LHCb Tracking Software

– A writeup to guide the design review –

Rutger van der Eijk,
Rutger Hierck,
Marcel Merk,
Matthew Needham

March 21, 2000

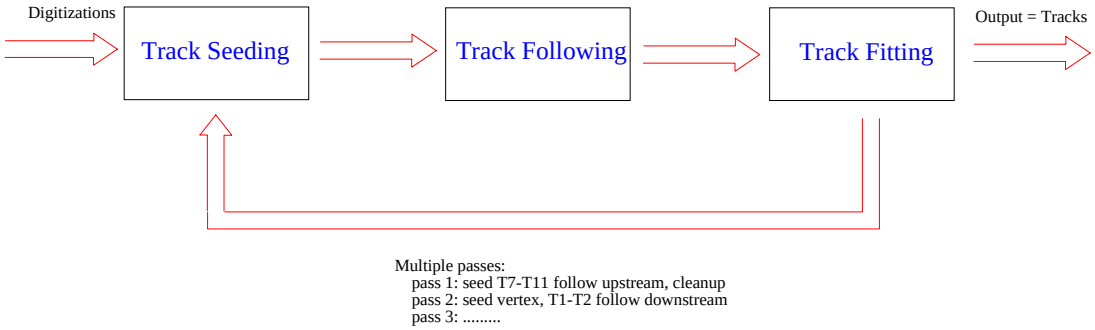

Figure 1: General overview of the track reconstruction

1 Tracking Overview

The task of track reconstruction is to reconstruct the particle trajectories from the measurements of several subdetectors. This task can be split in track finding (Pattern Recognition) and track fitting. The tracks are fitted to a track model and result in a set of track parameters and their covariances specified at several predetermined positions in the experiment (e.g. at the track vertex, at the entrance and exit points of the RICH detectors, etc.).

1.1 General procedure

Figure 1 shows the general procedure in track reconstruction.

The pattern recognition task of trackfinding is split in the subtasks of track *seeding* and track *following*. In addition the pattern recognition is executed in several passes. In the current philosophy these passes are:

pass 1: Track seeding in the field free region from station 7 to station 10 followed by upstream track following through the magnet towards the vertex detector.

pass 2: Track seeding in the vertex detector followed by downstream track-following through the magnet towards the calorimeters and muon detectors.

We foresee that pass1 and pass 2 will be followed by additional passes, trying to find more complicated trajectories, e.g. those containing kinks or electrons with hard bremsstrahlung.

Each found track is *refitted* in order to provide the best possible set of track parameters and covariances. In the refitting process information of various (non-tracking) subdetectors will be used (e.g. particle ID, particle energy).

Notice that track reconstruction uses as **input** the reconstructed subdetector data, e.g. silicon clusters, RT-calibrated drift times, particle ID. Track reconstruction is therefore of higher abstraction compared to a typical subdetector reconstruction task.

1.2 Current Implementation

In the current "released" version of the tracking code includes only realistic track refit. The pattern recognition is assumed to be ideal, i.e. it is simply taken from Monte Carlo truth. Fig 2 shows how the tracks are built and stored in the Transient Data Store. In this version the following "blobs" in the diagram are implemented:

- The *LayersOfHitsCreator* takes the the subdetector digitizations and stores these into the logical detection layers.
- The *TracksCreator* represents the ideal pattern recognition. On output the track object is a list of hits that are assigned to it.
- The *FitInitializer* provides a first estimate of the track parameters. Once realistic pattern recognition is implemented, the result of that will provide the input to the parameters x, y, t_x, t_y. The initial estimate of the momentum is obtained using the so-called p_t kick of the magnet.
- The *TrackFitter* performs the actual track fit.

In the optimal fit the input of almost all LHCb subdetectors is required.

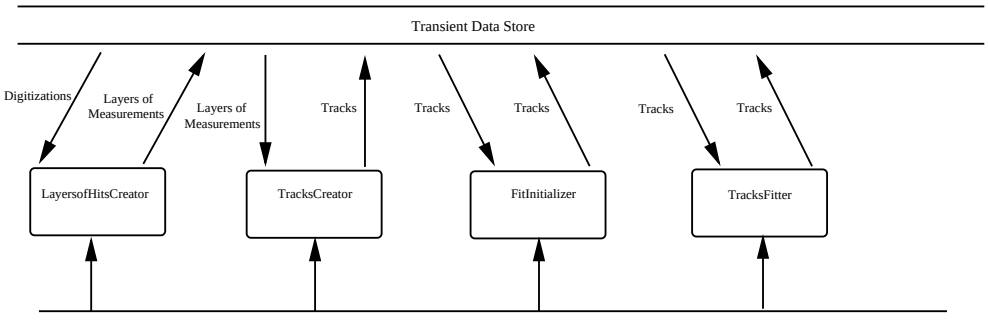

Figure 2: Current Implementation of the Fitting Code.

2 Tracking Concepts

Within track reconstruction several concepts are frequently used. Below the physics of a few important concepts are briefly explained. From a computing point of view they can be identified as *tools*, i.e. global algorithms.

2.1 Kalman Filter

The Kalman filter is a general mathematical technique to update the knowledge of a system *state* with *measurements* as they become available. The technique identifies two steps, *prediction* and *filtering*. The prediction step describes how the state evolves in time (space). The filtering step describes how to update of the best estimate the state with a new measurement.

Identifying the track parameters of a particle trajectory as the state, we can use the kalman filter technique as a progressive track fit. The hits in the detectors are the measurements. The *extrapolation* of the track parameters to a new measurement position can be seen as the prediction step. Furthermore updating the track parameters with the hits can be identified as the filtering step.

The kalman filter is equivalent to a least squares fit. Advantages over a global fit are: multiple scattering can elegantly be taken into account, it is fast, and it can be used in pattern recognition.

2.2 Extrapolation

Extrapolation (or "transport") is the prediction of track parameters (including covariance matrix) at a certain position along a trajectory from another position.

In the extrapolation the effects of an inhomogeneous magnetic field, of multiple scattering and of energy loss are appropriately taken into account. The extrapolation tool relies on the *navigation* tool to retrieve information.

2.3 Hit Clustering

Hit clustering is the process of grouping hits together that share a common property. One obvious common property of all hits created by a passing particle is that they are close to the particle trajectory. Hence the distance to a (predicted) track will be roughly the same for all hits on that track. We can use this fact to find hits belonging to a track.

2.4 Navigation

Several algorithms need quick access to information. This information can be subdetector event data or information from a datebase.

Specific examples are:

- To predict the trajectory of a particle through the magnet there must be fast access to the magnetic field value at a given position in space.
- In the extrapolation from position A to position B a trajectory might intersect with one or more material objects. The intersections and the thickness (in X_0) of the material must be accessed in a navigation tool.
- To assign candidate hits to tracks in pattern recognition, fast access is needed to subdetector data ("hits") inside a geometrical region of interest (e.g. $x \pm \delta x$, $y \pm \delta y$, $z \pm \delta z$)

In general what is needed is some mechanism to *navigate* through the experiment. Navigation is a key concept in track reconstruction. Frequent use of navigation tools makes track reconstruction one of the main consumers of CPU time. Therefore the navigation tools have to be fast.

3 Algorithm Description

As background information to this design review a short summary of the algorithms is given. The track seeding and trackfollowing serve to provide optimal assignment of hits to tracks (pattern recognition), while the refit serves to provide the best possible track parameters. Although the separate algorithms share tools described in the previous section, they might be applied in a course (fast) or precise (slow) way to provide the desired result.

To provide an example of how the tools in the previous section are used one of the algorithms (trackfollowing) is described in more detail.

3.1 Track Seeding

The first part of pattern recognition searches for segments of tracks which can be used as a seed to the track following algorithm. In pass 1 it finds tracksegments in the (almost) free field region from station T7 to T10.

Currently two algorithms are under development. Both initially create a 2-dimensional seed in the precision (i.e. bending) plane and subsequently a 3-dimensional seed by adding information of the stereo measurements.

- The first algorithm creates line segments inside a station, and subsequentially links those to form the track seeds.
- The second algorithm considers all hits in the finding stations and applies a global clustering method to create the seeds.

3.2 Track Following

The second part of pattern recognition starts from the track seeds at a given z position and follows the track from station to station, either in the upstream or downstream direction. While doing so the track parameters are progressively updated with increasing precision.

In some more detail, the following procedure is used:

- From the seed position extrapolate the track to the nearest station of interest.
- Open a search window (*Region of Interest* (RoI)), the size of which depends on the precision of the extrapolated track parameters.
- In the station, select all hits which are inside this RoI. Fast access is needed to these data using the navigation tool.

- Calculate the signed 3D distance (*residual*) from the measurement to the extrapolated track trajectory. Put the hits in the RoI in a vector, sorted in increasing distance parameter.
- Apply a cluster algorithm, grouping hits with alike distance. Each cluster forms a candidate for track continuation. Selection parameters considered are:
  - *size* of the cluster: i.e. the number of hits on the track continuation.
  - *width* of the cluster: i.e. compatibility of the hits to belong to one track extrapolation.
  - *distance* of the cluster: i.e. average distance of the hits to the track extrapolation.

  The exact numerical value of the selection parameters depends on the hit quality and track extrapolation precision.
- The hits of each clustered are filtered into the track using the Kalman procedure.
- In case more than one acceptable track continuation candidate is found the track following procedure will *branch* and all trajectories will be followed. Each track candidate has a quality defined based on the following
  - number of hits on the track
  - number of *faults* (i.e.: hits that are missed) on the track.
  - the χ^2 residuals of the hits

  Based on its quality track candidates can be defined dead or alive.

3.3 Track (re-)Fitting

The pattern recognition provides already fitted tracks. However a more detailed, and thus slower, refit procedure is needed to find the optimal track parameters and errors.
Typically track extrapolations will be more precise (slower) as compared to the pattern recognition algorithms. In addition, separate fits might be used for different particle ID's. (e.g. electrons)

4 Data Model

4.1 Objects

We can identify the following data objects:

- Digitizations (e.g. classes *OTDigi, ITDigi*) — 'Raw' Data from the detector, i.e. uncalibrated TDC counts.
- Hits (e.g. classes *OTHits, ITHits*) — Hits are the output of the subdetector reconstruction and hence the input for the track reconstruction. Hits know both about a digitization and the corresponding detector properties (geometry, calibration, alignment....).
- Measurements assigned to tracks (classes *OTHitOnTrack, ITHitOnTrack*) — Knows about a hit but also contains information that is only relevant because a hit is on a given track (e.g. solution of drift ambiguity in case of outer tracker and χ^2 contribution of hit to track). Each hit-on-track class is derived from the TrMeasurement abstract base class which specifies all information a measurement needs to be filtered.
- Layer of hits (classes *TrOTLayer, TrITLayer*) — Container of hits. In the data store we store an ObjectVector of TrOTLayers and one of TrITLayers for the outer and inner trackers respectively.
- Detection cell (classes *OTDetectionCell, ITDetectionCell*) — A detection cell is the detector element (e.g. wire and strip) that performed a measurement. It provides geometry, alignment and calibration information.
- Detection layer (classes *OTDetectionLayer, ITDetectionLayer*) — Provides geometry/alignment information for a given layer. It is a container of detection cells.
- Track State (class *TrState*) — A snapshot of track. Track parameters and covariance matrix at a given z position on the track trajectory. Some examples are: (x, y, t_x, t_y), $(x, y, t_x, t_y, Q/p_t)$, $(x, y, t_x, t_y, Q/p)$.
- Track (class *TrTrack*) — Container of information determined about the track. As the track reconstruction proceeds the information in the container. At the moment the track finally consists of:
  - List of pointers to measurements (TrMeasurements)

  - List of pointers to track states
  - Track charge
  - Track χ^2
  - Particle Type

  Notice that in order to fit electrons correctly the particle type needs to be supplied from somewhere. At the moment we cheat from the Monte Carlo truth. In the future this should come from RICH/calorimeter information.

- TrNode — Temporary container class for all information needed to do smoothing (HitOnTrack, transport matrix, track states)

4.2 Data model from the viewpoint of the Hits

Fig. 3 illustrates the data model from the point of view of the hits. A hit is an aggregation of a digitization and a detection cell. Knowledge of the detection cell allows the digitization to be aligned and calibrated. The hits are then stored by layer in TrOTLayers and TrITLayers for the outer and inner tracker respectively. At the moment to allow access to the Monte Carlo truth a derived OT/IT MCHit class is used. Notice that since in the case of the inner tracker clustering of digitizations is likely to be done the Monte Carlo truth information related to the digitizations and hits may be different. We **strongly** believe that it is essential to have a pointer from the hit directly to the MCTrackingHit rather than to the MCParticle as at present.

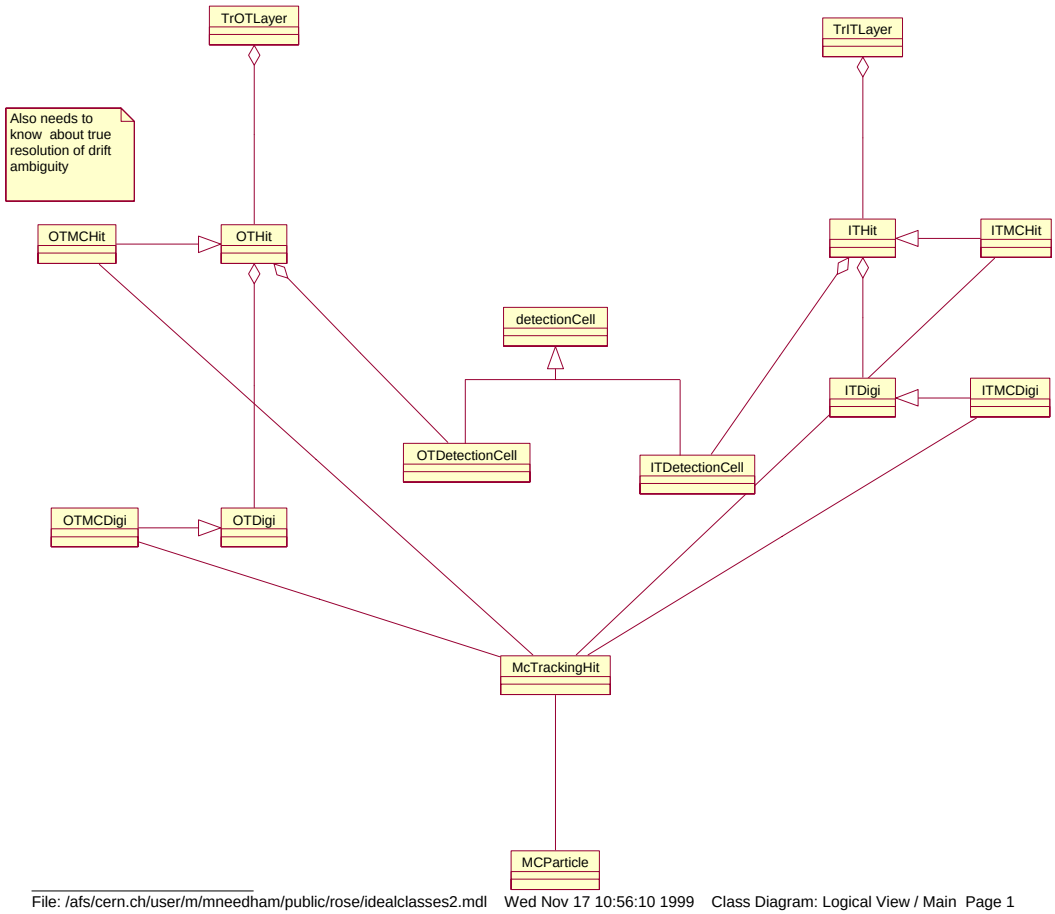

Figure 3: Class Diagram from the point of view of the Hits.

4.3 Data model from the viewpoint of a Track

Fig 4 shows the data model from the point of view of the tracks. All tracks are derived from the interface class TrTrack. One concrete implementation of a track is a TrMCTrack track which has a pointer to the Monte Carlo truth. Each track is an aggregation of states (track parameters at a given z) and measurements which in this case means things that can be fitted using the Kalman filter. Concrete implementations of the measurements are the IT/OT HitOnTrack classes.

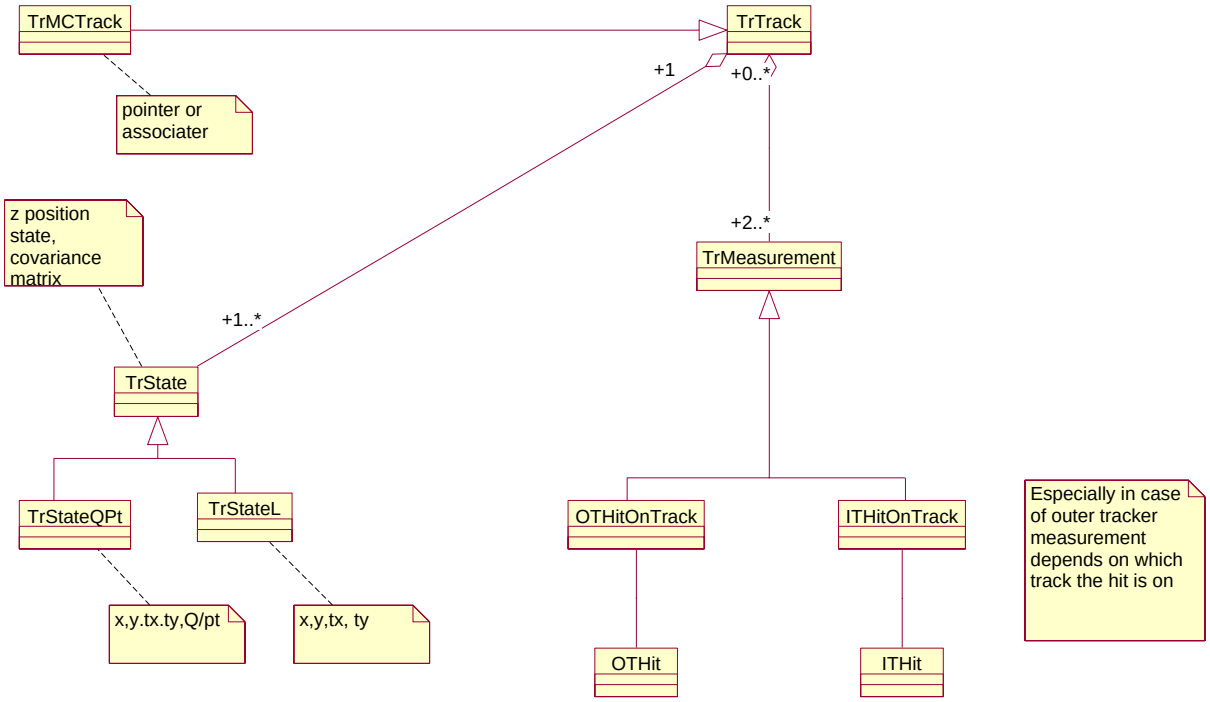

Figure 4: Class Diagram from the point of view of the Tracks.

5 Some Design Choices

Now we will discuss some problems we have encountered and the solutions we have chosen.

5.1 Track States

A track consists of a list of hits and a state. In our data model we have several types of states (see Section 4). Also different extrapolators can be used; for example linear, Runge-Kutta, parabolic and linear. Both the concrete extrapolator and concrete state classes have an interface base-class (resp. TrExtrapolator and TrState). This allows to implement a track reconstruction algorithm in terms of these interfaces without explicit knowledge of the concrete states and extrapolators used.
In order for this to work we use the so called "visitor" (or double dispatch) design pattern. This works as follows (see Fig. 5):

- Each state has an extrapolate member with a extrapolator as an argument. The state will choose the apropriate execute member of the extrapolator. ("The extrapolator visits the state")
- This is fine as long as the number of states is small. For every new state N new execute members must be written (with N number of implemented extrapolators)

A similar situation occurs when we want to project the state into the measurement space and a similar solution is chosen.

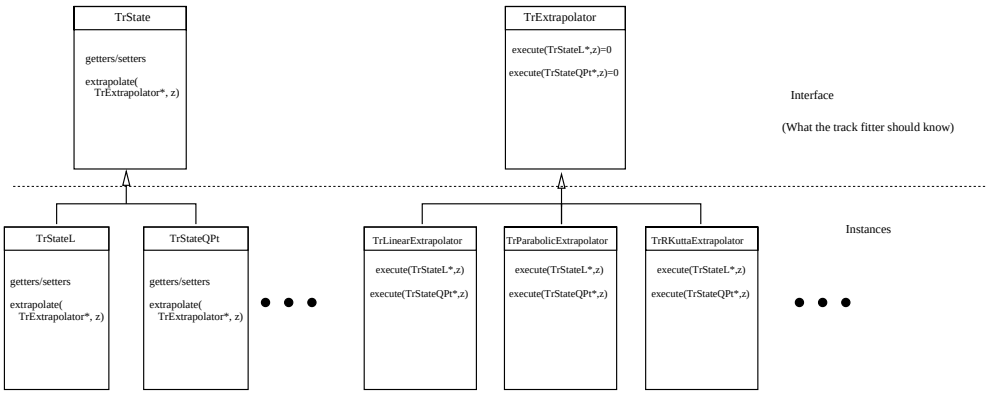

Figure 5: Visitor pattern.

5.2 Dealing with MC

For the performance of the tracking code to be understood and monitored fast and easy access to the Monte Carlo truth information is essential. For our hits and digitizations we have implemented this by deriving a MCHit from a normal hit class. This class has the same properties as a normal hit but in addition has a pointer to the Monte Carlo truth. We have done a similar thing in the case of the Tracks. Obviously this type of solution works well when the relationship between data and Monte Carlo is one-to-one. With full pattern recognition there will be no one-to-one correspondence between the reconstructed tracks and Monte Carlo particles. Therefore, at least in this case a different solution needs to be found. For example an associator or a SmartReferenceTable. We have considered using SmartReferenceTables but have found the syntax somewhat daunting!

From the point of view of the tracking since we have inputs from many sub-detectors a common detector solution to dealing with Monte Carlo truth is highly desirable.

6 Some problems encountered

In this section we discuss some of the remaining software problems we have.

6.1 Serialization of ObjectVectors

For persistancy reasons ObjectVectors now have a serialize member. For this member to work if you have an ObjectVector of (for example) TrTrack you must provide a default constructer for the TrTrack class. In our case this is not possible since the TrTrack is an interface class. To overcome this problem Gaudi needs to have a dictionary like in ROOT. At the moment to 'solve' this problem for our private use we have simply removed the serialize from the ObjectVector class.

6.2 Tools

In our different algorithms, seeding, following, fitting, we use some common 'subalgorithms' like the Extrapolator and Kalman filter. At the moment for each algorithm that uses the Kalman filter algorithm we make a new instance of the subalgorithm. This does not make sense — why are two instances of a class needed with the same functionality? A more natural implementation of this would be a 'tool service' where you have one instance of the subalgorithm. In our opinion a general concept of tools (i.e. global algorithms) in the Gaudi framework is needed. Recently there has been discussion on implementing a "Transport Service", which we consider an improvement.

6.3 Separation of data and algorithm

The separation of data and algorithm in the Gaudi framework has caused us some problems and complications — mainly related to sorting. Two examples are given below.

6.3.1 Quality Factor

In our cluster algorithm for the trackseeding/trackfollowing we define a quality factor for the 'goodness' of the cluster. There are several types of quality factors which we want to try. A quality factor has several properties (see section 3), which depend on, for instance, which station you are, what "type" of cluster you have (Inner or Outer Tracker). In your track following you want

to sort your clusters on quality factor (for example using the standard STL sort algorithms). So you would want to do something like:

```
sort(clusterList.begin(),clusterList.end(),sortByQualityFactorOne())
```

However, this implies that one property of your cluster is the ability to calculate this quality factor, but this implies the cluster knows more 'algorithm'-like details like what station it is in. So how do we proceed. Do we sort the list ourselves ? Are we doing something wrong ?

6.3.2 Ordering of hits in a track

Simply ordering by z does not work in the case of curling or steep tracks. This means a track cannot sort itself (as it would need algorithm related things like extrapolators). Therefore the only reasonable procedure to follow is to say that the relative order of hits determined by the pattern recognition is final and can not changed (though of course you are allowed to order in increasing or decreasing z). This then has other implications — for instance the container of hits should then remain hidden to the user to prevent them from ever trying to sort the container.

A related issue is that in the patten recognition we intend to seed tracks in both the up and downstream directions. For the re-fit all tracks will be fitted upstream. Should a track know if it was found in an upstream or downstream pass ? Note, upstream and downstream fitting *is* different because of energy loss via $\frac{dE}{dx}$.

7 Future Plans

A first version of the track fitting software exists in SICb providing the same functionality as in SICb. At present this code is not available in the standard public release of Gaudi. With the growing number of users (at present 7) we would like to make this code publically available as soon as possible. This requires some work in stupid things like making sure our histogram IDs, class IDs and where we put things in the Transient Data Store does not cause clashes with other subdetector code. We are also currently working on two notes (a developers and userguide) to document how to use the code.

For the fit to have completely the same functionality as SICb the use of Velo information in the fit needs to be implemented. With the data model we have this presents no technical difficulties once both the Velo data and geometry model exists.

Also at present we use a wrapped version of the SICb transport. In the near future we hope to replace the bulk of this FORTRAN with new C++ routines. Again, we foresee few difficulties in doing this in the framework of the current design. One complication is that at present the SICb transport makes use of a database containing a simplified description of the material geometry. When **all** detectors have a OO description of the database this database can be replaced with the tool provided in the transport service. Until then this part of code must remain wrapped FORTRAN.

The track seeding studies are at a very early stage. The actual algorithm (or algorithms) to be used are at a very developmental stage. Work will continue in this direction over the next few months.

The track following studies are at a much more advanced stage. Algorithms exist that starting from a seed (cheated using the Monte Carlo truth) allow us to attempt to follow the track through the detector and try to find track continuations (clusters). Obviously realistic track following brings new challenges and problems. For example you need to decide what you should do if you get two or more possible continuations from one track. At the moment we envisage that we keep the best N continuations from one seed in a container. Then at the end of the track fitting we simply select the best candidate one. Again in the coming months the track following studies will continue.