ROOT Data Model Evolution

Lukasz Janyst, Rene Brun, Philippe Canal

Table of Contents

1. Introduction

This short document aims to describe briefly the user interface and the implementation ideas of the subsystem of the ROOT framework supporting data model evolution. The goal of the subsystem is to enable users to load into current in-memory objects older versions of data model that had been serialized to ROOT files even in the situation when the data model changed significantly over the time.

1.1 Self description capabilities of the ROOT files

Every non-collection class serialized to a ROOT file has a corresponding TStreamerInfo object associated with it. The TStreamerInfo objects hold the information about the name, version, checksum, data members (their type, names and so on) and about the order in which they were stored into the buffers providing complete information about the data stored in the files.

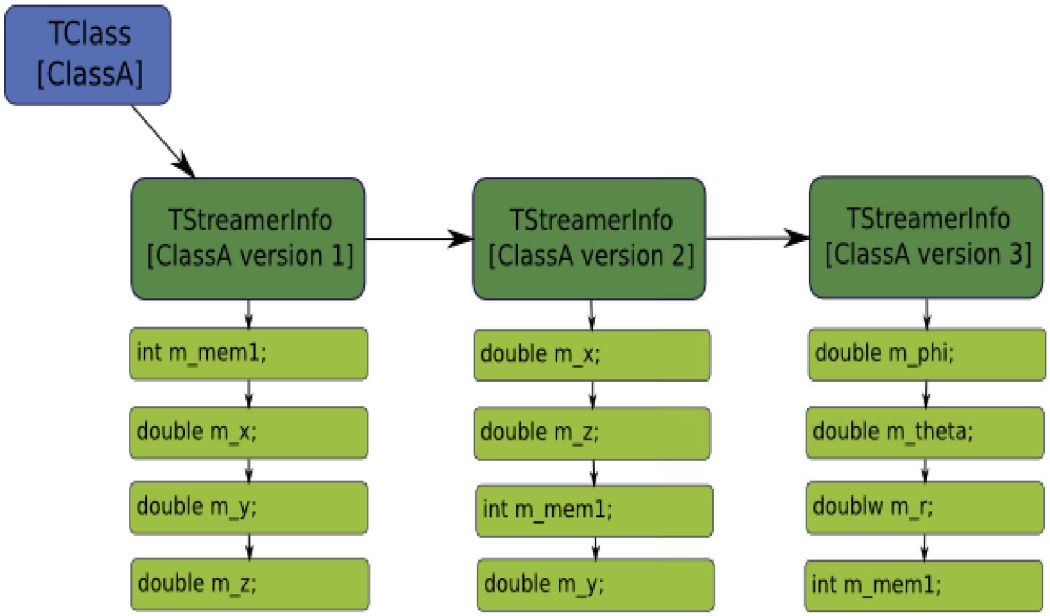
1.2 Streaming modes

There are several possible ways of arranging the data in a file. All the data can be placed sequentially in one buffer (non-split, object-wise mode) or in many buffers (split, member-wise mode). In non-split mode, when the streaming engine encounters a collection of objects it can handle it in two ways: object-wise - the elements are streamed one by one, or member-wise - the value of the first data member of all elements in the collection is stored first, then the second data member's values and so on. In the split mode the data object's data members are scattered between many buffers and can be accessed independently. For the case of collections of objects the elements are decomposed and their data members are grouped and stored together in separate buffers.

1.3 Currently implemented functionality and it's limitations

As for the day of writing this document the system's ability to load older versions of class shapes into memory works well for data written in object-wise mode. In this case the conversion can be handled using the Streamer methods. For member-wise (split-mode) streaming, however, the schema evolution abilities of the system are more limited. The framework can easily recalculate the memory offsets of the data members when they differ from the information stored in the file. This is done by simply matching the memory offset information delivered by the dictionary subsystem against the names stored in the appropriate TStreamerInfo objects. This is however possible only if the data members have the same name and contain the same logical information. When the names in the streamer information and in dictionary information match but the types of considered data member differ between disk and memory and the type is a basic type then the framework will attempt to do the conversion. The conversion between the STL collections (vectors, lists, deques, sets and multisets) works as well since those are represented by collection proxies having a uniform interface.

1.4 Problem description



Drawing 1: Graphical represenation of the relationship between TClass objects, TStreamerInfos and TSTreamerElements

Drawing 1 is a representation of a simple class containing in versions 1 and 2 an integer (mem1) and some Cartesian coordinates in three-dimensional space. For version 1 the data members were stored in some order, version 2 contains exactly the same data members but stored in different order. Theoretically this is still the same class but it is layouted in memory in different way, ie. the offsets of the data members are different. This results with different streamer info for version 2. The IO system is able to read version 1 and 2 of the serialized data into in memory object of ClassA version 2 but it is impossible to read versions 1 and to in-memory ClassA version 3, not only because the names of the data members changed but also because their logical meaning is completely different. In the remaining part we describe the proposal of a solution that addresses the problem of loading persistent data to objects of new or different shape than the one that has been used to write them. The solution has to work in both object-wise and member-wise streaming.

1.5 User interface and scalability requirements

We must support the case where many files containing different versions of the same data are being read in pure ROOT mode without any user defined objects and dictionaries. To do that we will provide the possibility of defining the conversion rules as strings containing chunks of valid C++ code which will be then wrapped in functions and compiled using ROOT's on demand compiling facility. Those code chunks will be then embedded into streamer info structure and serialized to the files. As an additional requirement the forward compatibility has to be kept, so that new files could be still readable by old versions of ROOT.

1.6 Use cases

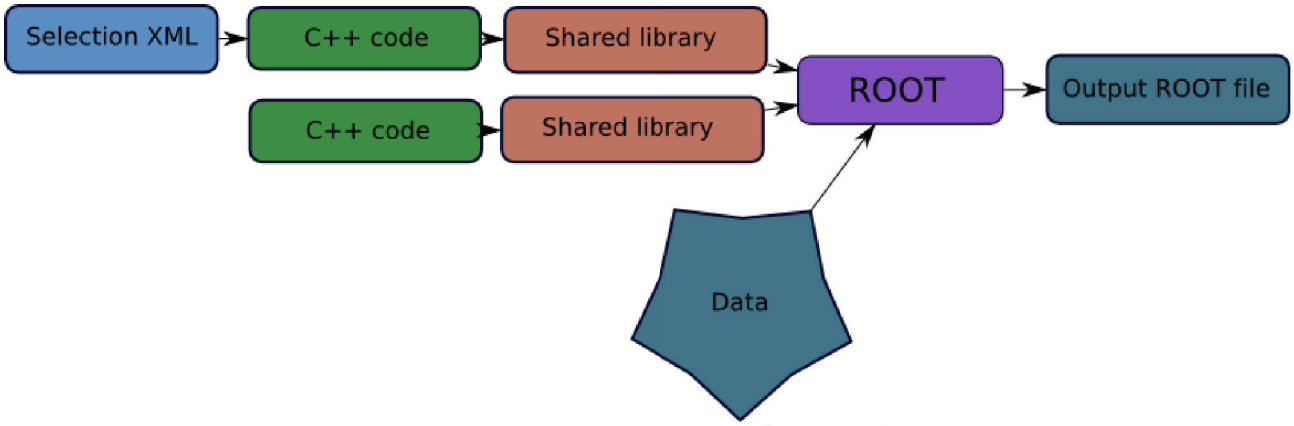
ID	Description
UC #1	**Assigning value to transient data member** A data member in in-memory object does not correspond to any persistent data member. It is to be set to some predefined value.
UC #2	**Changing the name of the class** The system should attempt load data into memory even when the names of in-

	memory and on-disk class differ
UC #3	**Changing the name of the data member** The name of the data member changed but the type remained the same or is convertible
UC #4	**Change of the class shape** The class shape has changed, or one class is converted to another in such a way that data members of in-memory class can be processed as a combination of data members of on-disk class.
UC #5	**A class was converted to collection and vice versa** We convert the collection to normal class having associated streamer info, possibly containing a collection of collections.
UC #6	**The way the object was streamed changed in such a way that there is a need to access the buffer data directly** The way ROOT handles arrays changed at some point in the past, but the old data must remain readable

2. Design proposal

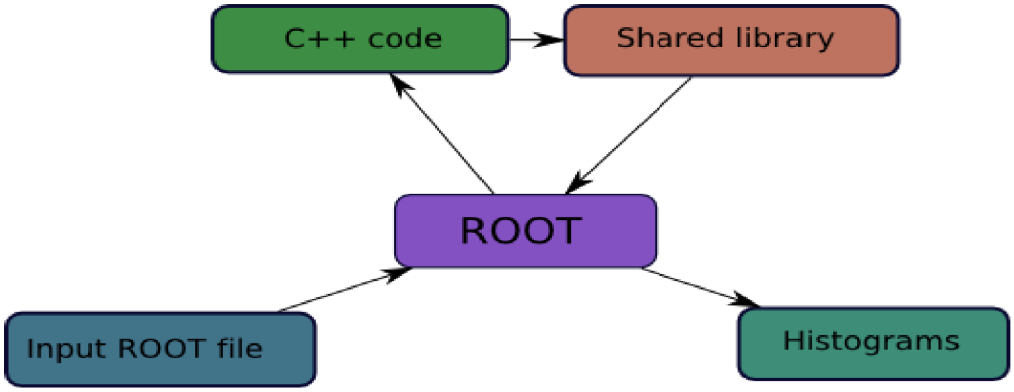
2.1 Definition of the conversion rules

There will be two possible ways of specifying the conversion rules. The user must be able to provide the rules as strings containing, among other indispensable information, the code snippets that should be compiled. The string format should be uniform for both CINT and REFLEX dictionaries. The other way is to register a pointer to a function that should be called while reading old data. The rules provided as strings should be processed and compiled at dictionary generation stage and the resulting function pointers should be registered to the system.



Drawing 2: Creating a root file with embedded schema evolution information

The rules that do not contain any references to user's code should be persistified to the files so that they could be processed, compiled and used in bare ROOT mode.



Drawing 3: Processing rules stored in ROOT file

2.1.1 Granularity and persistency of the conversion rules

To efficiently handle multiple storage modes and the ability to store the rules themselves to the files we need to face and solve some granularity problems, it is whether one rule should be specified for one data member or for entire object being read and answer the questions how to handle those rules when they are read back from the files. We have the legitimate use cases for both object-wise and member-wise rules.

Member-wise rules, where the target is a pointer to a data member of basic type of the object being read, are the most natural way of applying transformations while handling data stored in the split mode because we compute only the data member that is required at a given moment. On the other hand we may have to perform the same time/memory expensive operation to calculate the value of few correlated data members, in this case it would be good to be able to do that once and reuse the result, hence to have access

to the entire target object.

On the other hand we may need to call some methods of the target object in order to recreate it. To do that, naturally, we need access to the object pointer (hence to be able to define object-wise rule) and we need to have the user code. The fact that in object wise rules user can do both, data members setting and calling user code makes it difficult to persistify and handle them in bare ROOT mode because the user specified code segments are wrapped into function calls and combined together, so the messages indicating the compilation errors may be cryptic and incomprehensible. And also this limits the desired functionality of reading data without any user code.

In the first prototype we have decided to implement the version with the member-wise rules being persistified and object-wise rules being usable with the user code. If this proves insufficient or not memory/CPU effective enough we'll attempt to refine it to use object-wise rules also in bare ROOT mode.

2.1.2 Code segments in the rules

The user specified code snippets for the rules will be parsed and wrapped into functions before being compiled. For the purpose of a rule creation, user can assume that there are several variables defined for him:

- newObj - for object-wise rules if the user code is available; of the type of target object
- names of the data members of the in-memory object being read - the ones of basic type if the user code is not present or all of them otherwise
- oldObj - TObjectProxy object (described later in this document representing the data read from the input file
- buffer - for raw rules

2.1.3 Source version information format

The version string consists of comma-separated list of tokens enclosed in square brackets. The tokens can be specified in the following way:

x	the rule should be applied to version x
x-y	the rule should be applied to version greater or equal to x and smaller or equal to y
-y	the rule should be applied to all versions smaller or equal to y
x-	the rule should be applied to all versions greater than x

2.1.4 Raw reading rules

The raw reading rules are intended to support the use cases for which the layout of the data in the buffer is different than the one expected by the streamer element that is supposed to represent it, ie. the way the arrays are physically stored in the buffer changed some time ago and to read them back we need to perform some operations directly on the buffer.

The syntax of the rule as specified in the LinkDef file:

```
#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 rule as specified in the selection.xml:

```
<readraw>
<![CDATA[
   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 }"
]]>
<readraw>
```

source	name of source data member for which the rule should be applied
version	version of the input class which the rule should be applied for
checksum	checksum of the input class which the rile should be applied for
target	target data member (for member-wise rule) or target object (for object-wise rule), specified only if different than the source
embed	boolean flag determining if the rule should be embedded in a file, false by default
include	include files that should be attached while compiling the rule
code	C++ code snippet performing the conversion enclosed in curly brackets or C++ function enclosed in square brackets

Example:

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

2.1.5 Normal reading rules

The rules described in this section are meant to handle the usual cases of schema evolution. The syntax for the rule for LinkDef:

```
#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 }"
```

The syntax of the rule as specified in selection.xml:

```
<read>
<![CDATA[
  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 }"
]]>
<read>
```

source	list of data member of source object that sould be loaded, and be accessible via proxy, the class name if we need to load all of them
version	version of the input class which the rule should be applied for
checksum	checksum of the input class which the rule should be applied for
target	list of target data members that are being set by the rule or class name if the rule calls a method of newObj (for member-wise rule) or target object (for object-wise rule)
targetType	type of the target object, in case when we cannot get this information from elsewhere (when the dictionary is missing)
embed	boolean flag determining if the rule should be embedded in a file, false by default
include	include files that should be attached while compiling the rule
code	C++ code snippet performing the conversion enclosed in curly brackets or C++ function enclosed in square brackets

Example:

```
#pragma read target="TH1::fDirectory;TH1" include="<TDirectory.h> <TROOT.h>
<TH1.h>" code="
{
  if( TH1::AddDirectory() && !gROOT->ReadingObject())
  {
    fDirecotry = gDirectory;
    if( !gDirectory->GetList()->FindObject( newObj ) )
    {
      gDirectory->Append( this );
    }
  }
}
```

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

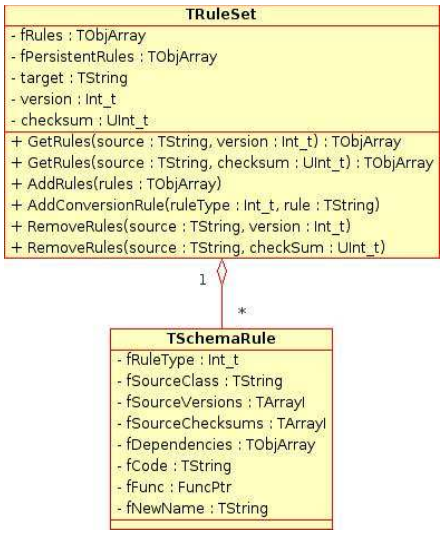
2.2 Means of storing and accessing the rules

2.2.1 Backward and forward compatibility issues

While storing a rule in ROOT file we need to make sure that new versions of ROOT are still able to read old files (backward-compatibility) and that old versions of ROOT are still able to read new files (forward-compatibility). To do that we'll put the structure (TRuleSet, described below) holding the conversion rules into the list that now holds the streamer info objects. When new version of ROOT reads old file it will just assume that no schema evolution information is present, and while the old version reads new file then it will ignore unknown objects found in the streamer info list and treat the file on the way it did so far.

2.2.2 Structure holding the conversion rules

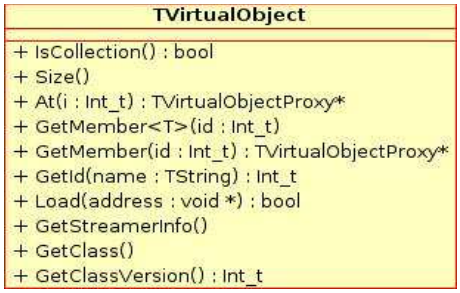
The structure holding the conversion rules, called TRuleSet, must provide a functionality for efficient querying for a set of rules given the name and version (or checksum) of the source class. We will have one TRuleSet object per every target class, target version pair.



Drawing 4: UML diagram of the TRuleSet structure

2.3 Object proxy

The proxy object is meant to represent on-disk data temporarily loaded into memory that is required to perform the transformations specified by the conversion rules. It should allocate enough memory to contain all the information represented by the streamer info object associated with the source class. Note that not all of the data members will be loaded to the proxy structure while performing the conversion. We can guarantee that only those specified by the user in the source tag of the rule will be present while executing the conversion function associated with the rule.

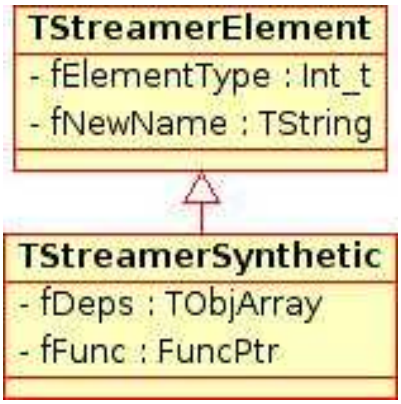


Drawing 5: The proxy object

IsCollection **Size** **At**	methods meant to determine if the object is a collection and to access the elements if that is the case
GetMember	data members accessors
GetID	returns the Id of given data member (to avoid doing string comparison while accessing proxified data)
Load	loads data contained by the proxy to in-memory class applying the appropriate conversion rules specified by the user
GetStreamerInfo **GetClass** **GetClassVersion**	return various information about the class represented by the proxy

2.4 Updates to streamer info structures

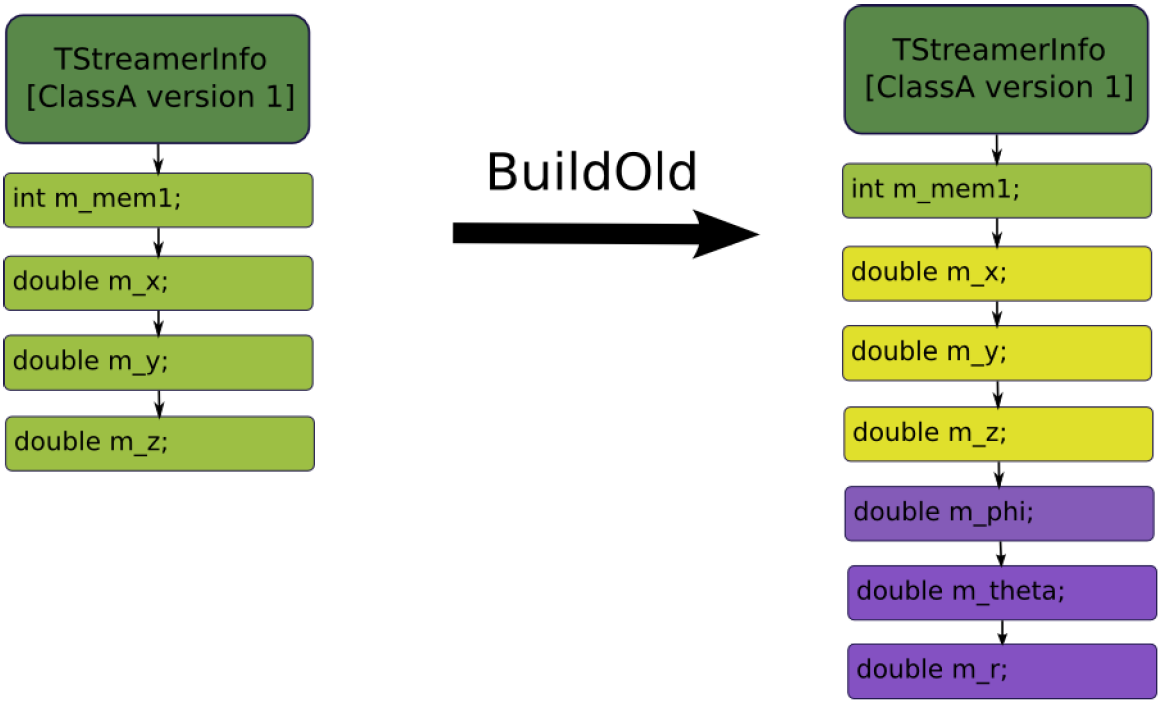
While processing the conversion rules the schema evolution engine will compile them and insert new, artificial, streamer elements to appropriate streamer info structures. Those streamer elements will be then processed by the reading facilities which will take the appropriate action. To handle this effectively we have to update the streamer element structure and add one additional concrete streamer element. TStreamerSynthetic is supposed to be associated with transient data member that is not represented directly with any persistent information but it's value may be computed using other available information.



Drawing 6: New data members in StreamerElement and new concrete streamer element class

fElementType	Provides information about the action that should be performed while processing this streamer element. It would be assigned to one of the following values depending on the operation that should be performed on information associated with it: **kDISABLED** - represents the data member that has been persistified but there is no need for this information any more, to be skipped during processing, **kBUFFERED** - the persistent information should be cached for further use in one of the conversion functions, **kREAL** - the persistent information should be put into right place into memory area occupied by the transient object, **kSYNTHETIC** - not associated with any persistent information but representing transient data member which should be assigned to the value computed by the transformation function.
fNewName	Used when the name of the data member was changed (UC#3). Of course this situation could be resolved easily by specifying the "identity" transformation function taking information associated with different streamer element marked as buffered, but this has time and memory penalties.

fDeps list of the streamer elements that this one depends on
fFunc pointer to the conversion function



Drawing 7: Modification of the streamer info object while loading ClassA v1 into in-memory ClassA v3 (see Drawing 1)

The above drawing shows the change of streamer info structure for ClassA v1 while loading into in-memory ClassA v3 (see Drawing 1). The green element representing the m_mem1 is marked as real, because it's representation did not change and can be loaded directly into memory. The yellow elements representing m_x, m_y and m_z are marked as buffer since they are not present in transient class but they are used to derive some necessary information. The purple elements representing m_phi, m_theta, m_r are syntetic elements, they are not associated with any persistent data, but they represent transient data members. While processing those elements their conversion functions are being called and the result is being copied into a memory area occupied by the transient object.

2.5 Updates to TTree structures

We need to take some special provisions while handling data in split mode, where every branch has a streamer element associated with it and delegates the the job of reading the actual data to the streamer info structure pointing it to the appropriate buffer. Since we add new streamer elements we also need to add new branches of special type that will call the conversion functions. This kind of approach makes it easier to integrate the new functionality with already existing components of ROOT.