Pattern - High Level Tools for Data Analysis

Version 0-30

HERA-B Note 02-002, Software 02-002

Thorsten Glebe
Max-Planck-Institut für Kernphysik, Heidelberg
T.Glebe@mpi-hd.mpg.de

January 18, 2002

Abstract

The `Pattern` project provides high-level tools for an object-oriented analysis of high-energy physics data. The `Pattern` concept allows to conveniently define signatures for decays and provides mechanisms to reconstruct and investigate decay candidates in real and Monte Carlo data. The `Pattern` project is part of the data analysis framework `BEE` [1].

Contents

1 Introduction **2**

2 Concepts and Ingredients **3**
2.1 A basic example . . . 3
2.2 Pattern Types . . . 5
2.3 Defining Track and Decay Pattern . . . 5
2.4 GTopp . . . 10

3 Track Pattern **11**
3.1 Track Classification . . . 11
3.2 More on Track Pattern . . . 12
3.3 Pattern Matches . . . 13

4 Decay Pattern **15**
4.1 Reconstructing decays . . . 15
4.2 Decay Pattern Matches . . . 21
4.3 A more complex example . . . 22
4.4 Storing Mother Tracks as RecoSegment . . . 26
4.5 Neutral Particles . . . 27

5 Monte Carlo **29**

1 Introduction

One of the main tasks of data analysis software in high energy physics experiments is the reconstruction and investigation of decays and decay chains of short living particles. A lot of information from different reconstruction algorithms (e.g. reconstructed primary vertices, particle identification, momentum determination...) must be combined in order to identify track combinations which have a common origin and belong to the decay of a particle candidate. The reconstructed decayed particle can be itself a decay product in a complex decay chain and used as an input for further decay reconstruction.

From the software point of view, reconstructing decay chains works always in a similar way. First tracks are classified according to their measured particle id (if possible), then selected tracks are combined and it is checked whether they form a vertex (or it is just checked whether the invariant mass is in the expected range). If yes, the tracks are accepted as a particle candidate if certain predefined cuts are fulfilled (invariant mass cut, displacement cut etc.) and histograms or ntuples are filled with the information of the found decay candidates. Altogether, analysis code of this kind involves a lot of programming of recurring tasks which can be implemented once as tools within an analysis framework (static part, see figure 1).

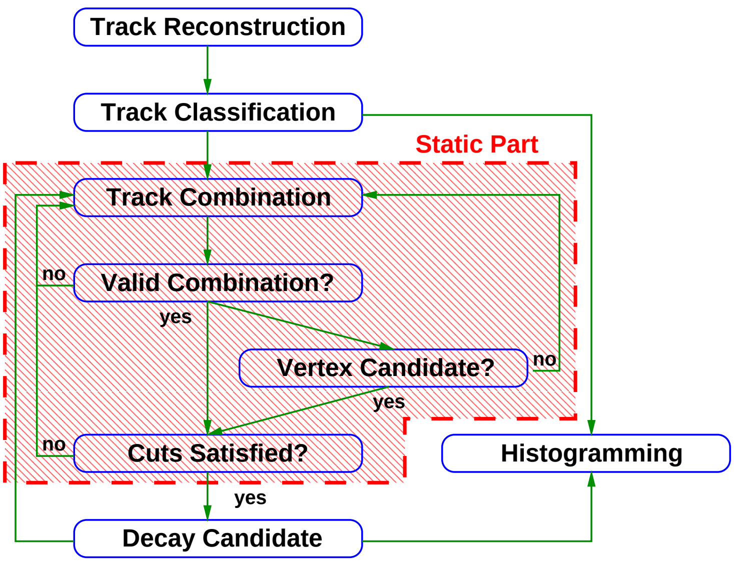


Figure 1: Decay chain reconstruction flow.

The `Pattern` project provides a framwork in which particle hypothesises and decay chains of arbitrary complexity can be defined. Standard code for track selection and decay chain reconstruction is provided and the vertexing is based for performance reasons on the very fast `SVertex` class [2]. In addition to this, the framework allows to assign one or more particle hypotheses to tracks and vertices. The framework provides a lot of bookkeeping such that the user can easily trace back to which particle hypotheses and vertex objects a reconstructed track was assigned during an analysis.

The advantage of such a framework is that standard tasks are programmed once and reproducable results can be achieved within this framework. The user does not have to program his own bookkeeping routines and does not have to bother about the exact syntax and usage of the vertexing package. The effort of programming an analysis job is very much reduced for the physicist without restricting the analysis, as it is always assured that access to all information obtained by the analysis is possible.

The `Pattern` project is based on the abstractions defined in the `BEE interfaces` project and can thus work with any track class inherited from the `TrackIf` interface [3].

Note: The code examples of this documentation are simplified for educational reasons because they are inteded to demonstrate the features of this software project. For realistic examples look at the code examples provided with the `BEE doc` project.

2 Concepts and Ingredients

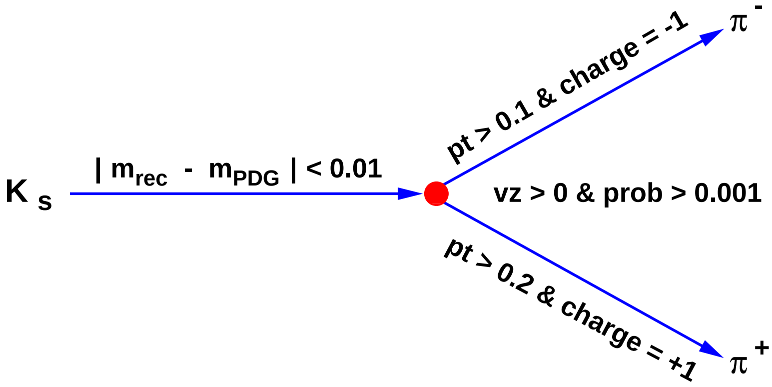
2.1 A basic example

In order to describe the way the `Pattern` framework works, lets start with an example, e.g. the reconstruction of K_S^0 (K-short) particles decaying via the process $K_S^0 \to \pi^- \pi^+$. Before starting the reconstruction, the physicist might have already an idea about basic cuts which he wants to apply on tracks and vertices, e.g. cuts on kinematic quantities of the measured tracks or cuts on tracks and vertices implied by the geometry and layout of the detector system.

The kind of decay (2-prong, 3-prong...) under investigation plus the particle types (π, K...) plus the cuts to be applied on the tracks (e.g. on transverse momentum: $p_t > 0.1$ GeV/c...) and vertices (e.g. on z position of vertex, $v_z > 0$ cm, or on the difference of the reconstructed invariant mass m_{rec} and the expected mass m_{PDG} [4]: $|m_{rec} - m_{PDG}|$) form a signature (or pattern) of a decaying particle as shown in figure 2(a) for which the physicist wants to search the data.

The above described particle signature can be represented in an object oriented way by the `Pattern` framework as shown in figure 2(b): The user defines two objects `Piminus` and `Piplus` which define the properties of the π^- and π^+ candidate tracks to be selected (the type of these objects has been omitted for sake of simplicity, for more details see section 2.2). Then one can use these objects to define an object `kshort` which represents the decay $K_S^0 \to \pi^- \pi^+$. The `kshort` object can be used to define cuts on various parameters of the vertex and the reconstructed track of the incoming particle.

Once the Track and Decay Pattern are defined, the `Pattern` project provides methods to extract the π^- and π^+ candidates from the list of reconstructed tracks, e.g. in the following way:



(a) Particle decay with demanded properties.

```
Piminus = pi_minus.with(pt > 0.1);
Piplus  = pi_plus.with(pt > 0.2);

kshort  = K_S.decaysTo(Piminus & Piplus);
kshort->with(vz > 0. & prob > 0.001);
kshort->massrange(0.01);
```

(b) Decay represented by object-oriented code.

Figure 2: A decay and it's object-oriented representation, see also figure 4.

```
Piplus->select(ArteTable<RecoTrack>());
Piminus->select(ArteTable<RecoTrack>());
```

This will select all `RecoTrack` objects from the `ArteTable` which match the defined properties.

Then, based on the result of the above selected π^- and π^+ candidates we can search for the combinations of tracks which form a vertex and match the defined K_S^0 properties (e.g. invariant mass) by issuing a single command:

```
kshort->combine();
```

The result, that means the list of all K_S^0 candidates matching the defined criteria, is available by using the `kshort` object. The list of selected π^- and π^+ candidates is available via the `Piminus` and the `Piplus` object (see section 3.3). Relations between π^+, π^- and K_S^0 candidates are set up for an in-depth investigation of the found decay candidates.

2.2 Pattern Types

The example of figure 2 shows that two types of pattern exist in the `Pattern` project:

Track Pattern represents a signature for measured (reconstructed) and Monte Carlo tracks (e.g. the `Piminus` and `Piplus` objects in figure 2(b)).

Decay Pattern represents the signature of a decaying particle (`kshort` object in figure 2(b)). It is assumed that for the decaying particle no measured track exists, thus it is reconstructed from the decay products. The reconstructed incoming track is not a separate object but is also represented by the vertex object (see [3], figure 2).

In the `Pattern` project, Track Pattern are represented by objects of type `TrackPattern`. This type is a `SmartPointer` type [6], defined via a `typedef` statement in the `P_trait.hh` header file of the `Pattern` project[1]. Smart pointers can be used as ordinary pointers and assure that only one instance of the defined Track and Decay Pattern objects exists, which improves the code performance and consistency.

For Decay Pattern the type of the pattern objects depends for performance reasons on the number of tracks in the vertex. These types are pointer types, too. The following types are predefined for Decay Pattern:

```
TwoProngDecay    // type representing 2-prong decays
ThreeProngDecay  // type representing 3-prong decays
FourProngDecay   // type representing 4-prong decays
FiveProngDecay   // type representing 5-prong decays
SixProngDecay    // type representing 6-prong decays
```

The following example shows the definition of a Decay Pattern for a 2-prong and a 3-prong decay:

```
TwoProngDecay   kshort;   // e.g. K0 -> pi+ pi-
ThreeProngDecay Dplus;    // e.g. D+ -> K- pi+ pi+
```

In the general case the type of a n-prong vertex is: `DP_trait<n>::DPMC_sp`, with n being the number of tracks in the vertex (for more information on the traits technique, see [5]).

2.3 Defining Track and Decay Pattern

Both Track and Decay Pattern are defined via global constant objects of type `GTopp` (see figure 3). The `GTopp` class represents constant properties of known particles, most of them taken from the Particle Data Group (PDG) tables [4]. A `GTopp` object represents the particle hypothesis assigned to the tracks of the pattern. See section 2.4 for more details on `GTopp` objects. In the reference manual a list of defined global `GTopp` objects can be found. The `with()` member function allows to define a cut expression for the selection of track objects. A pattern expression engine

[1]The exact definition is:
`typedef SmartPointer<SPatternMatchContainer> TrackPattern;`

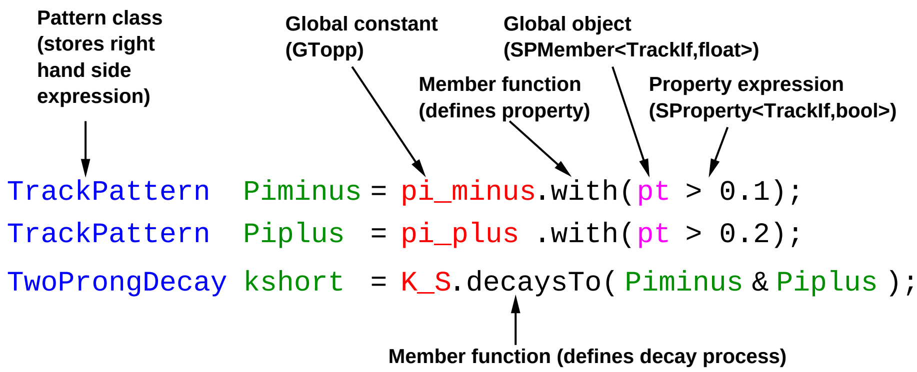


Figure 3: Structure of Track and Decay Pattern definitions.

has been developed which is able to parse, store and apply complex cut expressions which are built from global property objects (e.g `pt`, see figure 3). These global property objects are defined by the `SProperty` class hierarchy and work for all classes inherited from the `TrackIf` and `VertexIf` interface [3].

Table 4(a) shows the property objects which are defined for track selection expressions and table 4(b) shows the property objects which are defined for vertex selection expressions. With those objects the user can build complex cut expressions (see example below). For technical reasons these property objects are defined in the `Pattern` project (files `TrackIfPattern.C` and `VertexIfPattern.C`). The invention of new global property objects is straightforward.

Note: The following property objects are able to take one argument (the rest-mass m_0 of the particle): `energy`, `xf` and `rap`. If the argument is omitted, a rest-mass $m_0 = 0$ is assumed. The `SProperty` class hierarchy allows to define property objects which can take up to two arguments[2]

Note: The property objects of table 4(a) and 4(b) cannot be mixed within an expression!

In case a Track Pattern should be created without any user defined property, the `createPattern()` member function must be used:

```
TrackPattern Piplus = pi_plus.createPattern();
```

Note: The charge of the track defined by the particle hypothesis is always added to the user defined property, even if no user property is defined (when the `createPattern()` member function was used). The following example shows the properties which are assigned in case of the two different Track Pattern creation methods:

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);
TrackPattern Piplus  = pi_plus.createPattern();

cout << "Piminus: " << *Piminus->pattern() << endl;
cout << "Pplus: "   << *Piplus->pattern() << endl;
```

[2] In the current `Pattern` version no global property object with two arguments is defined.

property object	accessor function	description
x	TrackIf::x()	track x position
y	TrackIf::y()	track y position
z	TrackIf::z()	track z position
tx	TrackIf::tx()	track slope t_x
ty	TrackIf::ty()	track slope t_y
p	TrackIf::p()	track momentum p
pt	TrackIf::pt()	transverse track momentum p_t
pz	TrackIf::pz()	track momentum along z
chi2	TrackIf::chi2()	χ^2 of track fit
ndf	TrackIf::ndf()	degrees of freedom of track fit
azim	TrackIf::phi()	azimuthal angle ϕ
theta	TrackIf::theta()	polar angle θ
psrap	TrackIf::eta()	pseudorapidity η
charge	TrackIf::charge()	track charge
energy	TrackIf::energy()	track energy $E(m_0)$
xf	TrackIf::xf()	Feynman variable $x_F(m_0)$
rap	TrackIf::rap()	Rapidity $y(m_0)$
isValid	TrackIf::isValid()	track validity flag
all	`true`	always true

(a) Property objects for track selection (m_0 = particle rest mass, see page 6 for more details).

property object	accessor function	description
vx	VertexIf::vx()	vertex x position
vy	VertexIf::vy()	vertex y position
vz	VertexIf::vz()	vertex z position
vchi2	VertexIf::chi2()	χ^2 of vertex fit
prob	VertexIf::prob()	upper tail χ^2 probability
vndf	VertexIf::ndf()	degrees of freedom of vertex fit

(b) Property objects for vertex selection.

Figure 4: Global property objects for track and vertex selection.

The output is:

```
Piminus: pi-, ((charge == -1) & (pt > 0.10))
Pplus: pi+, (charge == 1)
```

It is possible to use round brackets `()` to specify the evaluation order of the expression, which is useful in complex expressions. The following

operator	description
$<$	is less than
$>$	is greater than
$==$	is equal to
$!=$	is not equal to
&	logical AND
|	logical OR
$+$	mathematical addition
$-$	mathematical subtraction
$*$	mathematical multiplication
$/$	mathematical division

Table 1: List of defined operators in property expressions.

example illustrates this:

```
const double pimass = pi_minus.mass();
TrackPattern Piminus = pi_minus.with(pt > 0.1 |
                       (xf(pimass) > -0.2 & xf(pimass) < 0.2) |
                        energy(pimass) > 1.5);

cout << "Piminus: " << *Piminus->pattern() << endl;
```

In this example a couple of operators have been used. For a list of defined operators see table 1.

The output is:

```
Piminus: pi-, ((charge == -1) & (((pt > 0.10) |
    ((xf(0.13957) > -0.20) & (xf(0.13957) < 0.20))) |
    (energy(0.13957) > 1.50)))
```

The vertex property can be defined via the `with()` member function of the Decay Pattern object (see example in figure 2(b)). It is always checked whether the invariant mass of the tracks in the vertex is in a certain mass window around the expected mass (which is defined by the particle hypothesis of the Decay Pattern). The radius of the mass window can be defined via the `massrange()` member function (see example in figure 2(b)).

Note: In case the radius of the mass window is not redefined by the user, a default radius of **0.1 GeV** is used.

In addition to the vertex property the track property of the incoming mother particle reconstructed from the decay products can be defined via the `with()` member function:

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);
TrackPattern Piplus  = pi_plus.with(pt > 0.2);

TwoProngDecay kshort = K_S.decaysTo(Piplus & Piminus);
kshort.with(vz > 0);    // vertex property
kshort.with(pt > 0.1); // track property

cout << "kshort: "  << *kshort->pattern() << endl;
```

The output is:

```
kshort: K_S -> pi+ + pi-
      pi+, ((charge == 1) & (pt > 0.20))
      pi-, ((charge == -1) & (pt > 0.10))
K_S, ((charge == 0) & (pt > 0.10))
vertex property: (vz > 0.00)
```

When a track property is defined for a Decay Pattern, the incoming track will be reconstructed automatically so that the property can be checked:

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);
TrackPattern Piplus  = pi_plus.with(pt > 0.2);

TwoProngDecay kshort = K_S.decaysTo(Piplus & Piminus);
kshort.with(vz > 0);   // vertex property
kshort.with(pt > 0.1); // track property

... // select pion tracks

// Vertex reconstruction using Kalman filter method,
// computation of mother track because kshort object
// has a track property defined:
kshort->combine();
```

However, the covariance matrix of the mother track will not be computed automatically, as this is not required by the defined property objects. In order to get the covariance matrix of the K_S^0 tracks the user has to compute them manually:

```
... (same code as in above example)

// combine, vertex & calculate mother track
kshort->combine();
// calculate covariance matrix of mother track
kshort->calc_mother_cov();
```

A more detailed description of possible ways to select candidate tracks can be found in section 3.1.

In case no track property was defined for a Decay Pattern, the mother track will not be reconstructed by the `Pattern` framework[3]. In this case the user can compute the mother track manually:

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);
TrackPattern Piplus  = pi_plus.with(pt > 0.2);

TwoProngDecay kshort = K_S.decaysTo(Piplus & Piminus);
kshort.with(vz > 0);   // vertex property

... // select pion tracks

// combine tracks, fit vertex with Kalman filter but do not
```

[3]The reason for this is: performance. The framework computes only the minimal amount of information required and avoids thus unnecessary overhead.

```
// compute mother track, as no track property is defined
kshort->combine();
// compute mother tracks and covariance matrix manually
kshort->calc_mother();
```

In case the Kalman filter was used for the vertex fit, the `calc_mother()` function computes both the track parameters and the covariance matrix of the mother track using the refitted tracks from the Kalman filter. Both track parameters and covariance matrix can be computed separately by the `calc_mother_tr()` and the `calc_mother_cov()` functions.

In case of a topological vertex fit (e.g. by the `combine_no_vertex()` method) no refit of the outgoing tracks takes place, therefore the track parameters of the mother track can only be computed using the track parameters of the decay products. This is done automatically by the `calc_mother_tr()` function. A call to the `calc_mother_cov()` routine in this situation would not result in the computation of the covariance matrix, the function would simply return `false` to signal the failure of this attempt.

In section 4.2 the access to the found decay candidates is explained.

2.4 GTopp

Both Track and Decay Pattern must be defined via a `GTopp` object. The `GTopp` class is defined within the `clue` project and is closely related to the `Topp` class, which represents constant particle properties [7]. The `GTopp` class contains the methods `createPattern()`, `with()`, and `decaysTo()` to create Track and Decay Pattern.

For each entry of the `Topp` table a global `GTopp` object exist which is nothing else than a named reference to the `Topp` table entry. The following example shows the `GTopp` object representing J/Ψ meson properties:

```
// global object which represents J/Psi constants
cout << Jpsi << endl;
```

Note: The user can only make use of `GTopp` objects if the `Topp` table is present. This is the case when you load a `BEE`-file which contains the `Topp` table.

In case no `Topp` table is present, an error will occur:

```
// example: no BEE file loaded -> no Topp table available
cout << Jpsi << endl;
```

The output is:

```
No Topp link available!
```

But with adding a file to the `TChain:`[4]

```
f->Add("rp17137.root")
// Note: it is important that an event was loaded!
f->GetEntry(0)
cout << Jpsi << endl;
```

The output is:

[4] For more information on how to set up ROOT see [9] or [10].

```
[295] Jpsi  mass: 3.097e+00, life: 7.480e-21, width: 8.800e-05,
chrg:   0, colo:  0, anti: [295]
```

Note: A list of all defined global `GTopp` objects can be found in the reference manual.

3 Track Pattern

3.1 Track Classification

Once the `TrackPattern` objects are defined, they can be used to select from a given list those tracks which match the properties defined by the pattern. The tracks selected by the Track Pattern are in this way classified by the pattern. A classified track is stored as a **Pattern Match** object (see section 3.3 for more details on Pattern Matches).

In case of `BEE` tracks will be most likely selected from the table of reconstructed tracks (`ArteTable<RecoTrack>`) or the table of MC truth tracks (`ArteTable<MC_Track>`) or, in case of photons, from the table of calorimeter clusters (`ArteTable<EcalHit>`). Those tables are filled by the HERA-B reconstruction framework ARTE [11, 12] and represent the information provided by the standard HERA-B event reconstruction software.

The `SPatternMatchContainer::select()` method allows to select tracks according to the defined pattern from a list (e.g. an `ArteTable`) given as a parameter to the function:

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);

Piminus->select(ArteTable<RecoTrack>());

cout << "No of Piminus matches: " << Piminus->size() << endl;
cout << "List of selected Tracks: " << endl;
cout << Piminus << endl;
```

The output looks e.g. like this:

```
No of Piminus matches: 1
List of selected tracks:
Track:  tx: -4.415e-02 ty:  2.627e-04 x:  -2.54e+00 y:   1.08e-01
z:   6.26e+01 p:   7.00e+00 chi2:   3.66e+01 charge: -1
Cmp: vxd patt magt bit16 clonerm wch    Mom: 2 RecoVertex: [1]
Pattern: pi-, ((charge == -1) & (pt > 0.10))
```

As one can see, the pattern is printed out together with the track which was matched to the pattern.

Note: The `select()` function works on any STL-compatible [13] list which stores a pointers to track objects. Therefore `select()` can also be used to select tracks from containers which are prepared and filled by the user instead of `ArteTables`.

Another possibility is to select the tracks manually. This is the preferred method in case the selection criteria cannot be represented by a simple cut expression. The `TrackPattern` objects have an `STL` container functionality and track objects can be added via the `push_back()` method:

```
// code example for manual track selection
TrackPattern Piminus = pi_minus.createPattern();
TrackPattern Piplus  = pi_plus.createPattern();

typedef ArteTableIterator<RecoTrack> TrackIt;

const double pimass = pi_plus.mass();
const TrackIt end = ArteTable<RecoTrack>::end();
for(TrackIt it = ArteTable<RecoTrack>::begin();
    it != end; ++it) {
  const RecoTrack& rtra = **it;

  // cuts on the PID, the number of hits and physical
  // quantities
  if((rtra.pid() != 0 && rtra.pid() != PID::pion) ||
     rtra.hsi() < 6                                ||
    (rtra.hou() + rtra.hin() < 16)                 ||
     rtra.pt() < 0.1 || rtra.p() < 2.              ||
     fabs(rtra.xf(pimass)) > 0.2)
    continue;

  const int chrg = rtra.charge();

  if(chrg > 0) { Piplus->push_back(rtra);  }
  if(chrg < 0) { Piminus->push_back(rtra); }
} // for RecoTrack
```

3.2 More on Track Pattern

When Track Pattern have been defined, one can use them to check whether tracks which match those pattern exist in a certain container. This can be e.g. useful in an event selection routine where only those events are accepted where a certain pattern is present. For this reason the `SPatternMatchContainer::exist()` method is provided. This method checks if at least one track which matches the pattern exists in the track container given as an argument:

```
TrackPattern Piplus = pi_plus.with(pt > 0.15);

// event loop
for(...) {
  // skip event if no Piplus match is present in ArteTable
  if( !Piplus->exist(ArteTable<RecoTrack>()) )
    continue;
  ...
}
```

Note: similar to the `select()` method the container given as an argument must be STL compatible and the objects stored in the container must be pointer to track objects.

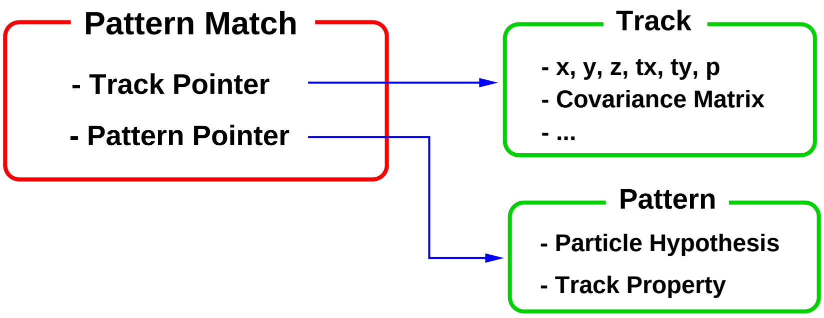


Figure 5: Structure of a Pattern Match object.

As shown in section 3.1, `TrackPattern` objects are containers for matched tracks. In case the `push_back()` method is used to fill the matched tracks manually into the `TrackPattern` container, one has to take care that the container is cleared at the beginning of a new event. Otherwise the results might be inconsistent or the program might crash. To clear the container, the `clear()` method is provided:

```
TrackPattern Piplus = pi_plus.with(pt > 0.15);

// event loop
for(...) {
  Piplus->clear();
  ...
  Piplus->push_back(...);
}
```

Note: in case the `select()` method is used to fill the `TrackPattern` container it is **not** necessary to clear the container manually. The `select()` function will clear the container automatically before filling in new pattern matches.

3.3 Pattern Matches

A track which has been selected by a `TrackPattern` object is called **Pattern Match**. The pointer to the track object is stored in an object of type `SPatternMatch`. Thus `TrackPattern` objects inclose containers for smart pointers to `SPatternMatch` objects. The `SPatternMatch` class is a composite of a pointer to the track object and a pointer to the pattern object which was used to select the track (see figure 5). Therefore we are able to do a bookkeeping of tracks and the properties associated to them. The `SPatternMatch` class is based on the `TrackIf` interface and behaves within `BEE` like any other track object. As a result of this we can e.g. use `SPatternMatch` objects for vertexing with the `SVertex` class.

Note: At the moment the `SPatternMatch` class contains code specialized for `RecoSegment` and `MC_Track` objects[5], thus the `Pattern` project is

[5]For example a relation between `SPatternMatch` and `RecoSegment` is provided.

not completly independent from Clue [7].

Note: A typedef allows to use the type `ParticleCandidate` instead of `SmartPointer<SPatternMatch>` in order to arrive at more readable user code.

The following example shows how to loop over the Pattern Matches stored in a `TrackPattern` object:

```
TrackPattern Piplus = pi_plus.with(pt > 0.2);
Piplus->select(ArteTable<RecoTrack>());

// Loop over PatternMatches
const ParticleCandidateConstIterator end =
                                   Piplus->matches().end();
for(ParticleCandidateConstIterator it =
    Piplus->matches().begin(); it != end; ++it) {
  const ParticleCandidate pm = *it;

  cout << "pattern: " << pm->getPattern() << endl;
  if(pm->isMCmatch())
    cout << "MC Track: ";
  else
    cout << "Reco Track: ";
  cout << pm->getTrackIf() << endl;
}
```

In case of `RecoTrack` and `RecoSegment` objects a One-to-Many relation is created to the `SPatternMatch` objects. In this way one can check to which pattern a track was associated. The relations are implemented using the `GRelation` project, which is part of `BEE` [8]. The following example shows how the relation to `SPatternMatch` objects is used to find all Pattern Matches of the `RecoTrack` objects:

```
Piplus = pi_plus.with(pt > 0.2);
Kplus  = K_plus.with(pt > 0.2 & pz > 1.);

Piplus->select(ArteTable<RecoTrack>());
Kplus->select(ArteTable<RecoTrack>());

typedef ArteTableIterator<RecoTrack> TrackIt;

const double pimass = pi_plus.mass();
const TrackIt end = ArteTable<RecoTrack>::end();
for(TrackIt it = ArteTable<RecoTrack>::begin();
    it != end; ++it) {
  const ArtePointer<RecoTrack> rtra = *it;

  // loop over relation to Pattern Matches
  typedef GRelationIterator<SPatternMatch> RelIt;
  const RelIt rend = rtra->pmatch.end();
  for(RelIt rit = rtra->pmatch.begin();
      rit != rend; ++rit) {
    const ParticleCandidate pm = **rit;
```

```
      cout << "RecoTrack " << rtra
           << " is matched to " << pm->getPattern().getName()
           << endl;
    }
  }
```

The output will look like this:

```
RecoTrack [1] is matched to pi+
RecoTrack [1] is matched to K+
RecoTrack [3] is matched to pi-
RecoTrack [4] is matched to pi-
```

Via this relation it is possible to find out to which decay chains a reconstructed track was assigned.

It is feasible to store an arbitrary amount of additional information together with a `SPatternMatch` object inside a `float` array. In this way one can associate information which was not forseen within the `Pattern` framework:

```
  TrackPattern Piplus = pi_plus.with(pt > 0.2);
  Piplus->select(ArteTable<RecoTrack>());

  // Loop over PatternMatches
  const ParticleCandidateConstIterator end =
                                   Piplus->matches().end();
  for(ParticleCandidateConstIterator it =
      Piplus->matches().begin(); it != end; ++it) {
    ParticleCandidate pm = *it;

    pm->set_user(2); // allocate user field of length=2
    float* user = pm->get_user(); // get pointer to array
    user[0] = ...; // store some value
    user[1] = ...; // store some value
  }
```

Deallocation of the user field is done automatically by the `ParticleCandidate` destructor.

4 Decay Pattern

The reconstruction of decay candidates can take place after the possible daughter tracks have been selected via the Track Pattern which have been used to define the Decay Pattern. In the following sections several methods to reconstruct decayed particles are described and it is explained how the results can be investigated.

4.1 Reconstructing decays

In order to reconstruct a decayed particle a track combination must be found which is likely to originate from a common vertex. To find such combinations, three methods are provided which make use of different information.

The first method checks only whether the invariant mass of the track combination is in the region of the expected decayed particle. This method uses a minimal amount of information (only the track momenta) and is thus the least biased method. The disadvantage is that a lot of combinatorial background can be expected. The following example shows how this method is used to reconstruct K_S^0 candidates via the decay $K_S^0 \to \pi^+\pi^-$:

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);
TrackPattern Piplus  = pi_plus.with(pt > 0.2);

// Decay Pattern without a property!
TwoProngDecay kshort = K_S.decaysTo(Piplus & Piminus);

Piminus->select(ArteTable<RecoTrack>());
Piplus->select(ArteTable<RecoTrack>());

// combine tracks, do not compute vertex, just check
// invariant mass
kshort->combine_no_vertex();

// print out all matches
cout << kshort << endl;
```

As one can see, the `combine_no_vertex()` method has to be called and the Decay Pattern must not have any property defined (otherwise the second method is applied, see below). The output will look like this (some output has been discarded for sake of clarity):

```
=================> Begin of K_S object
K_S -> pi+ + pi-
      pi+, ((charge == 1) & (pt > 0.20))
      pi-, ((charge == -1) & (pt > 0.10))
K_S, (charge == 0)
massrange: 0.10, rest-masses: 0.1396, 0.1396
SVertex:: no vertex computed
invariant mass: 5.2528e-01
===> Outgoing Tracks:
Track:  tx: -9.913e-03 ty: -4.864e-02 ... (output discarded)
Track:  tx: -4.982e-02 ty:  5.890e-02 ... (output discarded)
=================> End of K_S object
=================> Begin of K_S object
... (output discarded)
```

In the second method a vertex is computed in an analytical way[6] without using the errors of the track parameters. A track combination is accepted as a decay candidate if the invariant mass of the track combination is in the allowed region, the vertex could be calculated and the properties of the vertex fullfills the required cuts. This method can be applied

[6]The function `SVertex::calcVertex2D()` is used [2].

by using again the `combine_no_vertex()` method, but now with a vertex property defined for the Decay Pattern. The property will be checked automatically for each computed vertex and only those track combinations which passed the property check will be stored as a candidate for a particle decay:

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);
TrackPattern Piplus  = pi_plus.with(pt > 0.2);

// Decay Pattern with vertex property!
TwoProngDecay kshort = K_S.decaysTo(Piplus & Piminus);
kshort->with(vz > 0.);

Piminus->select(ArteTable<RecoTrack>());
Piplus->select(ArteTable<RecoTrack>());

// combine tracks, compute vertex via 2D topological method
kshort->combine_no_vertex();

// print out all matches
cout << kshort << endl;
```

The output will look like this:

```
=================> Begin of K_S object
K_S -> pi+ + pi-
      pi+, ((charge == 1) & (pt > 0.20))
      pi-, ((charge == -1) & (pt > 0.10))
K_S, (charge == 0)
vertex property: (vz > 0.00)
massrange: 0.10, rest-masses: 0.1396, 0.1396
SVertex::  vx: 9.031e-01 vy: -1.327e-01 vz: 2.358e+00
invariant mass: 4.798e-01
===> Outgoing Tracks:
Track:  tx:  2.415e-02 ty:  2.185e-02 ... (output discarded)
Track:  tx:  9.830e-02 ty:  6.581e-02 ... (output discarded
=================> End of K_S object
=================> Begin of K_S object
... (output discarded)
```

If a track property is defined for the Decay Pattern, a mother track is computed. In case of the `combine_no_vertex()` method this implies also the computation of the vertex, even if no vertex property is defined (otherwise the set of track parameters would be incomplete):

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);
TrackPattern Piplus  = pi_plus.with(pt > 0.2);

// Decay Pattern with track property, but without vertex
// property!
TwoProngDecay kshort = K_S.decaysTo(Piplus & Piminus);
kshort->with(pt > 0.);
```

```
  Piminus->select(ArteTable<RecoTrack>());
  Piplus->select(ArteTable<RecoTrack>());

  // combine tracks, compute vertex via 2D topological method
  // and compute mother track
  kshort->combine_no_vertex();

  // print out all matches
  cout << kshort << endl;
```

The output would be like this:

```
=================> Begin of K_S object
K_S -> pi+ + pi-
      pi+, ((charge == 1) & (pt > 0.20))
      pi-, ((charge == -1) & (pt > 0.10))
K_S, ((charge == 0) & (pt > 0.10))
massrange: 0.10, rest-masses: 0.1396, 0.1396
SVertex::  vx: 1.824e-01 vy: 9.410e-02 vz: 1.575e+01
Mother Track: x: 1.824e-01 y: 9.410e-02 z: 1.575e+01 tx: 8.106e-03
ty: -3.139e-02 p: 3.655e+01
invariant mass: 5.466e-01
===> Outgoing Tracks:
Track:  tx: -9.913e-03 ty: -4.864e-02 ... (output discarded)
Track:  tx:  1.126e-02 ty: -2.837e-02 ... (output discarded)
=================> End of K_S object
=================> Begin of K_S object
... (output discarded)
```

In the third method the vertex is fitted based on a Kalman filter, which includes the information from the covariance matrices of the daugther tracks. Before the Kalman filter can be applied, the tracks must have been prepared, that means the track parameters together with the covariance matrix must have been propagated to $z = 0$ and the inverse covariance matrix must have been calculated. To apply the Kalman filter based vertex fit, the `combine()` method has to be used. In case of `RecoTrack` objects the function

```
  RtEvent::PrepareRecoTracksForVertexing()
```

can be used to prepare all `RecoTrack` objects which are part of the table `ArteTable<RecoTrack>`. The following example shows again the reconstruction of K_S^0 candidates via the decay $K_S^0 \to \pi^+\pi^-$, but now using the Kalman filter based vertex fit which makes use of a maximum of information from the daughter tracks. A track combination is accepted in case the invariant mass of the tracks is in the expected region, the Kalman filter based vertex fit succeeded and the vertex (and eventually track) properties are fulfilled:

```
  TrackPattern Piminus = pi_minus.with(pt > 0.1);
  TrackPattern Piplus  = pi_plus.with(pt > 0.2);

  // Decay Pattern with vertex property!
  TwoProngDecay kshort = K_S.decaysTo(Piplus & Piminus);
```

```
  kshort.with(vz > 0);   // vertex property

  // prepare all tracks in ArteTable<RecoTrack>
  event->PrepareRecoTracksForVertexing();

  Piminus->select(ArteTable<RecoTrack>());
  Piplus->select(ArteTable<RecoTrack>());

  // combine tracks, compute vertex via Kalman filter method
  // and check vertex property
  kshort->combine();

  // print out all matches
  cout << kshort << endl;
```

The output would look like the following:

```
=================> Begin of K_S object
K_S -> pi+ + pi-
      pi+, ((charge == 1) & (pt > 0.20))
      pi-, ((charge == -1) & (pt > 0.10))
K_S, (charge == 0)
vertex property: (vz > 0.00)
massrange: 0.10, rest-masses: 0.1396, 0.1396
SVertex::  vx: -4.895e-01 vy: -2.765e-01 vz: 3.382e+01
chi2: 4.753e-01 ndf: 1
Cov. Matrix:
[    2.798e-05   2.875e-07  -4.489e-04
     2.875e-07   1.564e-04   4.551e-04
    -4.489e-04   4.551e-04   4.022e-02 ]
Kalman Information:
 ktx: -9.896e-03 kty: -4.860e-02      ...(output discarded)
 ktx: -4.999e-02 kty: 5.884e-02       ...(output discarded)
invariant mass: 5.251e-01
===> Outgoing Tracks:
Track:  tx: -9.913e-03 ty: -4.864e-02 ...(output discarded)
Track:  tx: -4.982e-02 ty:  5.890e-02 ...(output discarded)
=================> End of K_S object
...(output discarded)
```

The Kalman informations shown in the output are the refitted track parameters and covariance matrices. These are used e.g. to compute the invariant mass.

The `RecoSegment::PrepareForVertexing()` routine can be used to prepare `RecoTrack` objects for vertexing if the tracks are selected manually. This will be more performant than the above method, because one can avoid the effort of propagating tracks which are not used in Track Pattern objects:

```
  // code example for manual track selection
  TrackPattern Piminus = pi_minus.createPattern();
  TrackPattern Piplus  = pi_plus.createPattern();
```

```
TwoProngDecay Kshort = K_S.decaysTo(Piminus & Piplus);

// select RecoTrack objects manually
typedef ArteTableIterator<RecoTrack> TrackIt;

const double pimass = pi_plus.mass();
const TrackIt end = ArteTable<RecoTrack>::end();
for(TrackIt it = ArteTable<RecoTrack>::begin();
    it != end; ++it) {

  RecoTrack& rtra = **it;

  // cuts on the PID, the number of hits and physical
  // quantities
  if((rtra.pid() != 0 && rtra.pid() != PID::pion) ||
     rtra.hsi() < 6                               ||
    (rtra.hou() + rtra.hin() < 16)                ||
     rtra.pt() < 0.1 || rtra.p() < 2.             ||
     fabs(rtra.xf(pimass)) > 0.2)
    continue;

  // prepare only those tracks for Kalman filter based
  // vertexing which will end up in a Track Pattern
  // container
  rtra.PrepareForVertexing();

  const int chrg = rtra.charge();

  if(chrg > 0) { Piplus->push_back(rtra);  }
  if(chrg < 0) { Piminus->push_back(rtra); }
} // for RecoTrack

// reconstruct K-short decays using Kalman filter based
// vertex fit
Kshort->combine();
```

Note: When the `combine_no_vertex()` method is applied, it is **not** needed to prepare the tracks for vertexing, as no information from the track covariance matrices is used.

Note: If a property for the mother track is specified, then the `combine()` method automatically computes the mother track parameters, but not the covariance matrix. The Kalman filter based determination of the mother track covariance matrix can be done manually afterwards:

```
kshort->combine();
// compute covariance matrix of K_S tracks (only
// in case of Kalman filter based vertex fit!)
kshort->calc_mother_cov();
```

A situation where the track parameters together with the covariance matrix are computed automatically by the `Pattern` framework is shown in section 4.3.

Note: In case of more complex decay chains it might happen that a reconstructed track appears more than once in a decay chain (e.g. because it is part of different TrackPattern containers or tracks from the same container are used in different places of the decay chain, see e.g. section 4.3). For physical reasons this is not a valid combination and such combinations should be rejected by the code reconstructing the decay chain. A consistency check which assures that every track in a decay chain is used only once can be enabled by the user by passing a `true` value to the `combine()` and `combine_no_vertex()` methods:

```
// do consistency check for each track combination:
kshort->combine(true);
// or
kshort->combine_no_vertex(true);
```

For performance reasons the user should avoid this check whenever possible. In the above examples the check can most likely be omitted because under normal conditions a reconstructed track cannot be classified as a π^+ and a π^- candidate at the same time.

4.2 Decay Pattern Matches

A reconstructed decay candidate is called a **Decay Pattern Match**. These objects are implemented via the `SDecayPatternMatch` template class, which is based on the `DecayPatternMatchIf` interface defined in the `BEE interfaces` project [3]. The following typedefs are provided for readability by the `P_trait.hh` header file:

```
TwoProngCandidate   // type representing 2-prong decay
ThreeProngCandidate // type representing 3-prong decay
FourProngCandidate  // type representing 4-prong decay
FiveProngCandidate  // type representing 5-prong decay
SixProngCandidate   // type representing 6-prong decay
TwoProngCandidateIterator          // iterator
TwoProngCandidateConstIterator     // const iterator
ThreeProngCandidateIterator        // iterator
ThreeProngCandidateConstIterator   // const iterator
FourProngCandidateIterator         // iterator
FourProngCandidateConstIterator    // const iterator
FiveProngCandidateIterator         // iterator
FiveProngCandidateConstIterator    // const iterator
SixProngCandidateIterator          // iterator
SixProngCandidateConstIterator     // const iterator
```

In the general case the type of a n-prong decay candidate iterator is: `DP_trait<`n`>::SDPM_it`, with n being the number of tracks in the vertex.

The following code shows how one can loop over the reconstructed decay candidates using the above mentioned iterators:

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);
TrackPattern Piplus  = pi_plus.with(pt > 0.2);
```

```
TwoProngDecay kshort = K_S.decaysTo(Piplus & Piminus);
kshort->with(pt > 0.);

Piminus->select(ArteTable<RecoTrack>());
Piplus->select(ArteTable<RecoTrack>());

// combine tracks, compute vertex via topological method
// and compute mother track (to check kshort pt-property)
kshort->combine_no_vertex();

const TwoProngCandidateConstIterator end =
                                     kshort->matches().end();
for(TwoProngCandidateConstIterator it =
    kshort->matches().begin(); it != end; ++it){
    const TwoProngCandidate cand = *it;

    cout << "mass: "          << cand->mass()           << endl;
    cout << "mother track: " << cand->mother()          << endl;
    cout << "track1: "       << *cand->PatternMatch(0) << endl;
    cout << "track2: "       << *cand->PatternMatch(1) << endl;
}
```

Note: It is possible to store additional user defined information in a user field analogously to the `SPatternMatch` class (see section 3.3).

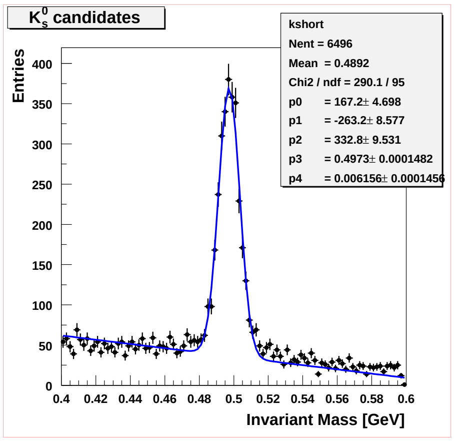
The `get()` member function can be used to obtain `PatternMatchIf` pointer to daugther tracks via the particle hypothesis by using the global `GTopp` objects. In the above example one can e.g. write:

```
... // see above example

const TwoProngCandidateConstIterator end =
                                     kshort->matches().end();
for(TwoProngCandidateConstIterator it =
    kshort->matches().begin(); it != end; ++it){
    const TwoProngCandidate cand = *it;

    // get the positive pion
    const PatternMatchIf& pipl = *cand->get(pi_plus);
    cout << "pi plus: " << pipl << endl;
}
```

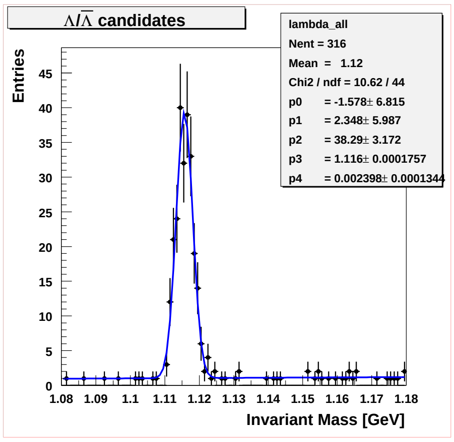
Figure 6 shows signals which have been reconstructed using data from the HERA-B year 2000 data taking period (single lepton trigger) using `BEE` and the Track and Decay Pattern framework.

4.3 A more complex example

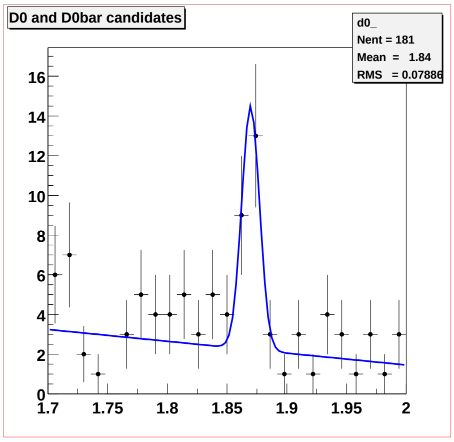
In the following a more complex example is described: We want to reconstruct the decay $D^+ \to \bar{K}^0 \pi^+$ with the anti-kaon decaying to $\bar{K}^0 \to \pi^+ \pi^-$. We do not want to use the Kalman filter based vertex fit (e.g. because we do not trust the errors of the track parameters) but only a topological vertex fit. From the decay chain itself it is clear that π^+



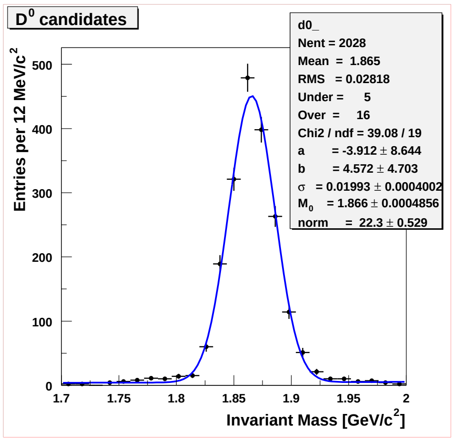
(a) Reconstructed $K_S^0 \to \pi^+\pi^-$ decays.



(b) Reconstructed $\Lambda \to p\pi^-$ and $\bar{\Lambda} \to \bar{p}\pi^+$ decays.



(c) Reconstructed $D^0 \to K^-\pi^+$ and $\bar{D}^0 \to K^+\pi^-$ decays.



(d) Reconstructed $D^0 \to K^-\pi^+$ decays selected via MC truth.

Figure 6: Decays reconstructed from HERA-B year 2000 data using Track and Decay Pattern. Figure (d) is produced from reconstructed Monte Carlo data.

tracks are used in two different places, thus we must check that the same π^+ track is not used twice in the decay chain. For educational reasons we use in the following only very simplistic cuts in the Track and Decay Pattern objects.

```
// Pattern definitions
```

```
TrackPattern Piminus = pi_minus.with(pt > 0.1);
TrackPattern Piplus  = pi_plus.with(pt > 0.2);

TwoProngDecay kzero = K_bar.decaysTo(Piminus & Piplus);
kzero->with(vz > 0.);
kzero->with(pt > 0.1);

TwoProngDecay dplus = D_plus.decaysTo(kzero & Piplus);

// event loop
for(...) {
  // clear track container (not necessary for decay container)
  Piplus->clear();
  Piminus->clear();

  // fill track container with selected tracks
  Piplus->select(ArteTable<RecoTrack>());
  Piminus->select(ArteTable<RecoTrack>());

  // reconstruct kzero candidates
  kzero->combine_no_vertex();

  // reconstruct Dplus candidates & check for same tracks
  // in decay chain
  dplus->combine_no_vertex(true);

  // look at reconstruction result
  const TwoProngCandidateConstIterator end =
                              dplus->matches().end();
  for(TwoProngCandidateConstIterator it =
      dplus->matches().begin(); it != end; ++it) {
    const TwoProngCandidate dpm = *it;

    cout << dpm << endl;

    // look at K_bar daughter tracks which are part
    // of a Dplus candidate
    const DecayPatternMatchIf& kbar =
    static_cast<const DecayPatternMatchIf&>(*dpm->get(K_bar));

    cout << "kbar track 0: " << *kbar.track(0) << endl
         << "kbar track 1: " << *kbar.track(1) << endl;
  }

  ... (continued in next example...)
```

Note: The `Pattern` framework takes care that all necessary information is available when the D^+ candidate is reconstructed, e.g. the $\bar{K}^0$ track is computed automatically if it is not already available. This is also the case if the `combine()` method is used. Then the $\bar{K}^0$ mother track including the covariance matrix would be computed automatically by the `dplus->combine()` statement.

The above example shows how one can navigate through the decay chain starting from the top of the chain. In case of decaying daughter tracks (e.g. the $\bar{K}^0$ in the above example) it is necessary to cast the `PatternMatchIf` pointer into a pointer of type `DecayPatternMatchIf` in order to access the $\bar{K}^0$ decay products. The `get()` function allows to easily access certain tracks via the global defined `GTopp` objects (`K_bar` in the above example).

Another possibility to look at decay chains is to use the `TrackPattern` container and iterate over the relations of Track Pattern objects to Decay Pattern objects in order to find out in which decay chains the Pattern Matches (i.e. the selected tracks) are used. In this way we navigate through the decay chain starting from the bottom, the decay products. In the following example we look at the π^+ candidates which occur in the D^+ decay chain:

```
  ... (see previous example)

  // loop over pi_plus candidates in the Piplus container
  const ParticleCandidateConstIterator tend =
                                  Piplus->matches().end();
  for(ParticleCandidateConstIterator tit =
    Piplus->matches().begin(); tit != tend; ++tit) {
    const ParticleCandidate pm = *tit;

    if(pm->vtx.size() == 0) continue;

    // loop over relations to decayed particles
    typedef ConstGRelationIterator<DecayPatternMatchIf> RelIt;

    const RelIt rend = pm->vtx.end();
    for(RelIt rit = pm->vtx.begin(); rit != rend; ++rit) {
      const DecayPatternMatchIf& cand = **rit;

      if(cand == D_plus)
        cout << "pi_plus from D_plus: " << pm << endl;
      else {
        // Does K_bar originate from D_plus?
        const RelIt rrend = cand.Vtx().end();
        for(RelIt rrit = cand.Vtx().begin(); rrit != rrend;
        ++rrit) {
          if(**rrit == D_plus)
          cout << "pi_plus from K_bar: " << pm << endl;
        }
      }
    }
  }
} // event loop
```

4.4 Storing Mother Tracks as RecoSegment

A reconstructed particle is represented by a `SDecayPatternMatch` object. In this way the full information of the decay chain is accessible. However it should be noted that it is not possible to use a `SDecayPatternMatch` object in another event[7], e.g. for event mixing purposes.

The `SDecayPatternMatch::createMotherSegment()` method provides a possibility to store the track parameters of the reconstructed particle into a `RecoSegment` object by omitting all information about the vertex and the reconstructed decay chain. This allows to use the reconstructed particles e.g. for event mixing in another event. The `createMotherSegment()` method is a factory function which creates a `RecoSegment` object on the heap and returns the pointer to it. Therefore the user has to take care for the destruction of the `RecoSegment` object, otherwise this will result in a memory leak. In case the `RecoSegment` object is again fed into a `TrackPattern` container, the Pattern Match object can take over the ownership of the `RecoSegment` object and destruct it when the Pattern Match object destructor is called. Thus the user does not have to care about deleting these `RecoSegment` pointers:

```
// pattern for K-short reconstruction
TrackPattern  Piminus    = pi_minus.with(pt > 0.1);
TrackPattern  Piplus     = pi_plus.with(pt > 0.2);
TwoProngDecay kzero = K_S.decaysTo(Piminus & Piplus);

// Pattern for K-short event mixing
TrackPattern kzero_mother = K_S.createPattern();

// event loop
for(...) {

  ...(fill Piminus and Piplus container)
  kzero->combine(); // reconstruct K_S vertices

  // loop ueber K_S Kandidaten
  const TwoProngCandidateIterator end = kzero->matches().end();
  for(TwoProngCandidateIterator it = kzero->matches().begin();
      it != end; ++it) {
    const TwoProngCandidate cand = *it;

    // create mother track as RecoSegment object on heap
    RecoSegment* rs = cand->createMotherSegment();
    cout << "RecoSegment: " << rs << ", " << *rs << endl;

    // fill RecoSegment in container, which takes over ownership
    // of object (signaled by second argument)
    kzero_mother->push_back(*rs, true);
  }
```

[7] The reason is that links to other objects in the `ArteTable`'s exist, thus inconsistencies or segmentation faults could appear.

```
}
```

4.5 Neutral Particles

In HERA-B, photons are reconstructed by clusters in the electromagnetic calorimeter, the ECAL. In `BEE`, the `EcalHit` class, which is part of the `clue` project, is forseen to represent photon candidates. However, in contrast to a reconstructed track, an `EcalHit` does not represent a track but just the impact point at the ECAL. A track can only be defined if the origin of the photon is given. In case of HERA-B the photon origin is either the primary vertex or any other decay vertex from which photons can emerge.

In the `Pattern` project it is possible to define Decay Pattern which involve photons, and the track properties of the photons together with the covariance matrix can be computed from the `EcalHit` objects by specifying a photon origin together with the errors of the origin. This information must be passed in the form of a `VertexIf` inherited class or a 6-dimensional vector to the `EcalHit::calc_track()` function:

```
TrackPattern Photon = photon.with(pt > 0.3);

// use the 1st wire as photon origin
const ArtePointer<Wire> w(1);
// prepare vector with information about origin
SVector<double,6> origin;
origin.place_at(w->xvec(),0);
origin.place_at(w->dxvec(),3);

// fill container with photon candidates
typedef ArteTableIterator<EcalHit> EcalIt;

const EcalIt end = ArteTable<EcalHit>::end();
for(EcalIt it = ArteTable<EcalHit>::begin(); it != end; ++it) {
  const ArtePointer<EcalHit> ehit = *it;

  // calculate track assuming wire as origin
  ehit->calc_track(origin);

  // check properties
  if(Photon->evalTrackProp(*ehit) == true) {

    // needed only in case of Kalman filter based vertex fit
    ehit->PrepareForVertexing();

    Photon->push_back(*ehit);
  }
}
```

Note: In case no photon origin was given, the point $(0, 0, 0)$ will be used as the origin, together with an error $\sigma = 50$cm on each coordinate.

The following example shows how the photon pattern can be used to reconstruct the decay $\pi^0 \to \gamma\,\gamma$:

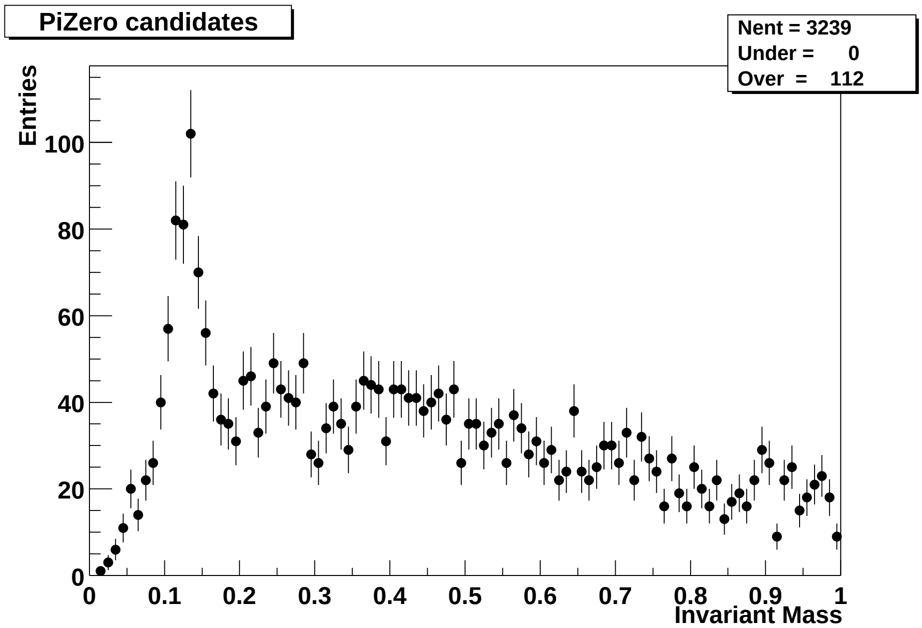


Figure 7: π^0 signal reconstructed using a Decay Pattern.

```
TrackPattern  Photon = photon.with(pt > 0.3);
TwoProngDecay pizero = pi_zero.decaysTo(Photon & Photon);

// ... fill Photon container (see above)

pizero->combine_no_vertex();
```

Note: In the above example the `Pattern` code detects automatically that tracks from the same container are combined and chooses the appropriate combination scheme. Therefore it is not necessary to check for invalid combinations (where the same track appeares more than once) as shown in section 4.3. This works even in case of more complicated Decay Pattern like $X \to ABB$, $X \to AABB$, $X \to AAABB$, $X \to ABCC\ldots$

Note: It is of course possible to generate Decay Pattern which contain one or more photons together with charged tracks. In case the decay vertices will be fitted with the Kalman filter (that means the `combine()` method is used) one has to call first (similar to charged tracks) the `EcalHit::PrepareForVertexing()` method for photon candidates before the vertex fit is performed (see example above).

Figure 7 shows a π^0 signal which was reconstructed in the HERA-B year 2000 data using the above described Track and Decay Pattern.

It is also possible to use the vertex of a reconstructed decay as the origin of the photon, which might be interesting in case of decays where a photon is involved, e.g. in the decay $\chi_c \to J/\psi\,\gamma$ (with $J/\psi \to \mu^+\mu^-$):

```
TrackPattern Myplus  = mu_plus.with(pt > 0.1);
```

```
TrackPattern Myminus = mu_minus.with(pt > 0.1);
TrackPattern Photon  = photon.createPattern();

// pattern for prompt JPsi's
TwoProngDecay JPsi = Jpsi.decaysTo(Muplus & Muminus);
JPsi->with(vz > - 6 & vz < -3.);
TwoProngDecay Chic = chi_c0.decaysTo(JPsi & Photon);

// fill Muons
Myplus->select(ArteTable<RecoTrack>());
Myminus->select(ArteTable<RecoTrack>());

// take primary vertex as origin
const PrimaryVertex& pvtx = ...; // get it from somewhere

// loop over photon candidates, set origin to primary vertex
typedef ArteTableIterator<EcalHit> EcalIt;
const EcalIt end = ArteTable<EcalHit>::end();
for(EcalIt it = ArteTable<EcalHit>::begin(); it != end; ++it) {
  const ArtePointer<EcalHit> ehit = *it;

  // take vertex position and errors to calculate photon track
  ehit->calc_track(pvtx);
  Photon->push_back(ehit);
}

// reconstruct prompt J/Psi candidates
JPsi->combine();

// reconstruct chi_c candidates, just look at invariant mass
Chic->combine_no_vertex();
```

5 Monte Carlo

The Pattern framework can not only be applied to reconstructed data, but also on Monte Carlo (MC) truth data. The MC truth tracks are represented by the MC_Track class and the MC truth vertices by the MC_Vertex class of the clue project. The following example shows how one can search for a specific decay (in this example $\phi \to K^+ K^-$) in the MC truth data by using the selectMCtruth() method. The result of the search is then accessible in the ususal way through the Decay Pattern object:

```
TrackPattern KMinus  = K_minus.createPattern();
TrackPattern KPlus   = K_plus.createPattern();

TwoProngDecay Phi = phi.decaysTo(KMinus & KPlus);

Phi->selectMCtruth(); // collect all MC truth decays

cout << Phi << endl;
```

The output will look like this:

```
=================> Begin of phi object
phi -> K+ + K-
      K+, (charge == 1)
      K-, (charge == -1)
phi, (charge == 0)
massrange: 0.10, rest-masses: 0.4937, 0.4937
SVertex:: no vertex computed
Mother Track: x: 2.253e-01 y: 2.011e-01 z: -5.586e+00 tx: 6.456e-02
ty: -2.154e-03 p: 1.045e+01
invariant mass: 1.019e+00
===> Mother Track:
phi          px:  6.73e-01 py: -2.25e-02 pz:  1.04e+01,
mass:  1.02e+00, flag: 2 Begin-Vtx: [   1] End-Tracks: [36] [35]
===> Outgoing Tracks:
Track: K+           px:  3.58e-01 py: -8.70e-02 ...
Pattern: K+, (charge == 1)
Track: K-           px:  3.16e-01 py:  6.45e-02 ...
Pattern: K-, (charge == -1)
=================> End of phi object
```

The output shows track parameters and relations to other `MC_Track`, `MC_Vertex`, `RecoTrack` or `Topp` objects.

In contrast to the `selectMCtruth()` method, which collects **all** decays, the `selectRECOfromMCtruth()` method collects only those decays which were reconstructed. A MC decay is considered to be reconstructed if all decay products were reconstructed and thus have a relation to a `RecoTrack` object. In case the mother track has not been reconstructed, a Kalman filter based vertex fit is performed using the found daughter tracks in order to get the reconstructed vertex.

```
  TrackPattern KMinus = K_minus.createPattern();
  TrackPattern KPlus  = K_plus.createPattern();

  TwoProngDecay Phi = phi.decaysTo(KMinus & KPlus);

  // prepare reconstructed tracks for vertexing
  event->PrepareRecoTracksForVertexing();

  // find all reconstructable decays and perform Kalman
  // filter based vertex fit on those
  Phi->selectRECOfromMCtruth();
```

The third function `selectNORECOfromMCtruth()` collects all decays which were **not** reconstructed, that means at least one of the decay products has no relation to a `RecoTrack` object.

```
  TrackPattern KMinus = K_minus.createPattern();
  TrackPattern KPlus  = K_plus.createPattern();

  TwoProngDecay Phi = phi.decaysTo(KMinus & KPlus);
```

```
// collect all decays which were not reconstructed
Phi->selectNORECOfromMCtruth();
```

The outgoing tracks are accessible through the Decay Pattern Match objects. In case the outgoing track was reconstructed, a Pattern Match of the `RecoTrack` object is stored in the Decay Pattern Match object. If the track was not reconstructed, a Pattern Match of the `MC_Track` object is stored. This allows to easily look at tracks which were missed by the track reconstruction and to analyse why decays could not be reconstructed. The following example shows how one can find out which of the kaon tracks was reconstructed and which not:

```
TrackPattern KMinus = K_minus.createPattern();
TrackPattern KPlus  = K_plus.createPattern();

TwoProngDecay Phi = phi.decaysTo(KMinus & KPlus);

// collect all decays which were not reconstructed
Phi->selectNORECOfromMCtruth();

cout << "No. of phi's: " << Phi->matches().size() << endl;

// look at decays which could not be reconstructed
const TwoProngCandidateConstIterator end = Phi->matches().end();
for(TwoProngCandidateConstIterator it = Phi->matches().begin();
    it != end; ++it) {
  const TwoProngCandidate cand = *it;

  for(unsigned int i=0; i<2; ++i) {
    const PatternMatchIf* t = cand->PatternMatch(i);

    cout << "found " << t->getPattern().topp().name();

    if(t->isMCmatch() == true) {
      cout << " is MC track" << endl;
      const MC_Track& m = t->getMC_Track();
      ...
    } else {
      cout << " is RECO track" << endl;
      const RecoTrack& r = t->getRecoTrack();
      ...
    } // if
  } // for
} // for
```

The output can look like this:

```
No of phi's: 1
found K+ is MC track
found K- is RECO track
```

References

[1] `http://www.mpi-hd.mpg.de/herab/clue` or `/afs/desy.de/group/hera-b/BEE`

[2] T. Glebe, SMatrix - A high performance library for Vector/Matrix calculation and Vertexing, HERA-B Note 01-134, Software 01-017.

[3] T. Glebe, BEE - Interfaces and Design Issues, HERA-B Note 01-136, Software 01-018.

[4] PDG (Particle Data Group), `http://pdg.lbl.gov`

[5] Nathan C. Myers, Traits: a new and useful template technique, C++ Report, June 1995.

[6] T. Glebe, SmartPointer, HERA-B Note 01-006, Software 01-004, 2001.

[7] T. Glebe, Clue - The BEE event model library, HERA-B Note 01-138, Software 01-019, 2001.

[8] T. Glebe, GRelation 1.12 - A universal concept for object relations, HERA-B Note 01-188, Software 01-014, 2001.

[9] `http://www.mpi-hd.mpg.de/herab/clue/docu/ROOTsetup.html`

[10] T. Glebe, BEE - Compilation and Environment, HERA-B Note 02-001, Software 02-001, 2002.

[11] `http://www-hera-b.desy.de/subgroup/software`

[12] T. Glebe, Cluearte - The ARTE interface to BEE, HERA-B Note 01-110, Software 01-015, 2001.

[13] Stepanov, A. and Meng Lee, The Standard Template Library, ANSI X3J16-94-0095 / ISO WG21-NO482