LHCb Note 2007-007
TRACKING

Tracking Event Model

E. Rodrigues

NIKHEF, Amsterdam, The Netherlands

J. A. Hernando

CERN, Geneva, Switzerland

September 2007

Abstract

This note presents the LHCb Tracking Event Model. It discusses the requirements and motivations and the subsequent choices that lead to the final implementation. The event model classes and tools as in software versions Brunel v30r17, LHCb v21r12, Lbcom v5r9 and Rec v3r14 are described; these versions were used for the physics production of the 2006 data challenge (DC06).

Contents

List of Tables

List of Figures

1 Introduction

This note describes the new Track Event Model endorsed by LHCb as of 2005 after the LHCb-wide review of all computing event models. The Tracking Event Model (TEM) was reviewed and approved by the "Track Event Model Task Force" defined by the Computing Technical Board.

The (present) model introduced in this note is the result of the development and reflects the conclusions and all the requirements imposed upon it during the review. It supersedes all other tracking event models used previously.

In the sub-section below we start with a reminder of the history of the TEM. In section 2 basic definitions are given, so as to clarify the concepts at hand. The following two sections, 3 and 4, explain in full the requirements and resulting design choices. A technical description of the C++ implementation of the TEM classes and tools is provided in sections 5 and 6, respectively. The last three sections deal with the access to data, the packaging and the pythonization of the TEM.

1.1 Motivations and brief history

In the past, before 2004, several track event models were available. There were essentially two groups of track models, for online and offline purposes. Furthermore, each environment had two track models. It was thought that such a situation was over-complicated and highly undesirable. Clearly, there is a lot to gain from having a common TEM used throughout all of the online and offline codes. Some benefits are:

- a standardised and universal model;
- less code to maintain;
- no duplication of similar code for essentially the same purpose;
- the event classes and tracking code can be shared transparently, without the need for converters.

Apart from the desired convergence of all event models, it has to be stressed that the old models were unable to deal with realistic detector geometries and a mis-aligned LHCb detector. Together, these two strong arguments motivated the review of the TEM.

The whole TEM implementation process was monitored regularly by a "Track Event Model Task Force" defined by the Computing Technical Board. It also took care of making sure the use cases and requirements were provided and met.

A re-design of the whole of the tracking code, both pattern recognition and track fitting, was therefore needed. For pattern recognition, the bulk of the philosophy has been retained; new ideas and algorithms have also been developed (such as a new approach to a seeding algorithm). For track fitting, a new fitting model was developed, based on the concept of trajectories, first introduced by the BaBar collaboration. The fitting model is fully described in [1].

2 Definitions

The review of the TEM started with a rather comprehensive discussion on basic definitions related to the event model classes. They are explained in the following sub-sections.

2.1 LHCbID

The LHCbID is a class that provides a single generic channel identifier throughout (the software of) LHCb. It makes available to the tracking a standardized and detector-independent channel identifier format for all the tracking detectors: the Vertex Locator (VELO), the Trigger Tracker (TT), the Inner (IT) and Outer (OT) Trackers and the Muon chambers.

2.2 Measurement

A measurement contains information on the tracking detector hit it originated from; it is the signature left by the particle passing an active element of the detector. A measurement is a cluster (or time, for the OT) that has been assigned to a track. There are different "types" of measurements corresponding to the several tracking sub-detectors.

2.3 State

A (track) state is a collection of parameters and their covariance matrix that defines a track in a given (finite) range of space.

States are defined in the global LHCb coordinate system. They represent straight line segments and are in general the result of the fit to a set of measurements on a track.

2.4 Track

In the LHCb model, a track is the representation of the path which a charged particle follows throughout the experiment.

The track representation is given in the canonical LHCb reference frame. This "global" representation is build from a set of "local" representations, the states [1]. As such, a track

[1] Note that "local" and "global" refer to the geometrical region in the experiment, not to the coordinate system.

can be seen as a collection of states. The "global" representation of the track's path is obtained via the states, which are parameterized as a function of an "external" parameter. Given the geometry of the experiment, this parameter is chosen to be the position projected onto the nominal beam-axis, *i.e.* the z-coordinate in the canonical LHCb reference frame.

A track is simply a list of LHCbIDs and a "seed" state in its minimal composition, given that measurements are uniquely identified by their LHCbID. The seed state and LHCbIDs are provided by the pattern recognition algorithm that found the track. The track is reconstructed from a collection of measurements, a modelling of the hits on the track. The track states are obtained as a result of fitting the measurements.

2.5 Node

A node is a class internal to the tracking code, not intended for general use. Its main purpose is to store information about the measurements and the track states at certain positions along a track's path, to easen the data access during the fit.

3 Requirements

In this section we list and discuss all the requirements imposed on the TEM.

3.1 General requirements

The TEM must comply with the general Gaudi design conventions and the LHCb event model separation of data and algorithms and tools. Data classes are simple objects whose data are then manipulated by dedicated tools and algorithms. The model classes must be described in XML using the Gaudi Object Description (G.O.D.). The header files are then automatically generated from the XML. Dependency between event model classes should be minimal.

3.2 Track requirements

A track should provide direct access to its parameterization at pre-defined positions along its path, *i.e* to the states it contains. These track states should be ordered according to their z-coordinate such that those states further from the track's origin vertex appear after those closer to it: for forward-going tracks [2], the order is in increasing z; for backward tracks, the order is in decreasing z.

[2]Forward/backward tracks are tracks propagating along the positive/negative z-direction (the LHCb detector goes also in the positive z-direction, with $z = 0$ being the nominal interaction point).

The distance and (χ^2) compatibility between a track's measurement and its associated state should be computable and available from the information contained on the track, be it for a track produced in the reconstruction phase (in Brunel) or for a track stored on and retrieved from a DST.

It should be possible to check the overlap of two tracks by counting and comparing the number of shared hits (identified by their LHCbIDs).

Following the Gaudi design rules, tracks stored on the transient event store (TES) should not be modified. Algorithms that update and/or modify tracks must do so by creating new tracks in different locations in the TES. An exception is accepted for track fitting, which updates the information on the track. But it should also be foreseen to publish fitted tracks on a different TES location from the input tracks found by the pattern recognition algorithms.

3.3 State requirements

A state represents (and stores) the track parameters and errors at a given point along the path, specified by a given z-position. All information relates to the global LHCb reference frame.

A state is required to contain the following information: track's position, direction (the so-called slopes), momentum and the corresponding covariance matrix. Given that the state's position and momentum have a redundant degree of freedom – since the z-coordinate is used to parameterize the states – the track's state is (locally) represented by the (x, y)-coordinate at the state's z-position, the slopes $(t_x = dx/dz, t_y = dy/dz)$ with respect to the x- and y-axes, and q/p, where q is the charge and p the magnitude of the momentum at that position.

A state of a "straight-line track" – such as a track in the VELO detector – should have a charge-over-momentum value $q/p = 0$. The model should naturally take this into account and always provide a momentum estimate that is consistent with it being a `Velo` track [3] or a track with momentum.

3.4 Measurement requirements

A measurement must contain information on the (tracking detector) cluster and type of cluster it originated from. Similar to states, a measurement is specified at a (defined) z-position; this is needed at the very least for ordering purposes.

[3] In LHCb a `Velo` track is a track that only contains hits from the VELO detector.

3.5 History requirements

A track must provide provenance in the form of a history. The history must comprise two parts:

- name of the pattern recognition algorithm that found the track;
- name of the algorithm that fitted and updated the track.

The history hence tells which algorithm "found" and made the track, and which algorithm updated and modified the track.

3.5.1 Pattern recognition history

The pattern recognition (PR) history identifies which algorithm has found the track and subsequently made it available in the TES.

Another type of PR history information may be the so-called ancestor tracks: PR algorithms such as the "matching" take as input a VELO and a T-stations seed track to form a long track. Information on these ancestors should be stored on the new track.

3.5.2 Fitting history

The fitting history identifies the specific track fitting algorithm. It also allows to distinguish between different modes of a single fitting algorithm. For instance, the fitting history should be able to tell whether the Kalman fit was "bi-directional" or "standard".

3.6 Fitting requirements

The track fit should be able to fit a track with different particle identification hypotheses, say as a pion or an electron. This effectively means applying different energy corrections for energy losses in material.

It should be possible to retrieve a track from a DST and refit it, using the minimalistic information stored therein, without having to rerun the PR algorithm. This requirement has consequences that overlap with the persistence requirements (see section 3.11): given a list of LHCbIDs stored on the track in the DST, it should be possible to reproduce the results of the track fit.

3.7 Goodness-of-fit and quality requirements

A track needs to provide "self-contained" access to information on its quality, in particular on the quality of the fit and whether it should be used or discarded for physics. Such quality quantities are:

- any quality variable defined in a pattern recognition algorithm;
- information on the status of the track fit – success or failure;
- goodness-of-fit information such as χ^2 and χ^2 probability;
- number of degrees of freedom.

This information should be quickly accessible, even after reading back a track from the persistent store (such as a DST); or readily re-made.

3.8 Extrapolation and transport requirements

One must be able to extrapolate a track and get a state at any specified point/location along its path, using a tool. The location at which the state is requested should at least be specifiable by either its z-coordinate or a plane. Other intersections (*e.g.* a spherical surface) are not considered to be necessary – for now. In the case of an extrapolation to a plane, it should be possible to specify the accuracy of the obtained state, *i.e.* how close the "target" state should be to the actual extrapolated state, by giving a tolerance. This accuracy may depend on the implementation and/or configuration of the actual extrapolator used.

The extrapolation may require access to the magnetic field and transport services in order to take into account both the magnetic field in the detector and the amount of detector material along the track's path. Both these aspects of the extrapolations must be user-configurable (likely via settings in options files).

3.9 Physics requirements

A physics analysis should be independent of the algorithm which produced the track(s). It should be possible, as far as an "end-user" analysis is concerned, to refit tracks when an improved alignment has been found, a new calibration performed, after adding or removing hits from a track, etc.

Physics analyses should also not have to deal directly with the extrapolation of tracks and the internal technicalities in doing so; the task should be delegated to the tracking code. In particular, there should be no need for explicit references to the track states actually used either for extrapolation or for providing information on the track: it is up to the tracking model to decide which states are best to use in order to provide the user-requested information such as position, momentum and corresponding covariance matrix.

3.10 Visualization requirements

The LHCb event display program, Panoramix, should be able to directly display tracks with no concern for the producing (PR) algorithm, and without the need for dedicated (user) configurations.

It should be possible to display the track's path as well as the measurements on the track - when requested.

3.11 Persistency requirements

It is crucial and mandatory to minimize the event size. The TEM should be optimized such that a minimum but comprehensive set of information is stored on those tracks persisted on DST.

The persistent representation should be as independent as possible from any external non-Gaudi software package. It should also not contain any duplication of information already present in the raw buffer, which is also stored on the DST (*e.g.* tracking detector clusters). Though minimal, the track data made persistent should be enough to recreate all the pattern recognition information, to provide the best possible state at any location in the detector, and to refit the track with the possibility of adding or removing measurements.

It is also important to allow for some flexibility in the amount of information persisted: by default some pre-defined states will be stored on the track, but a user configuration should be enough to change "on-the-fly" the set of stored states.

4 Design

In this section we will explain the main choices of design we have followed given the requirements and constraints discussed in the previous section 3.

4.1 General considerations

4.1.1 On-line and off-line frameworks

At the heart of the new TEM was the goal to design a model that could meet all the different, and at times relatively incompatible, requirements from the on- and off-line environments. As a general rule, we have tried to combine all the event model features needed by both environments. In case of incompatibility between the "old" on- and off-line models, a different approach was followed that would satisfy the needs. The set of classes has been kept to a minimum.

4.1.2 References versus Pointers

The C++ language is a complex language and gives substantial freedom to the user as far as the choices of implementation are concerned.

In the course of designing the new TEM we decided to make a more extensive use of references than in the old models, both in the tracking code and in particular in the tools (interfaces) methods. We have chosen to implement the tool interfaces with the design in mind that arguments use references instead of pointers to pass an object that will not be owned by the tool. As a consistency consequence, in the tracking code, user code that is passed a reference does not delete the object; if it is passed a pointer, the user code gets the ownership, and therefore the responsability to delete it, if and when needed. Let's take the following example:

```
StatusCode project( const State& state, Measurement& meas );
```

Just by the signature, and given our design choices, it then becomes clear that (1) the object `State` will not be changed (it is seen as a `const` object), and (2) the object `Measurement` may be changed internally by the method. Furthermore, as both are passed by reference, the method merely uses and/or updates the information, valid by definition, and both arguments are not to be deleted.

4.2 Classes

All event model classes have been defined in the `LHCb` namespace and implemented in XML. Other classes, in particular those specific to the fitting code, have been defined directly in C++, as no XML description is necessary for classes that are not to be made persistent.

We have chosen to make the "core" event model classes C++ base classes in order to have the flexibility of inheritance, if needed in the future. This goes for the following classes: `Track`, `State`, `Measurement` and `Node`. All other event model classes inherit from these (*e.g.* the concrete measurement classes). For the special cases of the `Track` and `State` classes, there is only one class; all those methods likely to require a change of implementation in the derived classes have been made `virtual`.

4.2.1 LHCbIDs and measurements

The new TEM has seen the introduction of the concept of an LHCbID compared to the old models (see its definition in 2.1). It has been introduced primarily to minimize the storage space required by the tracks.

LHCbIDs serve to uniquely identify the detector hits that were used to make the track. The measurements on a track are built from the list of LHCbIDs, in a one-to-one correspondence. Similarly to the LHCbID, a measurement contains information on the tracking sub-detector cluster. Measurements cannot be made persistent, by design. They are merely needed by the track fit – and typically built at the beginning of track fitting. The extra information on a measurement, compared to an LHCbID, is used by the track fit to calculate *e.g.* state-to-measurement residuals.

4.2.2 Pattern recognition versus fitted tracks

The tracking phase of the reconstruction contains two parts, pattern recognition, where tracks are found, and the subsequent fitting of the tracks. The list of LHCbIDs on a track is the outcome of the pattern recognition [4]; it specifies which tracking detector hits have been attributed to a track. By design, the list of LHCbIDs on a track output by a pattern recognition algorithm cannot be modified. Only the measurements can be deleted, *e.g.* during track fitting, when so-called outliers are removed. After a track has been fitted, the number of measurements may differ from the number of LHCbIDs, the difference corresponding to those "outlier measurements" removed by the fit.

4.3 Tools and interfaces

Every tracking tool implements a corresponding interface, to comply with the rules of the Gaudi framework. It may also happen that different tools implement the same interface, or follow an inheritance tree with intermediate concrete tools.

4.3.1 Extrapolators

The track extrapolators are tools that extrapolate a track to an input position and provide the user with the track parameters at that position. The track parameters are typically and handily provided via a track state.

Changes in the TEM also affect the implementation of the extrapolator set of tools. Care was taken in the redesign to minimize the adaptations. In particular, the whole structure was essentially kept as before.

Tracks (and their states) evolve in space following a "model": linear, parabolic, etc. Each "model" is implemented in a dedicated extrapolator tool, deriving from a common baseclass.

The extrapolators have two types of input signatures: the extrapolation of a track and the direct extrapolation of a track's state. The general user is strongly advised to use the former method; the direct extrapolation of a track's state serves mainly the internal tracking code. In fact, the "state extrapolators" are internally called by the "track extrapolators" for performing the actual calculations, making use of the closest state present on the track.

[4] Other information is also output, such as a minimum of a seed state, for example.

Apart from these types of input signatures, the requested positions of extrapolation can in practice mean one of several possibilities: track extrapolation to (1) a z-position, (2) a plane and (3) to a point.

The (track) extrapolation to a plane is done step-wise in a recursive way:

1. get the state from the track which is nearest to the plane;
2. find the z-position of the crossing of the linear extrapolation of the state to the plane;
3. extrapolate to this new z-position;
4. iterate the above until the distance from the extrapolated state to the plane is less than some tolerance value (user input).

By default a maximum of five iterations is allowed. No extrapolation is actually done if the closest state (on the track) is already within the tolerance.

The extrapolation to a point means in fact the extrapolation to the closest point (on the track) to the specified point.

The extrapolators set contains also a special extrapolator: the "master" extrapolator. It is recommended that the end-user calls this extrapolator directly for "standard" extrapolations. The "master" extrapolator internally performs the extrapolation with the "extrapolation models". The actual choice of which model to use (see above) is done via a small set of dedicated and configurable selector tools.

None of the extrapolation models takes into account the detector material, which results in multiple scattering (MS) and energy loss corrections. These corrections to both the state's vector (only the momentum variable of the state's vector is actually updated) and covariance matrix are controlled by the "master" extrapolator. The "master" extrapolation also handles the electron corrections in an appropriate manner.

5 Classes

This section discusses the technical aspects of the implementation of the TEM classes.

5.1 LHCbID class

The `LHCbID` class implements constructors for all the detectors channel identifiers (see table 1). There is a "LHCbID detector type" for each of these detectors: `Velo`, `TT`, `IT`, `OT`, `Rich`, `Calo` and `Muon`. They are all defined in an `enum`.

The class has essentially two types of methods: "is" and "ID" methods. The boolean "is" methods – *e.g.* `bool LHCbID::isOT() const` – answer whether the `LHCbID` is of a certain detector type. One can retrieve the detector-specific channel identifiers, the so-called "channel IDs", with the "ID" methods – *e.g.* `OTChannelID otID() const`.

General methods are available to (1) retrieve the detector type, `unsigned int detectorType() const`, and to (2) retrieve the internal detector-specific "channel ID", `unsigned int channelID() const`.

The `LHCbID` class defines a unique (private) data member, an `unsigned int`, which is the internal representation of the channel identifier. The detector-specific part is coded in 28 bits, leaving 4 bits to identify the sub-detector (type).

The (bitwise) comparison between two `LHCbIDs` is done at the level of their internal representation. But the `LHCbID` itself knows nothing about the internal representation of the actual detector-specific "channel IDs". For detailed detector-specific comparisons the task should be delegated to the "channel IDs" classes (*e.g.* to the `OTChannelID`).

LHCbID(	);
LHCbID(	unsigned int theID);
LHCbID(	const VeloChannelID& chanID);
LHCbID(	const STChannelID& chanID);
LHCbID(	const OTChannelID& chanID);
LHCbID(	const RichSmartID& chanID);
LHCbID(	const CaloCellID& chanID);
LHCbID(	const MuonTileID& chanID);

Table 1: Constructors for the `LHCbID` class.

5.2 Measurement classes

The TEM defines a set of measurement classes, with a base class, `Measurement`, and one implementation (concrete class) per tracking sub-detector. The inheritance diagram for all the measurement classes is shown in figure 1.

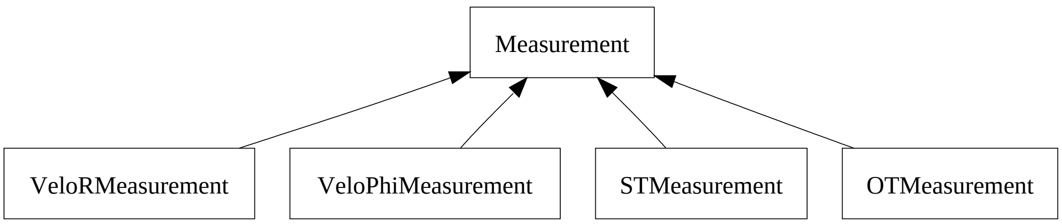

Figure 1: Inheritance diagram for the `Measurement` classes.

All the measurement classes contain one-dimensional information. Two-dimensional measurements are not foreseen at present, though they might be needed in the future, to accomodate for measurements in the muon stations.

A measurement is attributed a type and a z-position. It consists at least of a measurement value and error. As the measurements are built from the LHCbIDs on a track (see section 4.2.1), every `Measurement` instance contains the corresponding LHCbID.

The measurements are necessary for the LHCb track fit, whose dedicated fitting event model, described in [1], makes use of the concept of a trajectory [5]. Those trajectories are returned by the relevant tracking sub-detector given a LHCbID. The trajectory is conveniently stored in the measurement. Another object needed by the track fit is stored in this class: the (state) reference vector. It is used for instance during the projection of a track state onto the corresponding measurement.

In summary, the `Measurement` base class defines the following (protected) data members:

- the measurement type (`unsigned int`);
- the measurement value (`double`);
- the measurement error (`double`);
- the z-position of the measurement (`double`);
- the trajectory representing the measurement (`std::auto_ptr< Trajectory >`);
- the LHCbID corresponding to the measurement (`LHCbID`);
- the reference vector (`Gaudi::TrackVector` [6]);
- flag for the reference vector (`bool`).

The `Measurement` base class implements solely a default constructor and a copy constructor (see table 2).

Measurement( );
Measurement(const Measurement& other);

Table 2: Constructors for the `Measurement` base class.

All the measurement types are collected in an `enum`. The possible types are: `Unknown`, `VeloR`, `VeloPhi`, `TT`, `IT`, `OT` and `Muon`.

[5] In short, a trajectory is an object which contains a geometrical shape(*e.g.* a strip) and serves to decouple the tracking code from the specific detector geometry description.

[6] Several classes in the `Gaudi` namespace are found throughout the tracking code. The most commonly used are detailed in section 5.6

The measurement type can be retrieved, set and checked for, using, respectively, the methods `Measurement::Type type() const`,
`void setType (const Measurement::Type& value)` and `bool checkType( const Measurement::Type& value ) const`.

The measurement value can be accessed through `double measure() const`; and set with `void setMeasure(double value )`. *Mutatis mutandis*, there are methods to get and set the measurement error, the measurement z-position, the LHCbID, the trajectory corresponding to the measurement and the reference vector.

The boolean method `bool refIsSet() const` checks whether the reference vector has been set; it is used by the track fit.

5.2.1 VeloRMeasurement and VeloPhiMeasurement

The dedicated VELO measurements are implemented in the `VeloRMeasurement` and `VeloPhiMeasurement` concrete classes. Their available constructors are given in table 3. The first two methods are the default and copy constructors. Both other constructors take as input references to the VELO cluster `VeloCluster` from which the measurement is built, the VELO detector element `DeVelo` and the VELO "position tool" with interface `IVeloClusterPosition`.

VeloRMeasurement(	);
VeloRMeasurement(	const VeloRMeasurement& other);
VeloRMeasurement(	const VeloCluster& cluster, const DeVelo& det, const IVeloClusterPosition& clusPosTool);
VeloRMeasurement(	const VeloCluster& cluster, const DeVelo& det, const IVeloClusterPosition& clusPosTool, const Gaudi::TrackVector& refVector);

Table 3: Constructors for the `VeloRMeasurement` and `VeloPhiMeasurement` classes (The constructors for `VeloPhiMeasurement` are obtained replacing "R" by "Phi".).

The measurement value and error are calculated via the (specified) "position tool", given the input VELO cluster. For a `VeloRMeasurement` the measurement value is given by the radius of the sensor's strip plus the strip pitch times the fractional position (provided by the "position tool"); the measurement error corresponds to the local pitch of the sensor's strip times the fractional error (provided by the "position tool"). For a `VeloPhiMeasurement` the measurement value is given by the signed distance to the origin of the sensor's strip; the measurement error corresponds to the strip's ϕ-pitch (in radians) times the fractional error (provided by the "position tool").

Both classes store a pointer to the VELO cluster – of type R or ϕ, accordingly. The `LHCbID` stored in the class is built from the cluster's `VeloChannelID`. The z attribute of a `VeloRMeasurement` (`VeloPhiMeasurement`) is taken to be the z-coordinate of the R (ϕ) sensor (identified by the cluster's `VeloChannelID`). The reference vector is also saved when provided to the constructor.

A pointer to the VELO cluster can be obtained through the method `const VeloCluster* cluster() const`. Alternatively, the stored cluster can be set or updated via `void setCluster( const VeloCluster* value )`. A clone method for the `Measurement` class is also available.

5.2.2 STMeasurement

The TT and IT measurements are implemented in the `STMeasurement` concrete class.

The available constructors are given in table 4. The first two methods are the default and copy constructors. Both other constructors take as input references the ST cluster `STCluster` from which the measurement is built, the ST detector element `DeSTDetector` and the ST "position tool" with interface `ISTClusterPosition`.

STMeasurement(	);
STMeasurement(	const STMeasurement& other);
STMeasurement(	const STCluster& stCluster, const DeSTDetector& det, ISTClusterPosition& stClusPosTool);
STMeasurement(	const STCluster& stCluster, const DeSTDetector& det, ISTClusterPosition& stClusPosTool, const Gaudi::TrackVector& refVector);

Table 4: Constructors for the `STMeasurement` class.

The measurement value and error are calculated via the "position tool", given the input ST cluster. The measurement value is given by the sum of the local U (calculated from the strip) and the sector's pitch times the fractional position (provided by the "position tool"); the measurement error corresponds to the sector's pitch (the sector is found from the `STChannelID`) times the fractional error (provided by the "position tool").

The `STMeasurement` class stores a pointer to the ST cluster – of type `TT` or `IT`, accordingly. The `LHCbID` stored in the class is also built from the ST cluster's `STChannelID`. The z attribute of an `STMeasurement` is taken to be the z-coordinate of the center of the strip. The reference vector is also saved when given as input in the constructor.

A pointer to the ST cluster is returned by the method `const STCluster* cluster() const`. Alternatively, the stored cluster can be set or updated via `void setCluster( const STCluster* value )`. A clone method is also available.

5.2.3 OTMeasurement

The OT measurements are implemented in the `OTMeasurement` concrete class. The available constructors are given in table 5. The first two methods are the default and copy constructors. Both other constructors take as input references the OT time `OTTime` from which the measurement is built, the OT detector element `DeOTDetector` and the so-called left-right (LR) ambiguity.

OTMeasurement(	);
OTMeasurement(	const OTMeasurement& other);
OTMeasurement(	const OTTime& otTime, const DeOTDetector& det, int ambiguity=0);
OTMeasurement(	const OTTime& otTime, const DeOTDetector& det, int ambiguity, Gaudi::TrackVector& refVector);

Table 5: Constructors for the `OTMeasurement` class.

The LR ambiguity is stored in the measurement; it can take the values ± 1. The class also stores a pointer to the OT time. The measurement value is calculated as the ratio of the OT time's calibrated time (retrieved from `OTTime`) to the drift velocity (retrieved from `DeOTDetector`). The measurement error is presently fixed to the OT detector resolution of 200μm, as determined in beam tests. The `LHCbID` stored in the class is built from the OT time's `OTChannelID`. The latter serves to get the OT module's stereo angle, which is saved as the measurement's stereo angle. The z attribute of an `OTMeasurement` is taken to be the z-coordinate of the center of the straw (identified by the OT time's `OTChannelID`). The reference vector is also saved when specified in the constructor.

A pointer to the OT time is provided by the method `const OTTime* time() const`. Alternatively, the stored time can be set or updated via `void setTime( const OTTime*` `value )`. The LR ambiguity can be set and retrieved with `void setAmbiguity( int value )` and `int ambiguity() const`, respectively. The stereo angle is accessed via `double stereoAngle() const`. A clone method is also available.

5.3 State class

A state specifies the track parameters at one particular z-position along its path. The `State` class defines the following (protected) data members given the requirements and subsequent design choices:

- the z-position of the state (`double`);
- the state vector (`Gaudi::TrackVector`);
- the state covariance matrix (`Gaudi::TrackSymMatrix`);
- a bitfield containing a variety of state flags (`unsigned int`).

We define the 5-dimensional state vector as the state (transverse) position and direction (or slope), and the charge-over-momentum: $(x, y, t_x, t_y, q/p)$. The corresponding errors are stored in the 5-dimensional state covariance matrix.

The `State` class provides a default constructor and a constructor from a z-position, a state vector and covariance matrix, and a location (see table 6).

State(	);
State(	const Gaudi::TrackVector& stateVec, const Gaudi::TrackSymMatrix& cov, double z, State::Location& location);

Table 6: Constructors for the `State` class.

5.3.1 Enums

The available state locations are defined in the `State` class, in the `enum State::Location`; the possible values are `LocationUnknown`, `ClosestToBeam`, `FirstMeasurement`, `EndVelo`, `AtTT`, `AtT`, `BegRich1`, `EndRich1`, `BegRich2`, `EndRich2`, `Spd`, `Prs`, `BegECal`, `ECalShowerMax`, `EndECal`, `BegHCal`, `MidHCal`, `EndHCal` and `Muon`. They are used by both the pattern recognition and fitting algorithms to label useful track states that may not have the same precise z-coordinate but provide a handy snapshot of a track's parameters in a certain region of the LHCb detector.

One can check whether a state is at a given pre-defined location using `bool checkLocation( const State::Location& value ) const`. The location can be retrieved and set with `State::Location location() const` and `void setLocation( const State::Location& value )`, respectively.

5.3.2 State methods

Generally speaking there is always a "set" method for every "get" method. For simplicity we will restrict ourselves chiefly to a description of the "get" methods.

There are "direct" methods to access any of the state vector's quantities: *e.g.* the state x-position and slope along the y-axis – t_y slope – are returned with `double x() const` and `double ty() const`, respectively. When queried for its momentum via the method `virtual double p() const`, the `State` class returns either the absolute inverse value of the stored q/p or the `IEEE-754` standard infinity, `HUGE_VAL` (defined in `math.h`). This last option is solely used if one is dealing with a state representing a "straight line" – such as a track in the VELO detector – for which $q/p = 0$. The squared errors on all state parameters can also be retrieved individually. The squared error on the state's y-position is given via `double errY2() const`. Similar methods exist for all the state vector's parameters.

Other methods handily return several quantities "in one go". The stored state vector itself is returned with `Gaudi::TrackVector& stateVector()`, whereas `Gaudi::TrackSymMatrix& covariance()` returns its corresponding covariance matrix. Note that these methods have both a `const` and a non-`const` signature. The method `Gaudi::XYZPoint position() const` provides the state's position (3D-vector (x, y, z)). The corresponding error matrix is retrieved with `Gaudi::SymMatrix3x3 errPosition() const`.

A few "wrapper" methods provide in a single call most of the state's information: `e.g.` with `void positionAndMomentum (Gaudi::XYZPoint& pos, Gaudi::XYZVector& mom, Gaudi::SymMatrix6x6& cov6D ) const` one gets the 3D- position and momentum vectors together with the corresponding 6D-covariance matrix (the same method signature is available without a reference to the covariance matrix).

5.4 Track class

The `Track` class represents a track in LHCb. The class is rather complex in order to accomodate all the requirements coming from both the on- and off-line environments and from the needs of both pattern recognition and track fitting.

Two constructors are implemented: a default constructor and a constructor from an integer "container key" (see table 7).

The `Track` class defines the following (protected) data members:

Track( );	
Track(int key);	

Table 7: Constructors for the `Track` class.

- the number of degrees of freedom of the track (`int`);
- the χ^2 per degree of freedom (`double`);
- a vector of pointers to states, which are cloned and owned by the track (`std::vector< State* >`);
- a vector of LHCbIDs (`std::vector< LHCbID >`);
- a vector of pointers to measurements, which are cloned and owned by the track (`std::vector< Measurement* >`);
- a vector of pointers to nodes, owned by the track (`std::vector< Node* >`);
- a vector of references to ancestor tracks, not owned by the track (`SmartRefVector< Track >`);
- a bitfield containing a variety of track flags (`unsigned int`);
- a vector of additional information (`GaudiUtils::VectorMap< int, double >`).

The data members for the vectors of measurements and nodes are peculiar in that they are transient, *i.e.* they are not persisted once the track is saved on a file such as a DST.

5.4.1 Enums

The `Track` class defines a series of enums that serve to characterise the track: `Track::Types`, `Track::History`, `Track::PatRecStatus`, `Track::FitHistory`, `Track::FitStatus`, `Track::Flags` and `Track::AdditionalInfo`.

All enumeration values are set and stored on the track's bitfield data member; the exceptions are the `enum Track::AdditionalInfo` values, which are stored on the track's `VectorMap` of additional information (data member). The `Track::Flags` enumeration is different from the other bitfield enumerations in that it is "inclusive": a track can have any of the available `Track::Flags` enumeration values set at the same time. Conversely, a track has unique type, fit status, etc. Note that, by construction, all the `Track::AdditionalInfo` can also be set independently of each other, since they are used in a `VectorMap`.

The `enum Track::Types` specifies which are the possible track types: `TypeUnknown`, `Velo`, `VeloR`, `Long`, `Upstream`, `Downstream` and `Ttrack`.

The "history" `enum Track::History` contains all the names of pattern recognition algorithms that produce tracks. The possible track (PR) history values are `HistoryUnknown`, TrackIdealPR, TrackSeeding, PatVelo, PatVeloTT, TrackVeloTT, PatForward, TrackMatching, PatKShort and TsaTrack. Any track registered on the Transient Event Store (TES) should never have the default value, HistoryUnknown; every PR algorithm is responsible for setting the history flag as appropriate.

The enum Track::PatRecStatus specifies which state of the PR phase the track is in. A track contains LHCbIDs and/or measurements. Typically, the PR algorithms produce tracks with only LHCbIDs and set this flag accordingly. The fitting code then builds the measurements from the list of LHCbIDs and updates the PatRecStatus flag stored on the track. It is also the responsibility of every PR algorithm to specify what is the PR status of all the tracks that are output to the TES. The possible track (PR) status values are PatRecStatusUnknown, PatRecIDs and PatRecMeas. The default value, PatRecStatusUnknown, should never be found on any track.

The "fitting history" enum Track::FitHistory tells which fitting algorithm was used. For the moment the only available values are FitUnknown, StdKalman and BiKalman. These last two values refer to the same Kalman fitter, but to different run settings: standard or "bi-directional" Kalman filtering, respectively.

There is also a fitting status flag enumeration, the enum Track::FitStatus. Its values can be FitStatusUnknown, Fitted and FitFailed: they allow to check whether a fitter was called for this track and whether it succedeed. This status flag is always set by the fitting algorithms. Only tracks flagged with Fitted should be used in the remaining of the reconstruction and for physics analysis.

A "general flags" enumeration, enum Track::Flags, was introduced to comply with general needs. As of now the available values are Backward, Invalid, Clone, Used, IPSelected, PIDSelected, Selected, L0Candidate. The Backward, Invalid and Clone values are respectively used to flag all backward Velo tracks, to advertise a track invalid for any use (reconstruction and physics), and to flag a track as a clone of another track (it should then be discarded). The remaining enumeration values handily serve the (internal) needs of PR algorithms and the trigger.

The enum Track::AdditionalInfo contains additional information that can be assigned to a track by the PR algorithm that found it. At present the two values Likelihood and MatchChi2 are available, and used.

All the bitfield enumerations have "checking" methods to confirm a certain enum value is set:

- check the type of the track:
  `bool checkType ( const Track::Types& value ) const;`
- check the pattern recognition history of the track:
  `bool checkHistory( const Track::History& value ) const`

- check the pattern recognition status of the track:
  `bool checkPatRecStatus( const Track::PatRecStatus& value ) const;`
- check the fit history of the track:
  `bool checkFitHistory( const Track::FitHistory& value ) const;`
- check the fitting status of the track:
  `bool checkFitStatus( const Track::FitStatus& value ) const;`
- check the status of the general flag:
  `bool checkFlag( const Track::Flags& flag ) const.`

The enumeration value actually set is retrieved with the methods:

- retrieve the track type:
  `Track::Types type() const;`
- retrieve the pattern recognition algorithm that created the track:
  `Track::History history() const;`
- retrieve the pattern recognition status of the track:
  `Track::PatRecStatus patRecStatus() const;`
- retrieve the fitting algorithm that fitted the track:
  `Track::FitHistory fitHistory() const;`
- retrieve the fitting status of the track:
  `Track::FitStatus fitStatus() const;`
- retrieve the general flags:
  `Track::Flags flag() const.`

The enumeration values are set with:

- set the track type:
  `void setType( const Track::Types& value );`
- set the pattern recognition algorithm that created the track:
  `void setHistory( const Track::History& value );`
- set the pattern recognition status of the track:
  `void setPatRecStatus( const Track::PatRecStatus& value );`
- set the fitting algorithm that fitted the track:
  `void setFitHistory( const Track::FitHistory& value );`
- set the fitting status of the track:
  `void setFitStatus( const Track::FitStatus& value );`
- set one of the general flags:
  `void setFlag( unsigned int flag, bool ok ).`

The methods `unsigned int flags() const` and `void setFlags( unsigned int value )` also exist to retrieve and set the bitfield flags (the integer data member) altogether, respectively. This should be used with care.

Two extra methods deal with the bitfield data member: `unsigned int specific() const` and `void setSpecific( unsigned int value )` retrieve and set or update the track specific bits, respectively. These had not been discussed previously. In detail, the 32-bit bitfield data member is separated in several "sub-fields": 4 bits for the `Track::Types` values, 7 bits for the `Track::History`, 3 bits for the `Track::FitHistory`, 2 bits for the `Track::PatRecStatus`, 2 bits for the `Track::FitStatus`, 10 bits for the `Track::Flags` and 4 remaining "specific" bits unused so far. These "specific" bits allow the user to (privately) define and set 2^4 different enumeration values (via the two methods just mentioned).

The `VectorMap` of additional information and the related `Track::AdditionalInfo` enumeration values have dedicated "info" methods:

- check whether the track has information for the specified key [7]:
  `bool hasInfo( const int key ) const;`
- extract the information associated with the specified key (if there is no such information the default value will be returned):
  `double info( const int key, const double def ) const;`
- add or replace new information associated with the specified key:
  `bool addInfo( const int key, const double info );`
- erase the information associated with the specified key:
  `Track::ExtraInfo::size_type eraseInfo( const int key )` [8].

The methods `const ExtraInfo& extraInfo() const` and `void setExtraInfo( const ExtraInfo& value )` also exist to retrieve and update the additional information (the `VectorMap` data member), respectively. This should be used with care.

5.4.2 States

By design, a track should always contain at least one state – the "first state". It is the first element of the vector of states stored on a track. This "first state" can be retrieved with `State& firstState() const`. No valid track should ever have all its states deleted, and exceptions are thrown by methods that need the track's "first state" and do not find it.

The `Track` class provides a set of helper methods to manipulate the stored states:

[7] "key" refers here to any of the `Track::AdditionalInfo` enumeration values.

[8] `Track::ExtraInfo` is a `typedef` defined in the `Track` class: `typedef GaudiUtils::VectorMap< int, double > ExtraInfo`.

- retrieve the number of states on the track:
  `unsigned int nStates() const;`
- (`const`) retrieve a reference to the vector of pointers to all the states associated to the track:
  `const std::vector< State* >& states() const;`
- add a state to the list of states associated to the track:
  `void addToStates (const State& state );`
- remove a state from the list of states associated to the track:
  `void removeFromStates( State* value );`
- empty the content of the state vector associated to the track (the state vector data member is cleared):
  `void clearStates()`.

The states on the track are internally ordered, and stored in increasing z-position (decreasing z-position for `Track::Backward` tracks). For this reason the `void addToStates (const State& state )` method takes care of inserting the input state acording to its z value and the `Track::Backward` value set.

Removing states from a track should be handled with extreme care, not to break the rule of having at least a state on the track. In particular, no warning is issued if one removes all the states on the track, including the "first state".

Several methods exist to query the states on the track. The user may check for the existence of a state at a specific state location (see definitions in section 5.3.1) with the method `bool hasStateAt( const State::Location& location ) const`. The state can then be accessed via `const State& stateAt( const State::Location& location ) const`; a non-`const` version also exists. Note that an exception is thrown if a state at a non-existent location is requested - as these state locations do not have to be defined, and clearly the possible locations depend on the track type.

It is often desirable to get hold of the track state closest to a certain position – the extrapolators use this feature extensively. Two signatures exist (in their `const` and non-`const` versions), given as argument a z-position, or a plane:

- (`const`) retrieve the reference to the state closest to the given z-position:
  `const State& closestState( double z ) const;`
- (`const`) retrieve the reference to the state closest to the given plane:
  `const State& closestState( const Gaudi::Plane3D& plane ) const`.

An exception is thrown if no state is found since that would mean the track has no states defined.

5.4.3 LHCbIDs, measurements and nodes

As stated above, a track contains vectors of LHCbIDs, measurements and nodes. Just as for the states, a group of methods have been implemented to manipulate these vectors.

For the manipulation of the vector of LHCbIDs one has:

- retrieve the number of LHCbIDs on the track:
  `unsigned int nLHCbIDs() const`;
- check whether the given LHCbID is on the track:
  `bool isOnTrack( const LHCbID& value ) const`;
- (`const`) retrieve the vector of LHCbIDs:
  `const std::vector< LHCbID >& lhcbIDs() const`;
- add a LHCbID to the list of LHCbIDs associated to the track:
  `void addToLhcbIDs( const LHCbID& value )`;
- remove a LHCbID from the list of LHCbIDs associated to the track:
  `void removeFromLhcbIDs( const LHCbID& value )`;
- update the list of LHCbIDs associated to the track:
  `void setLhcbIDs( const std::vector< LHCbID >& value )`.

The list of LHCbIDs on a track, *i.e.* the actual (main) result of the PR, is by design frozen once the track is output to the TES. As a consequence, the method `void removeFromLhcbIDs( const LHCbID& value )` should never be used outside a PR algorithm.

There is a similar set of methods to manipulate and query the stored measurements:

- retrieve the number of measurements on the track:
  `unsigned int nMeasurements() const`;
- check whether the given measurement is on the track:
  `bool isOnTrack( const Measurement& value ) const`;
- check whether the measurement on the track corresponding to the input LHCbID is associated with the track:
  `bool isMeasurementOnTrack( const LHCbID& value ) const`;
- return the measurement on the track corresponding to the input LHCbID:
  `const Measurement& measurement( const LHCbID& value ) const`
- (`const`) retrieve a reference to the vector of pointers to all measurements:
  `const std::vector< Measurement* >& measurements() const`;
- add a measurement to the list associated to the track:
  `void addToMeasurements( const Measurement& meas )`;

- remove a measurement from the list of measurements associated to the track:
  `void removeFromMeasurements( Measurement* value )`.

A measurement is added to the track (see method above) if and only if its corresponding LHCbID is already present on the track. The measurements on the track are internally ordered, and stored in increasing z-position (decreasing z-position for `Track::Backward` tracks). The method `void addToMeasurements( const Measurement& meas )` takes care of inserting the input measurement according to its z value and the `Track::Backward` value set.

If one asks for a measurement given a LHCbID and it is not present on the track, an exception is thrown; the user is first required to check for its existence with the `bool isMeasurementOnTrack( const LHCbID& value ) const` method.

Furthermore, the `Track` class provides the method
`unsigned int nMeasurementsRemoved() const` to get the number of track measurements (outliers) removed when fitting the track (the number of LHCbIDs remains unchanged); this number is, by definition, equal to the difference between the number of measurements and LHCbIDs on the track.

One can add, remove and retrieve the nodes associated to a track with:

- (`const`) retrieve a reference to the vector of pointers to all nodes (a non-`const` version also exists):
  `std::vector< Node* >& nodes();`
- add a node to the list of nodes (note: track will take the ownership of this pointer!):
  `void addToNodes( Node* node );`
- remove a node from the list of nodes associated to the track:
  `void removeFromNodes( Node* value )`.

5.4.4 Ancestor tracks

As seen above, the `Track` class contains a "smart reference vector" data member that can be used to store pointers to ancestor tracks used to create the track at hand. Again, a series of methods are present for the manipulation of this information:

- add a track to the list of ancestors of this track:
  1) `void addToAncestors( const Track& ancestor );`
  2) `void addToAncestors( const SmartRef< Track >& value );`
  3) `void addToAncestors( const Track* value );`
- remove an ancestor track from the list of ancestors of this track:
  `void removeFromAncestors( const SmartRef< Track >& value )`.;

- (`const`) retrieve the vector of ancestor tracks that created this one (a non-`const` version also exists):
  `const SmartRefVector< Track >& ancestors() const;`
- empty the content of the ancestors vector of this track:
  `void clearAncestors().`

One should never call the method `void removeFromAncestors( const SmartRef< Track >& value )` outside a PR algorithm, since the list of ancestors is typically filled by a PR to specify, where applicable, the "sub-tracks" used to make the track at hand.

5.4.5 General methods

Several methods are available to retrieve basic properties of a track, such as its charge, momentum value and vector, slopes, χ^2 after fit, etc.

The track's assigned charge is given by `int charge() const`. There is no need for a "set" method since the track's charge is in fact determined from the q/p value of its "first state" (state vector data member, see 5.3).

Every fitted track is assigned a number of degrees of freedom and χ^2 per degree of freedom by the fitting algorithm. These quantities are accessed and set with:

- retrieve the number of degrees of freedom of the track (fit):
  `int nDoF() const;`
- retrieve the χ^2 of the track (fit):
  `double chi2() const;`
- retrieve the χ^2 per degrees of freedom of the track (fit):
  `double chi2PerDoF() const;`
- set the number of degrees of freedom of the track (fit):
  `void setNDoF( int value );`
- set the χ^2 per degrees of freedom of the track (fit):
  `void setChi2PerDoF( double value );`
- set the χ^2 and the number of degrees of freedom of the track (fit):
  `void setChi2AndDoF( double chi2, int ndof ).`

The number of degrees of freedom is defined as the number of measurements (after the track fit, `i.e.` after removal of possible outliers) minus the number of state parameters (5 in our model). The "set" methods are typically not used outside a fitting algorithm. Note, though, that both the χ^2 and the number of degrees freedom can be re-calculated from the information stored on the track (with the help of the track projectors).

The general user is most frequently concerned with the track properties at its production point and/or nearest to the beam-line. In our event model this track state is the "first state".

For this reason all the position and momentum related methods below give information at the "first state" position (they are, in some sense, shortcut methods):

- retrieve the position and momentum vectors and the corresponding 6D covariance matrix (position: 0→2,momentum: 3→5) at the first state:
  `void positionAndMomentum( Gaudi::XYZPoint& pos, Gaudi::XYZVector& mom, Gaudi::SymMatrix6x6& cov6D ) const;`
- retrieve the position and momentum vectors at the first state:
  `void positionAndMomentum( Gaudi::XYZPoint& pos, Gaudi::XYZVector& mom ) const;`
- retrieve the 3D-position and the corresponding covariance matrix at the first state:
  `void position( Gaudi::XYZPoint& pos, Gaudi::SymMatrix3x3& errPos ) const;`
- retrieve the 3D-position vector at the first state:
  `Gaudi::XYZPoint position() const;`
- retrieve the momentum vector and corresponding covariance matrix at the first state:
  `void momentum( Gaudi::XYZVector& mom, Gaudi::SymMatrix3x3& errMom ) const;`
- retrieve the momentum vector at the first state:
  `Gaudi::XYZVector momentum() const;`
- retrieve the momentum at the first state:
  `double p() const;`
- retrieve the transverse momentum at the first state:
  `double pt() const;`
- retrieve the slopes $(t_x = dx/dz, t_y = dy/dz, 1.)$ and the corresponding covariance matrix at the first state:
  `void slopes( Gaudi::XYZVector& slopes, Gaudi::SymMatrix3x3& errSlopes ) const;`
- retrieve the slopes at the first state:
  `Gaudi::XYZVector slopes() const;`
- retrieve the 6D (x, y, z, p_x, p_y, p_z) covariance matrix at the first state:
  `void posMomCovariance( Gaudi::SymMatrix6x6& cov6D ) const.`

5.4.6 Reset, clone and copy methods

The method `virtual void reset()` was introduced for the purpose of the trigger, to avoid copy construction. This method resets the track from all its contents, putting all the track data members to their default values.

Conversely, it may be desirable to copy the information from an input track to another track. The `virtual void copy( const Track& track )` does precisely this.

The clone method is different from the copy method in that it returns a pointer to a new track that is a perfect copy of the cloned track. The clone method calls internally the copy method. A track is cloned using the method `virtual Track* clone() const`; note that the caller then takes ownership of the returned pointer. A variant exists to clone a track and assign to the clone the same container key as the original track: `virtual Track* cloneWithKey() const`.

All the methods above are defined `virtual`, as they will need to be overridden by any class deriving from `Track`.

5.5 Node class

The node classes exist as helpers for the track fitting code. They are most useful for collecting information that is needed for or calculated during track fitting.

We have defined a `Node` base class from which any node-like class must inherit. At present LHCb provides a single (Kalman) track fitter; a single concrete class, `FitNode`, is therefore necessary. The inheritance diagram for the node classes is shown in figure 2. The concrete `FitNode` class is discussed in detail in [2], where our Kalman track fit is described.

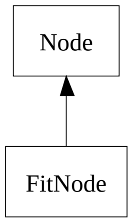

Figure 2: Inheritance diagram for the `Node` classes.

Being a helper class for fitting, any node-like class is typically made at any track state (estimate) position at which the fitting algorithm is requested to provide a best estimate of the track parameters. Any node is defined at a z-position that is the z of the state associated to the node. When the node refers also to a track measurement, both the state and the measurement are used in the track fit to project the state onto the measurement. The resulting residual (and error) are conveniently stored on the node. In summary, the `Node` base class defines the following (protected) data members:

- a pointer to the track state, owned by the node (`State*`);
- a pointer to the measurement, not owned by the node (`Measurement*`);
- the residual value (`double`);
- the residual error (`double`);

- the measurement error (`double`);
- the projection matrix (`Gaudi::TrackProjectionMatrix`).

Any `Node` class has to provide, by design, all the quantities listed above.

The `Node` base class implements a default constructor, a copy constructor and a constructor from pointers to a state and a measurement (see table 8). When instantiating via the default constructor both the state and the measurement data members are initialized as `NULL` pointers. The copy constructor copies all the content into the new node but it clones the state of the input node, since the state is owned by the node. The third constructor similarly clones the input state.

Node(	);
Node(	const Node& other);
Node(	State* state, Measurement* meas);

Table 8: Constructors for the `Node` class.

The following methods exist to manipulate the state and measurement stored on a node:

- retrieve a reference to the state on the node (a `const` version also exists):
  `State& state();`
- retrieve the 3D-position of the state on the node:
  `Gaudi::XYZPoint position() const;`
- update the state on the node:
  `void setState( const State& state );`
- check whether the node has a valid pointer to a measurement:
  `bool hasMeasurement() const;`
- retrieve a reference to the measurement on the node (a `const` version also exists):
  `Measurement& measurement() ;`
- remove the measurement from the node:
  `void removeMeasurement();`
- retrieve the measurement error:
  `double errMeasure() const;`
- retrieve the squared error on the measurement value:
  `double errMeasure2() const;`
- set/update the measurement error:
  `void setErrMeasure( double value ).`

For consistency, the `void setState( const State& state )` method internally clones the input state after deleting the previous state pointer stored on the node. When the measurement is removed from the node its corresponding pointer data member is set to `NULL` and the (data members) measurement value and residual value and error all set to zero (the node's χ^2 will then be zero as well).

The other basic node properties can be retrieved or set/updated as follows:

- retrieve the z-position of the node:
  `double z() const;`
- retrieve the residual value:
  `double residual() const;`
- retrieve the error on the residual:
  `double errResidual() const;`
- retrieve the squared error on the residual:
  `double errResidual2() const;`
- retrieve the local χ^2:
  `double chi2() const;`
- set/update the residual value:
  `void setResidual( double value );`
- set/update the error on the residual:
  `void setErrResidual( double value );`
- (`const`) retrieve the projection matrix:
  `const Gaudi::TrackProjectionMatrix& projectionMatrix() const;`
- set/update the projection matrix:
  `void setProjectionMatrix( const Gaudi::TrackProjectionMatrix& value ).`

The node χ^2 is defined as the ratio of the residual divided by its error, squared. The χ^2 is non-zero only for nodes with an associated measurement.

A "reset" method, `virtual void reset()` is also provided to clear a node: it sets all `double` data members to zero and the state pointer to `NULL` (after deleting the pointer). Conversely, a node can be cloned via `virtual Node* clone() const`.

5.6 Helper classes

In order to make the tracking code less coupled to the underlying mathematical packages, a helper file `TrackTypes.h` contains `typedefs` to vectors and matrices extensively used in the tracking code - they themselves are defined in the Gaudi framework as a function of native ROOT classes:

`typedef Gaudi::Vector5 TrackVector`: 5-dimensional vector type;

`typedef Gaudi::Matrix5x5 TrackMatrix`: 5-dimensional matrix type;

`typedef Gaudi::SymMatrix5x5 TrackSymMatrix`: 5-dimenional symmetrix matrix type;

`typedef Gaudi::Matrix1x5 TrackProjectionMatrix`: 1x5 matrix type.

Some of the state locations (see the `Location enum` in section 5.3) defined in the `State` class are largely used by the fitting algorithms to populate the fitted tracks with states at "reference" locations; the set of these fitted state locations depends solely on the track type. For consitency purposes throughout the tracking code it proved helpful to associate these commonly used locations with well-define z-positions. They have been collected in the helper file `StateParameters.h`. The possible values (in the `StateParameters namespace`) are:

`ZBegRich1 = 990. mm`;

`ZEndRich1 = 2165. mm`;

`ZAtT = 9410. mm`;

`ZBegRich2 = 9450. mm`;

`ZEndRich2 = 11900. mm`.

6 Tools

This section discusses the technical aspects of the implementation of the TEM tools.

6.1 Extrapolators

All the extrapolator tools derive from a common interface and an intermediate base class; the inheritance diagram for the track extrapolators is shown in figure 3.

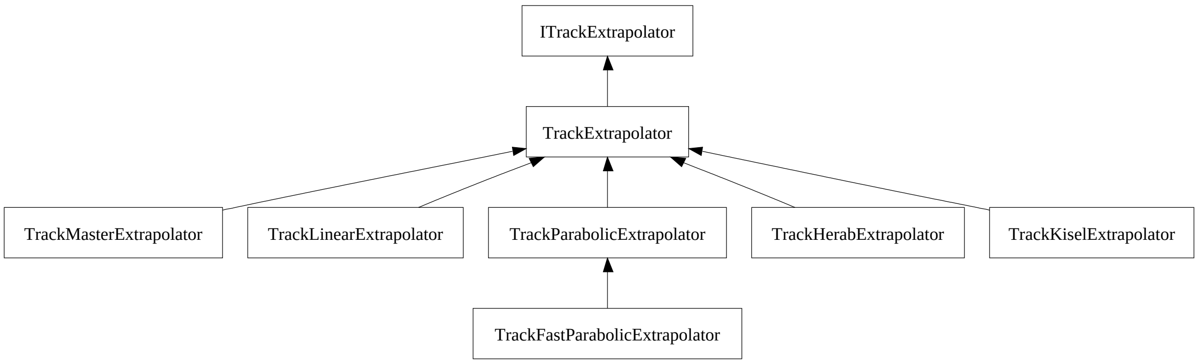

Figure 3: Inheritance diagram for the track extrapolators.

6.1.1 ITrackExtrapolator interface

All the extrapolators signatures are defined in the `ITrackExtrapolator` interface; the complete set is presented in tables 9 to 11. It comprises "propagation" and "access" methods.

Table 9 lists the available signatures for the propagation methods. There are two types of input signatures: the extrapolation of a track and the direct extrapolation of a track's state. Furthermore, one can extrapolate to a z-position and to a position closest to a plane or closest to a point.

virtual StatusCode propagate(	const Track& track, double z, State& state, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode propagate(	State& state, double z, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode propagate(	State& state, const Gaudi::XYZPoint& point, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode propagate(	State& state, Gaudi::Plane3D& plane, double tolerance = 0.01, ParticleID pid = ParticleID(211)) = 0;

Table 9: Extrapolators signatures for the "propagation" methods.

The interface also contains a few "access" methods providing direct (handy shortcuts) information on the parameters of the extrapolated track or state (tables 10 and 11).

6.1.2 TrackExtrapolator base class

The `TrackExtrapolator` base class is a tool deriving from the `ITrackExtrapolator` interface. It implements all the "access" methods (see tables 10 and 11) and those "propagate" methods (see table 9) whose implementations are not model-dependent (refer to section 4.3.1 for a discussion of the extrapolation models).

virtual StatusCode positionAndMomentum(	const Track& track, double z, Gaudi::XYZPoint& pos, Gaudi::XYZVector& mom, Gaudi::SymMatrix6x6& cov6D, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode positionAndMomentum(	const Track& track, double z, Gaudi::XYZPoint& pos, Gaudi::XYZVector& mom, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode position(	const Track& track, double z, Gaudi::XYZPoint& pos, Gaudi::SymMatrix3x3& errPos, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode position(	const Track& track, double z, Gaudi::XYZPoint& pos, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode slopes(	const Track& track, double z, Gaudi::XYZVector& slopes, Gaudi::SymMatrix3x3& errSlopes, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode slopes(	const Track& track, double z, Gaudi::XYZVector& slopes, ParticleID pid = ParticleID(211)) = 0;

Table 10: Extrapolators signatures for the "access" methods (part I).

6.1.3 TrackMasterExtrapolator

The "master" extrapolator `TrackMasterExtrapolator` is special in that it steers which "extrapolation model" to be used and applies multiple scattering (MS) and energy loss (dE/dx) corrections (see also the design discussion in section 4.3.1). The actual choice of which model to use depends on the tool configuration.

virtual StatusCode p(	const Track& track, double z, double& p, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode pt(	const Track& track, double z, double& pt, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode momentum(	const Track& track, double z, Gaudi::XYZVector& mom, Gaudi::SymMatrix3x3& errMom, ParticleID pid = ParticleID(211)) = 0;
virtual StatusCode momentum(	const Track& track, double z, Gaudi::XYZVector& mom, ParticleID pid = ParticleID(211)) = 0;
virtual const Gaudi::TrackMatrix&	transportMatrix() const = 0;

Table 11: Extrapolators signatures for the "access" methods (part II).

It is recommended that the end-user calls the `TrackMasterExtrapolator` directly for "standard" extrapolations, *e.g.* for extrapolations in physics analyses where energy corrections need to be taken into account. The "model" extrapolators should be used directly when dealing solely with propagations, regardless of energy correction issues. Indeed none of the extrapolation models takes into account the detector material, which results in MS and energy loss corrections.

6.1.4 Extrapolator models

There are five extrapolation models available:

- `TrackLinearExtrapolator`:
  linear extrapolator;

- `TrackHerabExtrapolator`:
  extrapolation with a 4^{th}- or 5^{th}-order Runge-Kutta method for the solution of the equation of motion of a particle in a magnetic field;

- `TrackParabolicExtrapolator`:
  parabolic extrapolator;

- `TrackFastParabolicExtrapolator`:
  parabolic extrapolator with a fast and approximate calculation of the transport matrix;

- `TrackKiselExtrapolator`:
  extrapolator using an analytic formula (a power series expansion in terms of the magnetic field) for the extrapolation in non-homogeneous magnetic fields.

6.2 Measurement provider

A few extra general tools have been incorporated in the new TEM. Of particular importance is the "measurement provider" tool, which allows the creation ("loading") of the measurements from the list of LHCbIDs on a track – in a one-to-one correspondence. This loading of the measurements is needed for track fitting and is typically done during the first phase of the fitting sequence.

The measurement provider tool `MeasurementProvider` inherits from the interface `IMeasurementProvider`, see figure 4.

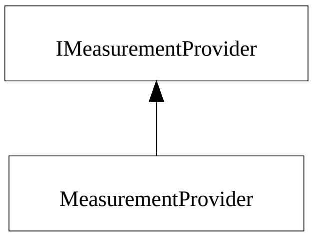

Figure 4: Inheritance diagram for the measurement provider.

All the signatures for a measurement provider are defined in the `IMeasurementProvider` interface; the complete set is presented in table 12.

virtual void load(	) = 0;
virtual StatusCode load(	Track& track) = 0;
virtual Measurement* measurement(	const LHCbID& id, double par = 0.) = 0;

Table 12: IMeasurementProvider interface signatures.

The method `void load()` needs to be called once per event to load into the provider the VELO and ST clusters and the OT times that are needed for the construction of measurements. The actual load of all the measurements on the track is done via `StatusCode load( Track& track )`. The last method listed in table 12 allows a detector-independent way of making (base class) measurements from a LHCbID; it is called from inside the `StatusCode load( Track& track )` method.

7 Access to and manipulation of data

The Gaudi framework and C++ foresee several ways of accessing the data.

7.1 Looping by index

Objects in Gaudi containers can be looped over by index. It can easily be done as follows:

```
// Get the tracks from a container
Tracks* tracks = get<Tracks>( container );

// Get (for the sake of example) the first track in the container
Tracks::const_iterator iTrack = tracks -> begin();
const Track& track = *(*iTrack);

const std::vector<LHCb::LHCbID>& allids = track.lhcbIDs();
for ( unsigned int it = 0; it < allids.size(); ++it ) {
  debug() << allids[it].detectorType();
}
```

7.2 Sequential access

Most often STL-like access is used when looping over the objects in a container. An example of a sequential access is:

```
// Get the tracks from a container
Tracks* tracks = get<Tracks>( container );

// Loop over the tracks
Tracks::const_iterator iTrack;
for( iTrack = tracks->begin(); iTrack != tracks->end(); ++iTrack ) {
  const Track* track = *iTrack;
  // add code here ...
}
```

7.3 Functors

There are a series of helper functors defined in the `TrackFunctors.h` file. They are extensively used by the tracking code – in particular in the `Track` class itself – but can also serve the end-user for common tasks such as the calculation of the number of measurements of a certain type, the removal of an element from a `std::vector`, etc.. We hereafter point to some possible applications:

- to remove the clone tracks from a list of tracks (`std::vector<Track*>& allTracks`):

```
std::vector<Track*>::iterator iCloneTracks =
  std::remove_if( allTracks.begin(), allTracks.end(),
                  TrackFunctor::HasKey<Track,const Track::Flags&>
                  ( &Track::checkFlag, Track::Clone ) );
allTracks.erase( iCloneTracks, allTracks.end() );
```

- to calculate the number of tracks of `Track::Long` type in the "best" container:

```
Tracks* tracks = get<Tracks>( TrackLocation::Default );
unsigned int nLongTracks = std::count_if( tracks->begin(), tracks->end(),
   TrackFunctor::HasKey<LHCb::Track,const Track::Types&>
   ( &Track::checkType, Track::Long ) );
```

- to remove a state (a `State*`) from a list of states associated to a track (`std::vector<State*>& states`):

```
TrackFunctor::deleteFromList<State>( states, state );
```

8 Packaging

8.1 Packages structure

- Tracking Event Model packages:
  - **Event/TrackEvent** : the core TEM classes;
  - **Kernel/LHCbKernel** : the core TEM classes that are also necessary outside the tracking in the definition of the tracking sub-detectors event models;
- Tracking Event Model packages related with fitting:
  - **Tr/TrackFitEvent** : the track fitting TEM classes;
- Common packages:
  - **Tr/TrackInterfaces** : central package for all tracking tools interfaces that do not access MC information;
  - **Tr/TrackExtrapolators** : package providing a series of track extrapolation tools;
  - **Tr/TrackTools** : collection of tracking-related tools that access reconstruction information.

9 Pythonization

A comprehensive LHCb note explains how all the classes and tools of the tracking event model have been exposed to Python [3]. It also presents examples of how to do tracking in Python and how to best use all the functionality available.

Acknowledgements:
The authors would like to thank the members of the "Track Event Model Task Force" for all their expertise brought to the review of the TEM and the careful reading of this note: O. Callot, M. Cattaneo, M. Merk, P. Koppenburg and G. Raven. We also thank E. Bos for commenting on the note.

References

[1] E. Bos, M. Merk, G. Raven, E. Rodrigues, J. van Tilburg, *The Trajectory Model for Track Fitting and Alignment*, LHCb Tracking Note LHCb 2007-008

[2] E. Rodrigues, *The LHCb Track Kalman Fit*, LHCb Tracking Note LHCb 2007-014

[3] E. Rodrigues, *Tracking in Python*, LHCb Tracking/Software Note LHCb 2006-014