ROOT data model evolution

Lukasz Janyst

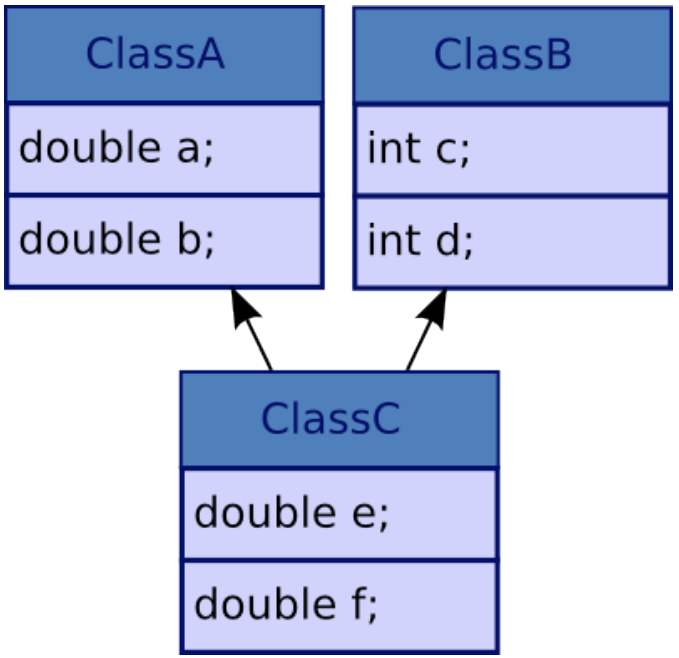
CERN/FAIS-UJ

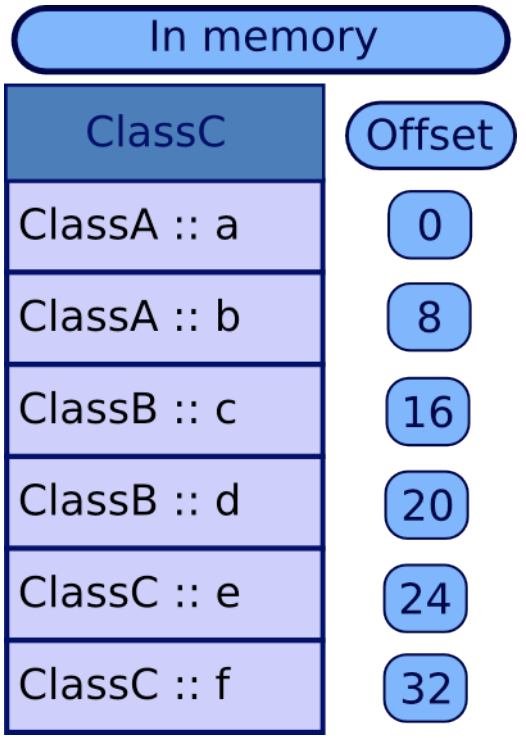
June 4, 2008

Outline

1. Memory representation of C++ objects and dictionary information
2. How does the streaming system work?
3. Currently implemented schema evolution capabilities
4. What's missing?
5. Goals of the new system
6. Design principles
7. Timescale

Memory representation of C++ objects

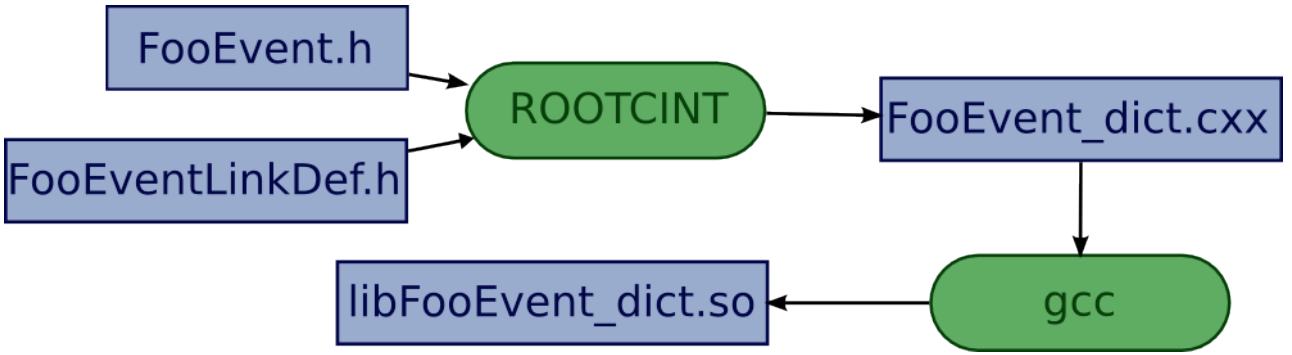





```
class ClassA { double a, b; }
class ClassB { int c, d; }
class ClassC: public ClassA,
              public ClassB
{ double e, f; }
```

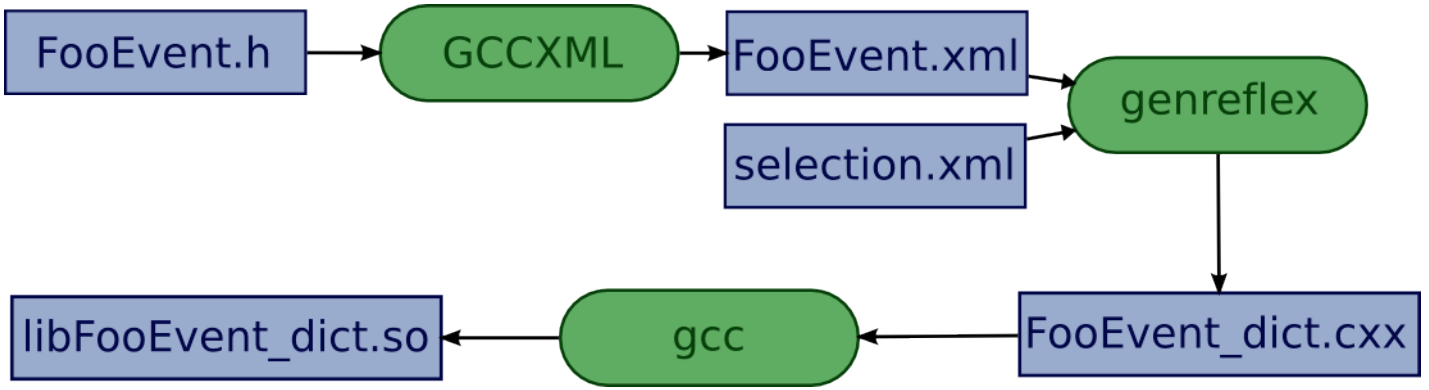
```
ClassC *ptr = ...
int &val_c =
 *(int*)((char*)ptr1 + 16);
val_c = 47;
```

How do we know the offsets? - CINT



- The header file containing data model definition and the LinkDef file are being processed by ROOTCINT resulting with dictinary C++ code
- The code is then compiled and loaded

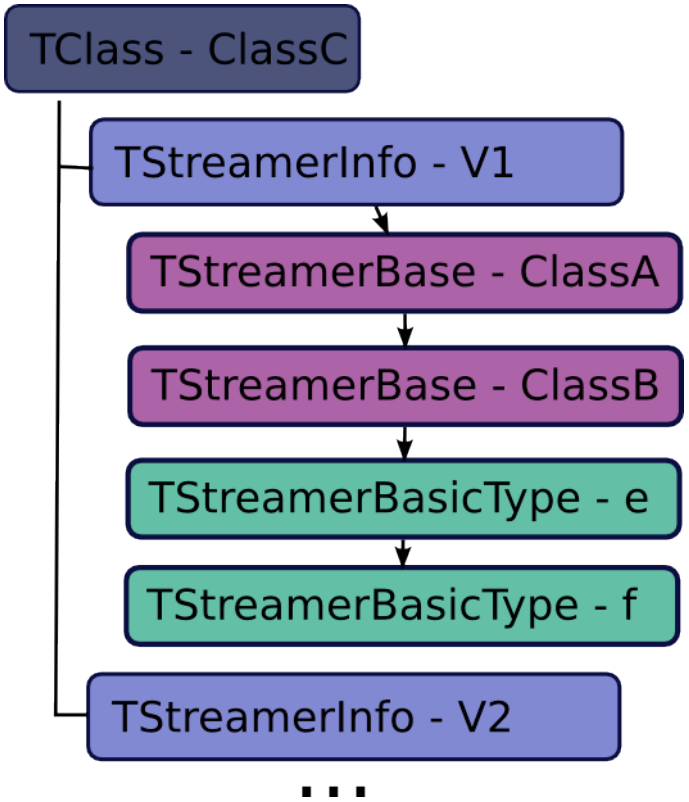
How do we know the offsets? - REFLEX



The process of dictionary generation is similar for REFLEX.

The IO subsystem works with ROOT's native introspection mechanisms so REFLEX dictionaries need to be interfaced - this is done by CINTEX.

TStreamerInfos



- Basing on dictionary information TStreamerInfo objects are created
- TStreamerInfos contain class name and version information and a list of streamer elements corresponding to persistent data members and base classes
- TStreamerElements hold information about names, types, sizes and offsets of entities they represent

- TStreamerInfos are persistified in ROOT files and contain all the information about what was written to the buffers

Collections vs ROOT IO

- While storing and reading collections we're just interested in accessing their elements in sequential way, ie. to treat them as ordinary arrays of objects
- In ROOT IO collections are wrapped into TCollectionProxies providing uniform interface to access their elements via [] operator and implementing some common operations like cleaning or resizing
- The approach works with both the STL collections and ROOT collections

Writing data and different streaming modes

Split mode

- ROOT tries to decompose objects storing data members in separate buffers if that is possible
- It usually results with smaller files since the metadata describing the objects is being handled more efficiently
- Not always supported

Non-split mode

- All data members are stored in the same buffer
- Collections can be arranged in different ways in the buffer (member-wise or object-wise)
- If we deal with collections of polymorphic pointers then we need to store the information about actual concrete object with every element of the collection

Currently implemented schema evolution capabilities

Automatic schema evolution

- Reading is done by looping over streamer elements (or branches that refer to streamer elements) and putting read data into right chunks of memory
- When new data members are added, removed or just rearanged the memory layout of the objects changes
- ROOT deals with this by matching the names of the streamer elements to the names of the data members of transient classes and adjusts offset information in streamer elements accordingly
- If some data members are missing then the corresponding streamer elements are skipped while processing the buffer(s)

Currently implemented schema evolution capabilities

Automatic schema evolution

- ROOT is also able to do the conversion between basic types, between collections (since they share the same interface) and also handle some renaming

Manual schema evolution

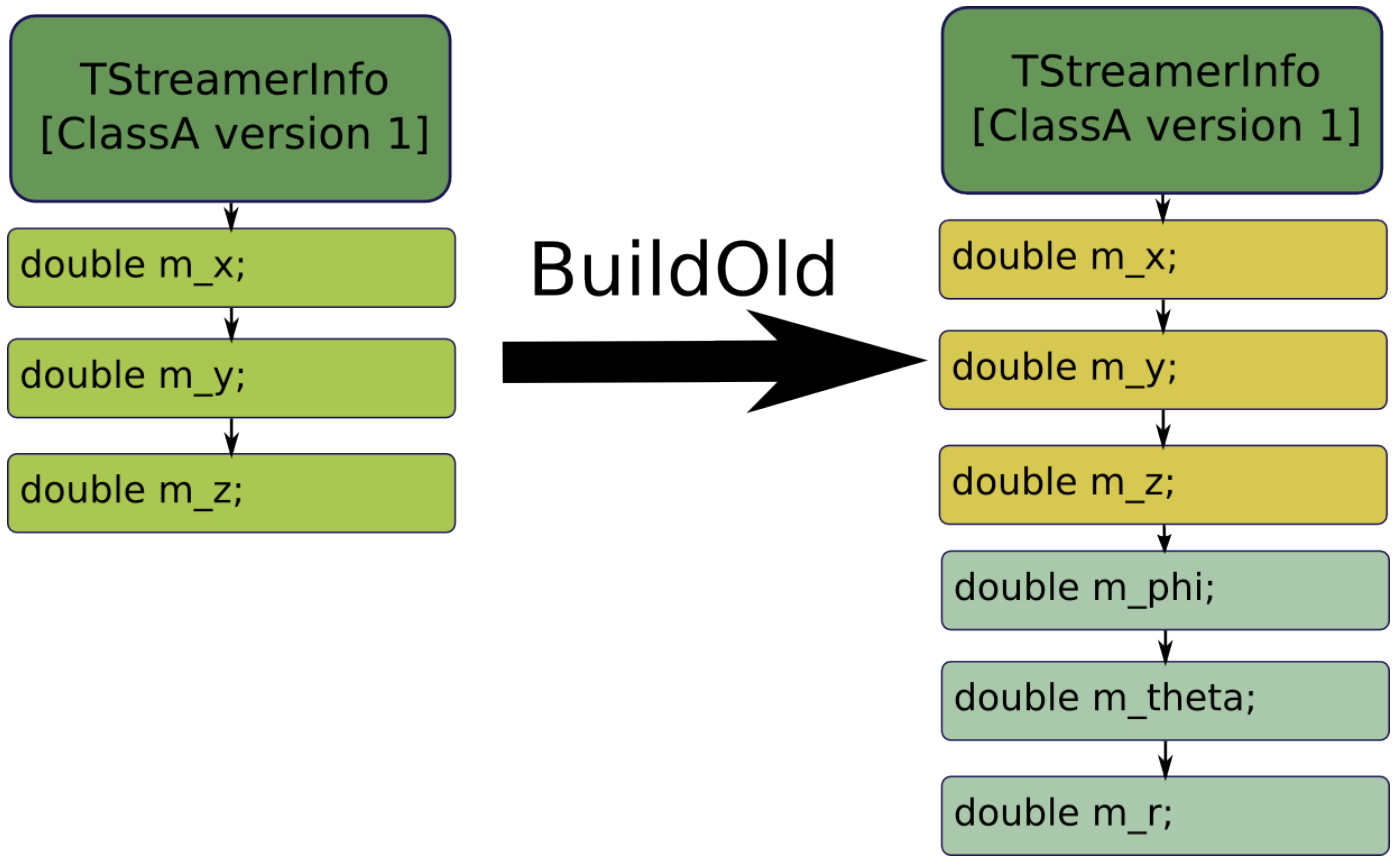
- Works only in non-split mode
- The user has to write and register a custom streamer
- The code has to manipulate the buffer directly and write the code to handle all of the versions that has ever been written even those that in principle would not need evolving

Goals of the new system

- Rename classes
- Rename data members
- Change the shape of the data structures or convert one class structure to another
- Change the meaning of data members
- Access the buffer directly
- Assign values to transient data members
- Ensure that the objects in collections are handled in the same way as the ones stored separately
- Make things operational also in bare ROOT mode

The main idea behind the design

The new functionality will be based on enhancements to the TStreamerInfo structure. New TStreamerElement concrete types will be introduced to call the conversion functions.



The main idea behind the design

- When we ancounter old versions of persistent objects we get corresponding streamer info (in-memory) and adjust it to memory shape of the object
- We search for the rules of conversion (function pointers) and insert artificial (blue) streamer elements
- We check the dependencies on old data members (yellow)
- While reading we buffer "yellow" information in memory, supply it to conversion function associated to "blue" streamer element
- The function puts the result in the right place in the memory

The conversion rules

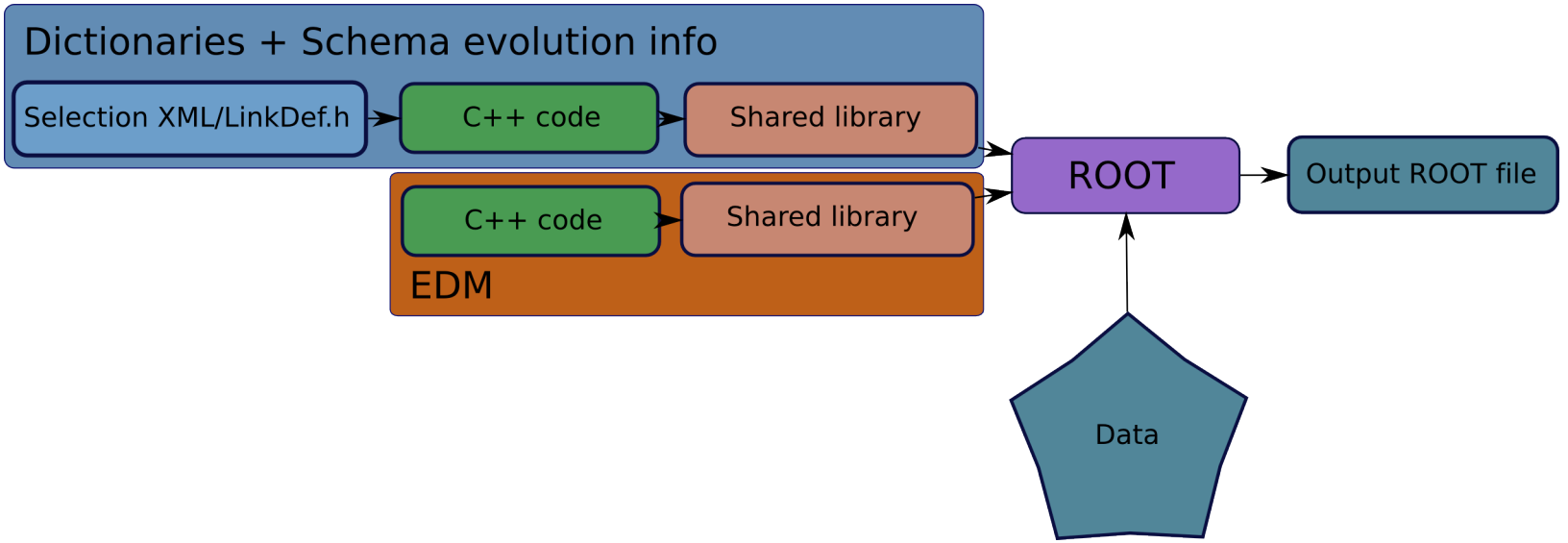
The system has to be supplied with information how to perform the conversion - a conversion function. We have forseen two ways to do that:

- (1) a code snippet supplied as string in the selection xml of REFLEX or LinkDef.h of CINT; such snippet will be then preprocessed and compiled using ACLIC functionality at dictionary generation time
- the rules supplied as strings can be stored in ROOT files to enable users to analize data files containing different versions of data with the same code
- (2) via C++ API by passing either the strings or the function pointers
- the rules provided via the API can be compiled on the fly or interpreted

The user code is available

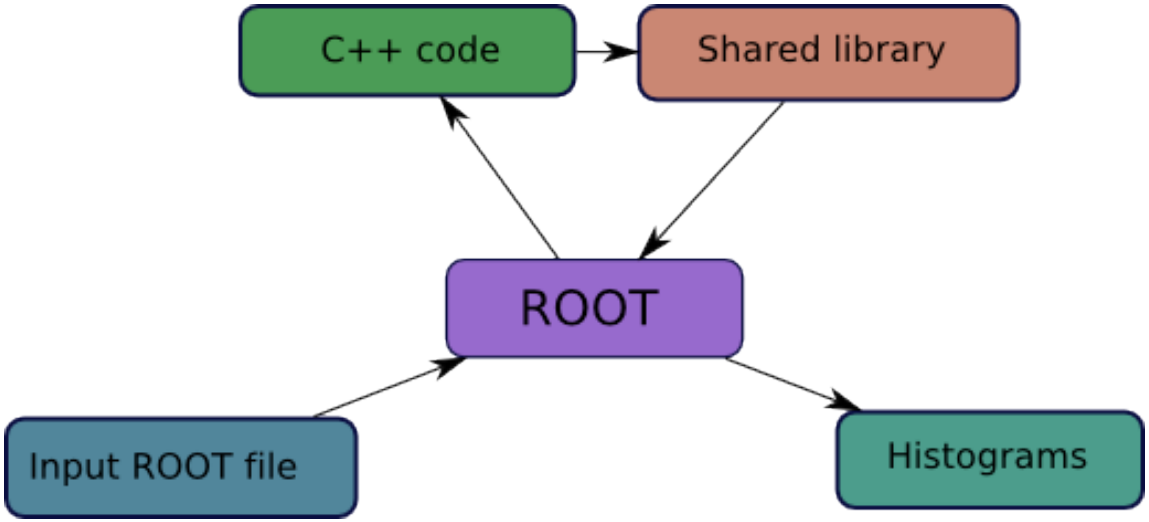
We will support two approaches for accessing files.

- When the user code (EDM libraries, dictionaries and schema evolution info) then we can write the data in the shape of current version EDM and embed the coonversion rules in the files. Naturally we could also read the files containing different versions of data evolving it to the current shape.



Bare ROOT mode - no user code

- Another tipical use case is when users look at the chains of ROOT files possibly containing different versions of the Data Model in bare ROOT mode. We also want to support the case of applying the schema evolution rules to emulated classes (the ones recreated from TStreamerInfos) basing on the code snippets stored in the files. Of course the rules describing how to evolve old files will be stored in the new ones and the functionality will be available only if the information is present.



The conversion rules

There will be two types of conversion rules:

- raw - old data is being read directly from the buffer
- normal - old data is being read from proxified “virtual” object

In the user defined code snippets the following variables will be available (prepended by the preprocessor before compilation)

- newObj - of the type of target object if the user code is available
- names of the data members of the in -memory object being read
- oldObj - TVirtualObject object holding the data read from file
- buffer - for raw rules

The syntax of the rules - LinkDef.h

```
#pragma read
   source="ClassA::m_a;ClassA::m_b;ClassA::m_c"
   version="[4-5,7,9,12-]" checksum="[12345,123456]"
   target="ClassB::m_x" targetType="int"
   embed="true" include="iostream,cstdlib"
   code="{ some C++ code }"
```

```
#pragma readraw
   source="ClassA::m_a" version="[4-5,7,9,12-]"
   checksum="[12345,123456]" target="ClassB::m_x"
   embed="true" include="iostream,cstdlib"
   code="{ some C++ code }"
```

The syntax of the rules - LinkDef.h

```
<read source="ClassA::m_a;ClassA::m_b;ClassA::m_c"
    version="[4-5,7,9,12-]" checksum="[12345,123456]"
    target="ClassB::m_x" targetType="int" embed="true"
    include="iostream,cstdlib">
<![CDATA[
some C++ code
]]>
<read>
```

```
<readraw source="ClassA::m_a" version="[4-5,7,9,12-]"
       checksum="[12345,123456]" target="ClassB::m_x"
       embed="true" include="iostream,cstdlib">
<![CDATA[
some C++ code
]]>
<readraw>
```

Example - Raw rule

```
#pragma readraw source="TAxis::fXbins"  \
        version="[-5]"                  \
        include="TAxis.h" code="        \
{                                       \
  Float_t *xbins = 0;                   \
  Int_t n = buffer.ReadArray( xbins );  \
  fXbins.Set( xbins );                  \
}"
```

The example shows how to read the old data when the layout in the buffer is different than the one expected by the system.

Normal reading rule

```
#pragma read source="ClassA::m_a;ClassA::m_b" \
        version="[-3]" target="ClassA::m_a"   \
        code="{                               \
   static int a_id( oldObj->GetId( "m_a" ) ); \
   static int b_id( oldObj->GetId( "m_b" ) ); \
   m_a = oldObj->GetMember<int>(a_id)*        \
         oldObj->GetMember<int>(b_id);        \
}"
```

This simple example shows how to handle the reading when ClassA version 1,2 and 3 had two data members (m_a and m_b) and the current shape has just one data member (m_a) that is a result of the multiplication of the old ones.

Real life example

```
class State {   // version 1
  unsigned int           m_flags;
  Gaudi::TrackVector     m_stateVector;
  Gaudi::TrackSymMatrix  m_covariance;
  double                 m_z; };
becomes:
class State { // version 2
  unsigned int           m_flags;
  StateVector            m_stateVector;
  Gaudi::TrackSymMatrix  m_covariance; };
with:
class StateVector {
  Gaudi::TrackVector m_parameters;
  double             m_z; };
```

Real life example - the rule

```
<read
   source="State::m_stateVector;ClassA::m_z"
   version="[1]"
   target="State::m_stateVector"
   embed="true">
<![CDATA[
new (&m_stateVector) StateVector(
oldObj->Get<Gaudi::TrackVector>("m_stateVector"),
 oldObj->Get<double>("m_z") );
]]>
</read>
```

Implications of the rules for collections

Let's consider the following example:

```
class Event
{
  std::vector<Track> fTracks;
};
```

- If we have a rule set for the Track class, then the rule will be run individually on each Track object before they are inserted to the vector.
- If we have a rule involving fTracks data member or for std::vector<Track> then on disk Tracks will be loaded to an emulated collection and the user will be responsible for dealing with them on his own.

Backward - forward compatibility + trees

The schema evolution rules will be stored in the same list as TStreamerInfos. If a new file is being read by a new version of ROOT the information will be read and the appropriate TStreamerInfos (in-memory) updated. If a new file is being read by old version the warning is printed that unidentified object was found in the streamer info list and the schema evolution information will be ignored.

In split mode the reading requests are forwarded to streamer infos pointing out the appropriate streamer element and buffer. To handle new "syntetic" streamer elements we will need to add some "transient" branches.

Timescale

- the first discussions started in September 2007
- the real work started in March 2008
- the document describing design principles ready in May 2008
- I'm currently working on the implementation
- Philippe Canal is going to start to work on it in the second half of July
- the first working prototype should be released at the end of August
- too late to put it int the June release