Clue - The BEE event model library

Version 0-86

HERA-B Note 01-138, Software 01-019, UPDATED

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

January 22, 2002

Abstract

`Clue` is the event model library of the data analysis framework `BEE` and provides the necessary code for the representation and I/O of HERA-B data.

Contents

1 Introduction

`BEE` is a data analysis framework for particle physics experiments. A set of abstractions keep the main functionalities of the framework independent from a concrete and detector dependent data representation [1]. The `Clue` project provides a set of classes to represent measured and Monte Carlo (MC) data suitable for the HERA-B experiment [2]. Data persistency is implemented by making use of ROOT's `TTree` concept [3], which allows to access the data on an event-by-event basis as well as in a NTUPLE like way. Another advantage of ROOT is the `Streamer` functionality, which allows to modify data structures and maintain backward compatibility when reading data. Thus the `BEE` analysis framework is open for modifications of the event model. In the following we will call the ROOT files with the HERA-B event structure, as defined by `Clue`, `BEE-files`.

2 Data reduction issues

The event structure provided by `Clue` is filled in the HERA-B reconstruction framework ARTE [4] by the Cluearte [5] code. After the reconstruction step, a lot of information about an event is available in ARTE. However, only a subset of the information which is found to be neccessary for a physics analysis is stored in the `BEE` event structure defined by `Clue`. In addition to this, data reduction algorithms for event or track selection can be called. Thus a considerable data reduction can be achieved. The advantage of this is the speedup of an physics analysis in `BEE`. It has been taken care that the data represented by the `Clue` classes occupy a minimal amount of disk space by avoiding redundant information and optimizing the data types (e.g. by using a `short int` instead of `int` wherever possible). A third advantage is that the ROOT files, to which the event structure is written, feature a transparent compression via the `gzip` algorithm.

As an example a file size comparison between a run stored in MINI-DST[1] format as it is produced by the ARTE reprocessing 2 (REPRO 2) and a `BEE`-file is shown[2]:

$$\frac{\text{size(MINI, 154 files)}}{\text{size(BEE+Cloneremove+anl\_evselect)}} = \frac{11\,902\,100\ \text{kB}}{826\,796\ \text{kB}} \approx 14$$

A coarse event selection plus a track selection (Cloneremove, [7]) has been applied on the MINI data before writing the information to the `BEE`-file. The file size reduction achieved by `BEE` allows to store several runs on the local disk of a state-of-the-art PC or Laptop independent from a large computing infrastructure.

In case of Monte Carlo data, the data reduction compared to ARTE DST files is different:

$$\frac{\text{size(MINI, 17 files)}}{\text{size(BEE)}} = \frac{1\,572\,494\ \text{kB}}{187\,596\ \text{kB}} \approx 8$$

For the above comparison, minimum bias MC data which was produced with ARTE-03-08-r2 has been used (`/acs/mc3/99.1013/10000` at DESY). Only a part of the MC information available in ARTE is stored in `BEE`-files (see section 6). The Cluearte project [5] allows to reduce the MC data further, e.g. by storing only MC tracks which are above a certain energy threshold. This allows to skip the information from low energetic particles like photons from electromagnetic showers.

[1]DST = **D**ata **S**ummary **T**ape, a ZEBRA [6] based format.
[2]The single-lepton run 17137 from the year 2000 was used.

As an example, the file size comparison is shown for the case that we omit all MC particles which have an energy E below 100 MeV:

$$\frac{\text{size(MINI, 17 files)}}{\text{size(BEE, } E \geq 100 \text{ MeV)}} = \frac{1\,572\,494 \text{ kB}}{61\,368 \text{ kB}} \approx 26$$

3 Data structure

The information in a BEE-file is organized in a hierarchical structure. The data structure is drawn schematically in figure 1.

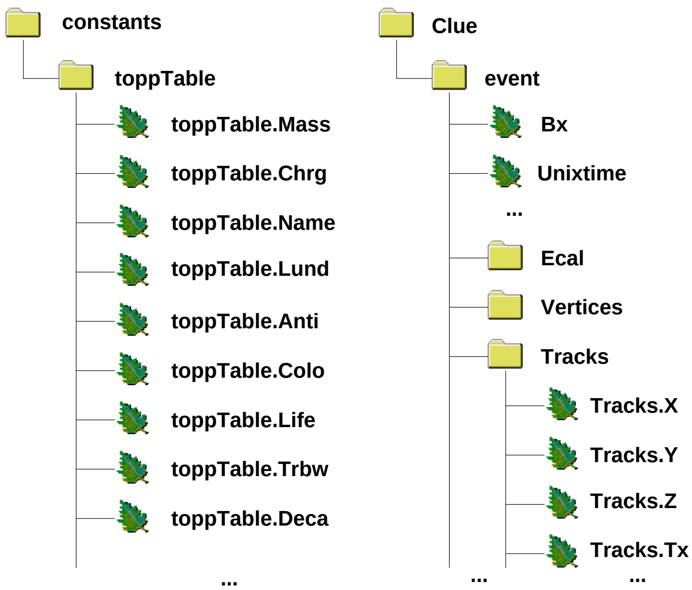


Figure 1: Event structure defined by `Clue`.

A `BEE`-file consists of two `TTree` objects:

`constants`: This tree is filled once at the beginning of a job and contains data which is independent from an individual event.

`Clue`: This tree contains the information of individual HERA-B events.

Each tree consists of a single `TBranch` object:

`toppTable`: Branch for the **T**able **o**f **p**article **p**roperties (Topp). The Topp table contains an entry for each known particle (from quarks to nucleons) with basic properties (charge, color, ...). The Topp table is most important for Monte Carlo data and pattern based analysis code.

`event`: Branch for HERA-B events. The event structure is defined by the `RtEvent` class. It contains event header information like the bunch crossing number ("Bx") and the unix time ("Unixtime") of the event. In addition to this, containers exist to hold the information about tracks, vertices, ECAL cluster and more. The containers are implemented via the GClonesArray class, which is a STL compatible [8] template wrapper for the ROOT TClonesArray container class (see figure 4). The GClonesArray class is defined in the BEE project Gate.

The event header information can be found at the top level of the event branch. Each container opens a new level in the hierarchy. The leaves of the tree structure represent the information stored in the BEE-files. For example, "Tracks.Tx" represents the x component of a track slope for reconstructed tracks. In ROOT, plots can be generated interactively by a double-click on the leaf symbol in the file browser (TBrowser).

For a complete list of information supported by RtEvent, see the Clue reference manual.

Note: Accessing one single data member in a branch implies reading the full object.

The reason for this data structure is *extensibility*. For example, the detector geometry could be easily added as a separate branch of the constants tree without affecting backward compatibility of the code. This would allow to write an event display based on ROOT functionality. Another advantage is that TBranch objects can be loaded individually. This allows to load only a part of the event structure in case one is only interested in a specific part of the event. Partial loading of events will increase the performance of analysis code because the amount of slow harddisk access can be reduced.

Note: To reduce the time needed for loading events, unused branches can be switched off before loading BEE-files. For more information see section 4.

To integrate the Clue classes into the ROOT OO DB[3], they must inherit from ROOT's TObject class. In addition to this, the Clue classes must comply with the interfaces defined in the Interfaces project. Both is achieved by multiple inheritance. The following picture shows the inheritance tree for a class representing reconstructed vertices:

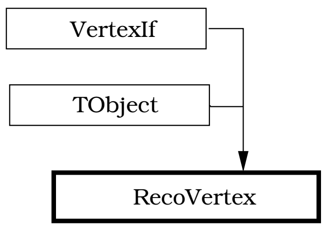


Figure 2: Inheritance tree of the RecoVertex class.

In the event structure these vertices are found in the "Vertices" container. Table 1 shows the full list of containers available in RtEvent and the Clue classes of the objects which are stored in the container. In addition the ARTE data type is listed to which the Clue data type corresponds.

[3]ROOT's Object Oriented Data Base

container name	Clue data type	ARTE data type
Tracks	RecoTrack	RTRA
Vertices	RecoVertex	RVER
Ecal	EcalHit	RCCL
Wires	Wire	RTAR
Clones	TrackClones	—
Vds	VDS_Track	RSEG
Otr	OTR_SpacePoint	RSEG
Rich	RICH_Track	RSEG
Muon	MuonTrack	RSEG

Table 1: Table of data types (classes) stored in event containers.

The data structure of a `BEE`-file is *dynamic*. A `user` field in the event header is implemented as a STL `vector<float>` object which can be filled with an arbitrary amount of information by the user. This vector is stored together with the event header in the `BEE`-file. The `user` vector allows to store information which is not forseen in the event structure and gives the `BEE`-user some flexibility when writing `BEE`-files.

`Note:` The `user` field must be later on decoded manually by the user, as no description of the data structure is currently forseen.

`Note:` The structure of the `Clue` classes used to represent data from the ARTE framework (see table 1) are very similar to the corresponding ARTE classes. However in case of reconstructed tracks an important difference exists: since `BEE` version `BEE`-0-99, the momentum dependent elements of the covariance matrix do not depend on Q/p but on p (Q = track charge, p = track momentum)[4]. The transformation is done in the following way:

$$\begin{aligned}
\mathrm{cov}(x,p) &= -Q \cdot p^2 \cdot \mathrm{cov}(x, Q/p) \\
\mathrm{cov}(y,p) &= -Q \cdot p^2 \cdot \mathrm{cov}(y, Q/p) \\
\mathrm{cov}(t_x,p) &= -Q \cdot p^2 \cdot \mathrm{cov}(t_x, Q/p) \\
\mathrm{cov}(t_y,p) &= -Q \cdot p^2 \cdot \mathrm{cov}(t_y, Q/p) \\
\mathrm{cov}(p,p) &= p^4 \cdot \mathrm{cov}(Q/p, Q/p)
\end{aligned}$$

The `Clue RecoSegment` class provides two member functions in order to do the above transformation in both directions:

`sPinvToP()` for the transformation $\mathrm{cov}(Q/p) \to \mathrm{cov}(p)$ and

`sPToPinv()` for the transformation $\mathrm{cov}(p) \to \mathrm{cov}(Q/p)$.

Therefore the user has the possibility to switch back to the definition of the covariance matrix as it is used in ARTE.

Note: Before a `RecoSegment` or `RecoTrack` object can be used for vertexing with vertexing code based on Vt [10] (e.g. the `Vt++` [11] or `SMatrix` [12] package), some computations must be done before. This preparation can be done either for all `RecoTrack` objects:

[4]The vertexing algorithm expects $\sigma(p)$, see [10].

```
// prepare all tracks in ArteTable<RecoTrack> for
// vertexing:
event->PrepareRecoTracksforVertexing();
```

or for each RecoSegment based object individually. This can improve code performance, because the amount of computations (e.g. matrix inversion) can be reduced:

```
typedef ArteTableIterator<RecoTrack> Track_it;

const Track_it end = ArteTable<RecoTrack>::end();
for(Track_it it = ArteTable<RecoTrack>::begin();
    it != end; ++it) {
  RecoTrack& rtra = **it;

  // skip all tracks with too few hits
  if(rtra.hsi() < 6 ||
     (rtra.hou() + rtra.hin() < 16))
     continue;

  // Do preparation only for useful tracks!
  rtra.PrepareForVertexing();

  ...
}
```

Preparing a track for vertexing means to propagate it to $z = 0$ and to fill the inverse covariance matrix.

4 Data access

With data organized in ROOT TTree objects it is possible to a have both a vertical data access (like with PAW NTUPLE's) and an event-by-event access (TTree split mode 1). Both access methods work via a ROOT TChain object. To make the Clue event structure accessible in ROOT, a RtEvent object must be created. The TChain object gets the name of the TTree object in the data file (here: "Clue") as a constructor argument to be able to read the tree. Finally the event branch is attached to the TChain object:

```
RtEvent* event = new RtEvent();
TChain* f = new TChain("Clue");
f->SetBranchAddress("event",&event);
```

In case a branch should be ignored in order to speed up the loading of an event, this can be done via the TChain object. The following example shows how one can ignore the ECAL clusters:

```
f->SetBranchStatus("Ecal.*",0)
```

When an event is loaded, the Ecal branch will remain empty.

Note: Relations do not care whether a branch was ignored or not. Thus it might be that relations exist to objects which are not available because the corresponding branch was not loaded.

Now data files can be attached to the chain. The files in the chain are logically treated as one big file:

```
f->Add("/data/rp02_17137.root");
f->Add("/data/rp02_17139.root");
```

Both vertical and event-by-event data access can now take place via the `TChain` object `f`.

Note: it might be that ROOT gives a warning message when a data file is added to a `TChain`:

```
root [0] f->Add("bee-file.root")
Warning in <TClass::TClass>: no dictionary for class TrackIf is available
Warning in <TClass::TClass>: no dictionary for class VertexIf is available
Warning in <TClass::TClass>: no dictionary for class ToppIf is available
```

The reason for this warning is that ROOT stores the layout of the data types together with the data in a `BEE`-file. When reading the data structure back, ROOT checks whether the data types in the `BEE`-file appear in the ROOT dictionary (which is compiled into the `Clue` library). This is of course not the case for interface classes, because they are not inherited from ROOT's `TObject` class [1] (see figure 2). Therefore the warning is meaningless and can be safely ignored.

4.1 Vertical data access

Sophisticated plots can be generated by making use of the `TChain::Draw()` command. In this way the `BEE`-files can be used for a vertical access of the data, like in PAW NTUPLE files. This has the advantage that plots can be generated interactively and very fast. ROOT scripts can be written and executed on the fly in the CINT interpreter. For more examples see the ROOT homepage [3]. An example is shown in figure 3.

Note: When accessing the data interactively via ROOT's CINT interpreter and the `TChain::Draw()` command, the data must be addressed via the name of the *private* data member (like “Tracks.Tx” or “Bx”) as it appears in the `BEE`-file. In case of event-by-event access the private data members are hidden and access functions must be used (see section 4.2).

4.2 Event-by-Event data access, ARTE tables

The ROOT `TChain` class provides the functionality of looping over single events:

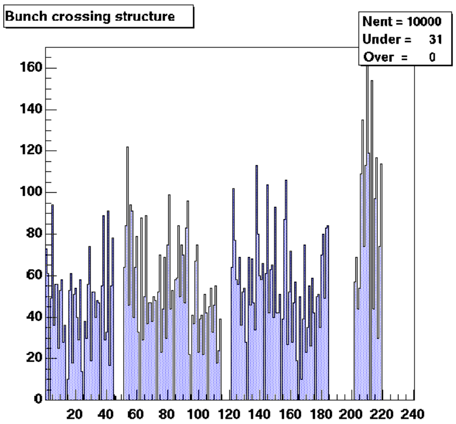
```
const Stat_t NEVENTS = f->GetEntries();
cout "No of events in chain: " << NEVENTS << endl;

for(unsigned int i=0; i < NEVENTS; ++i) {
  f->GetEntry(i);   // load ith event
  ...
}
```

The event header information has to be accessed always via the pointer to the `RtEvent` structure. Access to the containers is possible in two ways:

```
// example:  plot BX spectrum
TH1F* bx = new TH1F("bx","Bunch
crossing structure", 240, 0.5,
240.5);
bx->SetFillStyle(3002);
bx->SetFillColor(4);
f->Draw("Bx>>bx","","0",10000);
bx->Draw();
```

(a) ROOT macro code



(b) Resulting plot

Figure 3: Example of vertical access in BEE-files.

RtEvent: As shown in section 4, a global `RtEvent` object must be attached to the `TChain` object `f`. The pointer to the `RtEvent` object, in our example called `event`, gives access to the full event structure provided by the `RtEvent` class (see example below).

ArteTable: BEE supports the `ArteTable` syntax to access event information in the same way as it is possible in ARTE. This allows to easily port code running in ARTE to BEE. `ArteTable` objects are global objects which refer to the appropriate container in the BEE event structure. `ArteTable`'s are important because they are used for a safe implementation of releations between the objects stored in ARTE tables (see section 5). The `ArteTable` interface is defined in the BEE project `arte`. Figure 4 shows how `ArteTable` objects are used to give a uniform access to information stored in completly different ways.

Example for using the `event` pointer to access the `GClonesArray` members of the `RtEvent` class:

```
f->GetEntry(10); // get 11th event in chain
cout << "Event information: " << *event << endl;

typedef GObjArray<RecoTrack*>::const_iterator  Track_it;

// loop over RecoTrack container in current event
const Track_it end = event->tracks().end();
for(Track_it it = event->tracks().begin();
    it != end; ++it) {
   cout << " tx: " << (*it)->tx()
        << " ty: " << (*it)->ty()
```

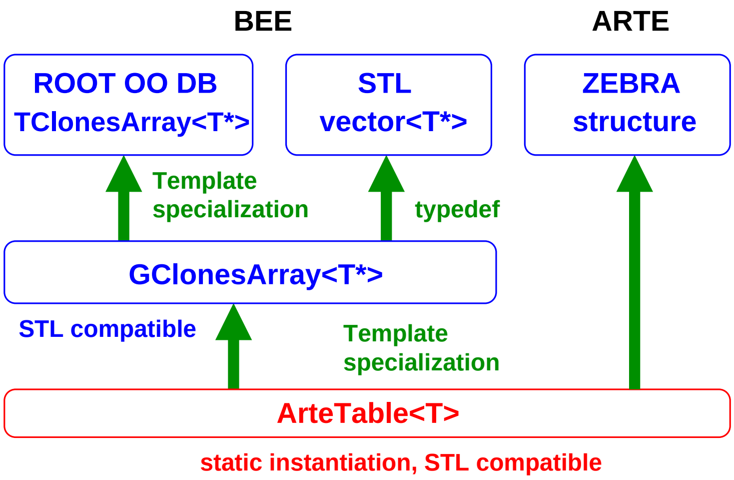


Figure 4: Uniform data access via the `ArteTable` interface

```
        << endl;
}
```

Example for using the `ArteTable` interface:

```
f->GetEntry(5); // get 6th event in chain
cout << "Interaction rate: " << event->ir() << endl;

typedef ArteTableIterator<RecoTrack> Track_it;

// loop over Tracks in event
const Track_it end = ArteTable<RecoTrack>::end();
for(Track_it it = ArteTable<RecoTrack>::begin();
    it != end; ++it) {
  const ArtePointer<RecoTrack> ptr = *it;
  cout << " tx: " << ptr->tx()
       << " ty: " << ptr->ty()
       << endl;
}
```

Note: Both access methods are equivalent.

Note: The `end()` optimization is done in both cases for performance reasons.

The example shows that `ArteTable` objects are global and allow to access the event containers for tracks, vertices etc. without knowing about the concrete implementation of the container. In case of ROOT absence, the containers can be e.g. STL `vector` containers. `ArteTable` objects are *Singletons* because the template structure allows only one table per data type. This assures an unamgiuous mapping of a data type to an array (table). The `ArteTable` interface is available not only in compiled code, but also in an interactive ROOT/CINT session. The following example shows how the contents of an `ArteTable` can be printed to screen in an interactive ROOT session:

```
root [3] cout << ArteTable<RecoTrack>() << endl;
```

The objects stored in ARTE tables can be referenced by a special pointer (`ArtePointer`). An `ArtePointer` represents the row number of the ARTE table to which it is pointing. The object stored in the ARTE table can then be accessed by dereferencing the pointer or by pointer-access (see figure 5).

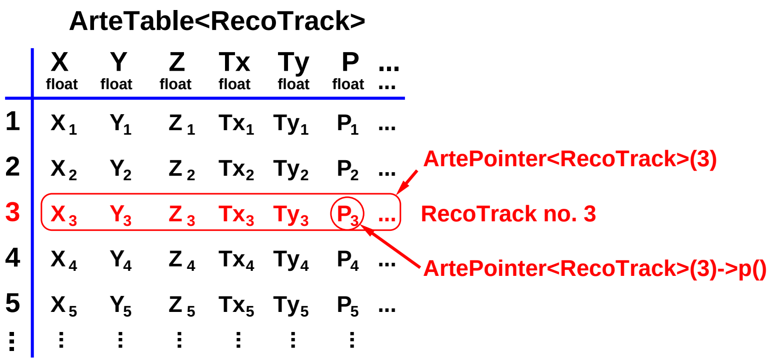


Figure 5: Data representation in `ArteTable<RecoTrack>`

5 Object relations

Because of the unambiguous nature of ARTE tables, we are able to define relations between objects of same or different data type stored in those tables. Object relations are implemented via the `ClueRelation` classes, which are defined in the `GRelation` project [9].

The three relation types implemented in `BEE` are **One-to-One**, **One-to-Many** and **Many-to-Many** (see figure 6).

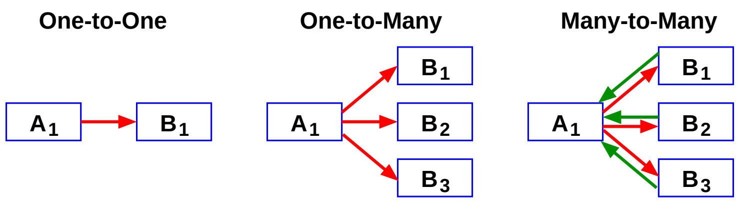


Figure 6: Relations between objects of type **A** and **B**.

Table 2 shows the relations which are defined in `Clue`. The relations are filled when the `BEE`-file is created in ARTE by the Cluearte project. A relation is represented by an `ArtePointer` object. An `ArtePointer` is superior to a simple table index (of type `unsigned int`) because it

data type 1	data type 2	relation type	description	name
RecoSegment	RTRA	One-to-One	link to track in ArteTable<RTRA>	rtra()
RecoSegment	PatternMatch	Many-to-Many	link to matched track pattern	pmatch
RecoTrack	MC_Track	Many-to-Many	matched MC truth track	mtrack
RecoTrack	EcalHit	Many-to-Many	ECAL Clusters matched with track	ehit
RecoTrack	TrackClones	Many-to-Many	link to Clone types (= track segments)	clones
RecoTrack	Topp	One-to-One	particle hypothesis	topp()
RecoTrack	RecoVertex	Many-to-Many	reconstructed vertices to which track belongs	rvertex
RecoVertex	RecoTrack	Many-to-Many	link to outgoing tracks	out
TrackClones	RecoTrack	Many-to-Many	link to tracks which share the same segment	rtrack
EcalHit	RecoTrack	Many-to-Many	tracks matched to ECAL cluster	rtrack
MC_Track	Topp	One-to-One	true particle type	topp()
MC_Track	MC_Vertex	One-to-One	begin vertex (origin of MC track)	bVertex()
MC_Track	MC_Track	One-to-One	true MC mother track	mother()
MC_Track	RecoTrack	Many-to-Many	matched reconstructed track	rtrack
MC_Track	MC_Vertex	One-to-Many	end vertices (true interactions)	endVtx
MC_Track	MC_Track	One-to-Many	true outgoing MC tracks	endTrack
MC_Vertex	MC_Track	One-to-One	true MC mother track (incoming track)	mother()
MC_Vertex	MC_Track	One-to-Many	true outgoing MC tracks	out
Topp	Topp	One-to-One	anti particle	anti_ptr()

Table 2: Table of relations defined in Clue.

remembers the type of the related object (and thus avoids run-time errors which are hard to debug) and provides methods to access the related object.

In case of a One-to-One relation, a single pointer is stored. Using the relation means to access the related object via the relation pointer:

```
// Example: loop over MC tracks and access the
//          mother vertex via the bVertex relation
typedef ArteTableIterator<MC_Track> Track_it;

const Track_it end = ArteTable<MC_Track>::end();
for(Track_it it = ArteTable<MC_Track>::begin();
    it != end; ++it) {
  const ArtePointer<MC_Track> mtra = *it;

  // get relation to begin vertex (exists always)
  const ArtePointer<MC_Vertex> mver = mtra->bVertex();
  cout << "Track origin: " << mver->vpos() << endl;
}
```

In case of One-to-Many and Many-to-Many relations, a list of pointers to objects is stored. This list can be iterated via a `ClueRelationIterator` object [9]:

```
// Example: loop over relations to outgoing tracks
//          in case of reconstructed vertices
typedef ArteTableIterator<RecoVertex> Vertex_it;

const Vertex_it end = ArteTable<RecoVertex>::end();
for(Vertex_it it = ArteTable<RecoVertex>::begin();
    it != end; ++it) {
  const ArtePointer<RecoVertex> rver = *it;

  // loop over relations to outgoing tracks
  typedef ClueRelationIterator<RecoTrack> RelTrack_it;

  const RelTrack_it tend = rver->out.end();
  for(RelTrack_it rit = rver->out.begin();
      rit != tend; ++rit) {
    const ArtePointer<RecoTrack> rtra = *rit;

    cout << "Track: " << *rtra << endl;
  }
}
```

Note: The relations are implemented such that they must be known at compile time. A disadvantage of this is that relations cannot be freely defined and used during run time. The big advantage is performance, because for each relation a separate container is generated per object at compile time and thus the list of relations to be iterated is always small.

Relations are not stored together with the objects, but separately at the end of each event. A special class `StreamObject` exists which is able to read and write the relations. For technical reasons, a `StreamObject*` pointer must be the last member of the `RtEvent` class in order to stream the relations at the end of an event.

6 Representation of Monte Carlo data

Monte Carlo truth information is stored in a very similar way to ARTE. However, not the full information as it is available in ARTE is forseen at the moment in `Clue`. The representation of MC truth data is done by the `MC_Track` and the `MC_Vertex` classes.

The way the MC truth data is represented is different from the way it is done with reconstructed data. A MC truth track has relations to

1. one mother track
2. one mother vertex
3. one or more end tracks and
4. one or more end vertices.

The reason for this is that several physical processes can be represented in this way:

Stable particles with interactions: In case a MC track has some non-destructive interactions with matter, e.g. electromagnetic interactions in a part of the detector, an end vertex is generated for each of the interactions. The outgoing track is then a photon which can e.g. convert later on into an $e^+ e^-$-pair. See figure 7a.

Decaying particles with interactions: In case of a decaying particle with non-zero track length, interactions with matter may occur like in the case of a non-decaying particle. In addition to this, one end vertex describes the decay of the particle. This decay vertex also defines the track length. See figure 7b.

Resonances: A resonance is a particle which decays immediatly, that means at the place of it's generation. Therefore no end vertex exists and only the relation to the end tracks has to be filled. The track length is 0 (see figure 7c).

Note: The relation to end tracks is filled in a different way as it is done in ARTE. In ARTE a relation to end tracks is only created for MC tracks which have generated at least one MIMP[5]. In contrast to this, in `Clue` an end track relation is created to **all** outgoing tracks.

Note: A relation to a mother track exists for all MC tracks, except for the initial particles (e.g. a quark pair). A begin vertex is available for all MC tracks, even for the initial particles (as they are generated in a primary vertex).

The current format of the MC truth representation does not allow to disentangle detector acceptance from reconstruction efficiencies, as no information of MC impact points (MIMP's) is present in `Clue`. `BEE` is not forseen to be an alternative for any kind of MC studies which can be

[5]A **M**onte Carlo **Im**pact **P**oint, which is a hit on a sensitive detector element.

carried out in ARTE. The goal is to provide the key information needed for a physics analysis. The advantage of this is that the size of BEE-files containing MC truth information is much smaller than in case of ARTE (see section 2). On the other hand the current MC format is not considered to be final and is open for future modifications.

The MC_Track class has a special function to draw the MC decay chain: print_decay(). In the following an example is shown which draws the decay of all ϕ particles found in an event:

```
typedef ArteTableIterator<MC_Track> MCIt;

const MCIt end = ArteTable<MC_Track>::end();
for(MCIt it = ArteTable<MC_Track>::begin(); it != end; ++it) {
  const ArtePointer<MC_Track> mtr = *it;

  if(*mtr == phi) {
    mtr->print_decay(0,'r');
    cout << endl;
  }
}
```

The first argument of the print_decay() function specifies the level of the decay tree to which the decay chain should be printed. The second argument gives the type of information to be printed (see reference manual for details).

7 Representation of Track Clones

In case of HERA-B, track reconstruction takes place in an (almost) independent way in the various subdetectors like vertex detector, inner and outer track, calorimeter and muon system. After reconstruction of the track segments in the subdetectors, a matching procedure is called to connect track segments to a complete track, assign momentum information and do a global refit. During this track reconstruction procedure it may happen that the same track segment is used in different reconstructed tracks, because it could be matched to several other track segments. As it might not be possible at reconstruction time to decide which of the tracks sharing the same segment is the physical track, one has to introduce a bookkeeping of these “clone” tracks in order to decide later during a physics analysis which clone is a physical track and which clone is a fake track (“ghost”).

In Clue, the TrackClones class was introduced to allow a bookkeeping of the segments shared by different reconstructed tracks. An object of type TrackClones is created for each track segment which is used in more than one reconstructed track. All TrackClones objects are stored in the ArteTable<TrackClones> table. Therefore the ArteTable<TrackClones> represents the list of all track segments which are used more than once. The subdetector in which the multiple used track segment was reconstructed is stored via the component bit mask in the TrackClones object. The component bit mask is implemented via the RsegC class (see the reference manual).

Note: ARTE does not have a table which corresponds to the Clue table ArteTable<TrackClones>, the ids of the used segments are stored directly in the RTRA object.

Between the TrackClones and the RecoTrack class a Many-to-Many relation is defined. Therefore it is possible to loop directly over all reconstructed tracks using the same segments and, from the other side, to loop over all segments of a track which are reused in other tracks.

The following example shows how one can loop over all reconstructed tracks which share a segment with a given track (represented by the rtra pointer):

```
const ArtePointer<RecoTrack> rtra = ...; // get from somewhere

typedef ClueRelationIterator<TrackClones> Clone_it;

// loop over all clone segments
const Clone_it cend = rtra->clones.end();
for(Clone_it it = rtra->clones.begin();
    it != cend; ++it) {
  const ArtePointer<TrackClones> clone = *it;

  typedef ClueRelationIterator<RecoTrack> Track_it;

  // loop over all tracks using this segment
  const Track_it tend = clone->rtrack.end();
  for(Track_it rit = clone->rtrack.begin();
      rit != tend; ++rit) {
    const ArtePointer<RecoTrack> ctra = *rit;

    // rtra will also occur in the list, so skip it
    if(ctra != rtra) {
      cout << "Clone Track: " << *ctra << endl;
    }
  }
}
```

A big advantage compared to ARTE is that one has a direct access to the clone tracks, whereas in ARTE one has to loop over all tracks in the RTRA table in order to find the clone tracks.

References

[1] T. Glebe, BEE - Interfaces and Design Issues, HERA-B Note 01-136, Software 01-018, 2001.

[2] E. Hartouni et al., HERA-B Design Report DESY-PRC 95/01, 1995.

[3] Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch.

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

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

[6] `http://wwwinfo.cern.ch/asd/cernlib`

[7] M.-A. Pleier, Cloneremove, HERA-B Note 01-062, Software 01-009, 2001.

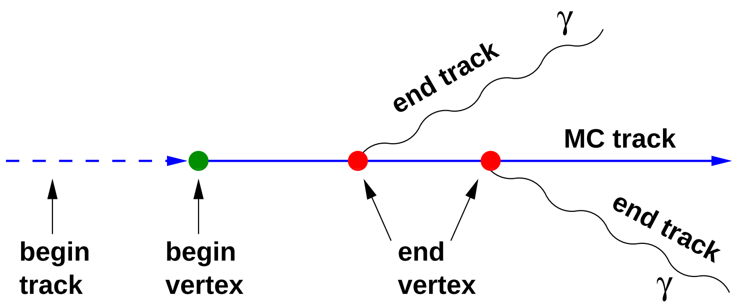
[8] Stepanov, A. and Meng Lee, The Standard Template Library, ANSI X3J16-94-0095 / ISO WG21-NO482

[9] T. Glebe, GRelation 1.12 - A universal concept for object relations, HERA-B Note 00-188, Software 00-014, 2001.

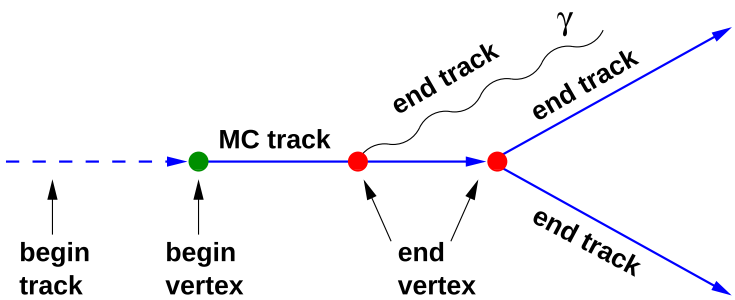
[10] T. Lohse, Vertex Reconstruction And Fitting, HERA-B Note 95-013

[11] T. Glebe, Vt++ Version 1.0, HERA-B Note 00-175, Software 00-013, 2000.

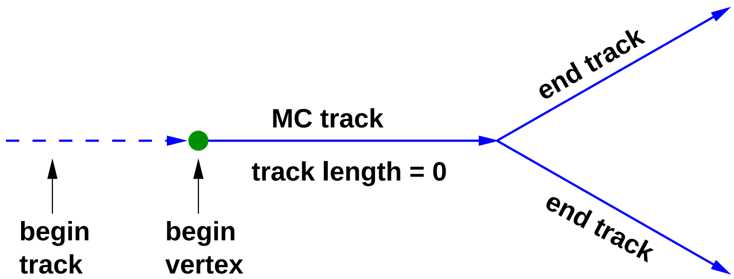
[12] T. Glebe, SMatrix - A high performance library for Vector/Matrix computation and Vertexing, HERA-B Note 01-134, Software 01-017, 2001.



(a) Non-decaying particle with electromagnetic interactions.



(b) Decaying particle with track length $\neq 0$.



(c) Representation of a resonance (track lenght $= 0$).

Figure 7: Representation of different physical processes in MC truth.