSummary` and `PerigeeParameters` kept.

8.4 Examples of Track Creation and Use

Tracks are produced mainly by the track fitters as the estimate of the reconstructed track parameters together with the full trajectory information. The common track class and the generalised form of the abstract base classes `MeasurementBase` and `TrackParameters` allow to implement track fit algorithms in a purely mathematical and detector-independent way.

Tracks then serve as internal objects for some of the alignment and calibration algorithms as well as for the later phase of event reconstruction when information from different detectors is combined. They are the input objects for the algorithms reconstructing high-level physics objects such as vertices and identified electrons. Information from tracks is extracted for detector monitoring and event display and, in reduced-size form, for physics analyses.

8.4.1 Creating a simple Track

The simplest possible track is one which contains only the Perigee information. The following code-snippet will create one of these tracks.

```
// assuming values already defined.

const Trk::FitQuality*      fitQuality      =
      new FitQuality(chiSquared,numberDegreesOfFreedom);
const Trk::Perigee*         perigee         =
     new Perigee(d0,z0, phi, theta,  qOverP );

DataVector<const Trk::TrackStateOnSurface>* trackStateOnSurfaces
    = new DataVector<const Trk::TrackStateOnSurface>;

//  put the perigee object into a TrackStateOnSurface

std::bitset<TrackStateOnSurface::NumberOfTrackStateOnSurfaceTypes> typePattern(0);
typePattern.set(TrackStateOnSurface::Perigee);
const Trk::TrackStateOnSurface* trackStateOnSurface =
        new TrackStateOnSurface(0, perigee, 0,  0, typePattern);
trackStateOnSurfaces->push_back(trackStateOnSurface);

//create the track

Trk::Track myTrack = new Track(Track::KalmanFitter,
                               trackStateOnSurfaces,
                               fitQual);
```

More sophisticated tracks with information e.g. about the contained measurements and the reconstructed parameters at each detector surface are usually produced within a track fitter.

8.4.2 Getting Information from a Track

To extract information about the trajectory a loop over the `TrackStateOnSurface`s must be set up:

```
void MyClass::analyseTrack(const Trk::Track&  inputTrack) const
{
  const DataVector<const TrackStateOnSurface>*
    tsos = inputTrack.trackStateOnSurfaces()
  if (tsos && !tsos->empty()) {
    DataVector<const TrackStateOnSurface>::const_iterator it    = tsos->begin();

    // loop over track states
    for ( ; it!=tsos->end(); ++it) {

      // analyse only states with detector hits
      if ((*it)->type(Trk::TrackStateOnSurface::Measurement) ||
          (*it)->type(Trk::TrackStateOnSurface::Outlier)     )
      {
         const Trk::MeasurementBase* measurement = (*it)->measurementOnTrack();
         const RIO_OnTrack* rot = dynamic_cast<const RIO_OnTrack*> (measurement);
         if (rot) analyseMeasurement( (*it)->measurementOnTrack(),
                                      (*it)->trackParameters()    );
      }
    }
  }
}
```

8.5 Utilities to Work with the Tracking EDM

A couple of useful classes to perform simple tasks on the EDM classes are also provided, located in the package TrkEventUtils [17]. These classes are:

- Definitions for use of EDM classes in STL sorting functions:
  `PrepRawDataComparisonFunction`,
  `MeasurementBaseComparisonFunction`,
  `RIO_OnTrackComparisonFunction`,
  `TrackStateOnSurfaceFunction`,
  They define a geometrical "is less than" relation between two EDM objects such that the STL methods can be used to sort a set of hits onto a trajectory or locate the nearest/farthest hit. The input when constructing any of the comparison functions can be either a reference point, a reference radius or a reference direction according to which the relation will be defined.

- Measurement extractors solve the problem that the `Track` class does not provide easy access to the original `PrepRawData` or `RIO_OnTrack` objects. The classes performing this task are `CreatePRD_VectorFromTrack` and `RoT_Extractor`.

8.6 Validation Tools

8.6.1 Event Data Model Monitor

To try to help track down memory leaks, we have developed a simple tool which tracks the number of instantiations of certain key tracking objects (see Table 7). The way this works is that the base classes increment a variable for each call of their constructors and decrement it for each call of their destructors, which allows us to know how many objects exist at any time.

Package	Class
TrkSurfaces	`Trk::Surface`
TrkPrepRawData	`Trk::PrepRawData`
TrkRIO_OnTrack	`Trk::RIO_OnTrack`
TrkSegment	`Trk::Segment`
TrkSpacePoint	`Trk::SpacePoint`
TrkTrackSummary	`Trk::TrackSummary`
VxVertex	`Trk::Vertex`

Table 7: *List of classes (and their packages) which are currently instrumented for use in the EDM Monitor.*

Since StoreGate should delete EDM objects at the end of an event[5], if the tool finds that the counters are not zero at the start of the next event it is probable that a leak has been found.

For reasons of performance the Event Data Model Monitor only works in debug builds and will print an error message if run in opt.

References

[1] V. Boisvert et al, Final Report of the ATLAS RTF, `ATLAS note ATL-SOFT-2003-010`

[2] ATLAS Collaboration, ATLAS Computing TDR, `CERN-LHCC-2005-022`

[3] `http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas` : Tracking/TrkEvent/TrkTrack

[4] L. Lönnblad, `Comp. Phys. Comm. 84 (1994) 307.` `http://wwwasd.web.cern.ch/wwwasd/lhc++/clhep/`

[5] `http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas` : Tracking/TrkEvent/TrkEventPrimitives

[6] A. Strandlie and R. Frühwirth. Track fitting with ambiguities and noise: a study of elastic tracking and non-linear filters, `Comp. Phys. Comm. 120 (1999) 197.`

[7] `http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas` : Tracking/TrkEvent/TrkMultiComponentStateOnSurface

[8] `http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas` Tracking/TrkEvent/TrkParameters

[9] `http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas` : Tracking/TrkEvent/TrkPrepRawData

[10] `http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas` : Tracking/TrkEvent/TrkMeasurementBase

[5]The exceptions to this rule are the detector surfaces, some of which are kept in DetectorStore instead of StoreGate. A future extension will independently track the surfaces which are 'owned' by detector elements (and so persist for the entire run), and those which are 'free' (and so should be deleted at the end of the event).

[11] http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas : Tracking/TrkEvent/TrkRIO_OnTrack

[12] http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas : Tracking/TrkEvent/TrkCompetingRIOsOnTrack

[13] http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas : Tracking/TrkEvent/TrkSegment

[14] http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas : Tracking/TrkEvent/TrkSpacePoint

[15] http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas :
Tracking/TrkDetDescr/TrkSurface
Tracking/TrkDetDescr/TrkGeometry
Tracking/TrkDetDescr/TrkVolumes

[16] http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas : Tracking/TrkEvent/TrkTrackSummary

[17] http://isscvs.cern.ch/cgi-bin/viewcvs-all.cgi/offline/?cvsroot=atlas : Tracking/TrkEvent/TrkEventUtils

[18] Report from Monte Carlo Truth task force, version 2, 15th of November, 2005.