ROOT Data Model Evolution

Lukasz Janyst, Rene Brun, Philippe Canal
# Table of Contents

1. Introduction...........................................................................................................3  
   1.1 Self description capabilities of the ROOT files..............................................3  
   1.2 Currently implemented functionality and it's limitations...........................3  
   1.3 Problem definition.........................................................................................3  
   1.4 User interface and scalability requirements.................................................4  
   1.5 Use cases........................................................................................................4  

2. Design proposal....................................................................................................5  
   2.1 Conversions on member-wise level...............................................................5  
   2.2 Conversions on object-wise level....................................................................6  
   2.3 Dealing with situations where class name changes.......................................8  
   2.4 Conversion rules as strings...........................................................................8  
   2.5 Handling conversion rules.............................................................................8  

3. Examples.................................................................................................................9
1. Introduction

This short document aims to describe briefly the user interface and 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 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 streaming, however, the schema evolution abilities of the system are very 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 uniform interface.

1.3 Problem definition

![Diagram](image)

Drawing 1: Graphical representation 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 laid out in memory in different way, i.e. 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 that the one that has been used to write them. The solution is has to work in both object-wise and member-

### 1.3 User interface and scalability requirements

We must also 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. While reading the user will provide the system with the number of desired version and the conversion will be done on the fly.

### 1.4 Use cases

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC #1</td>
<td>Assigning value to transient data member</td>
</tr>
<tr>
<td></td>
<td>A data member in in-memory object does not correspond to any persistent data member. It is to be set to some predefined value.</td>
</tr>
<tr>
<td>UC #2</td>
<td>Changing the name of the class</td>
</tr>
<tr>
<td></td>
<td>The system should attempt load data into memory even when the names of in-memory and on-disk class differ</td>
</tr>
<tr>
<td>UC #3</td>
<td>Changing the name of the data member</td>
</tr>
<tr>
<td></td>
<td>The name of the data member changed but the type remained the same or is convertible</td>
</tr>
<tr>
<td>UC #4</td>
<td>Change of the class shape (simple case)</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>UC #5</td>
<td>Change of the class shape (complicated case)</td>
</tr>
<tr>
<td></td>
<td>The layout of the class changed significantly and the loading cannot be handled any more by UC #4</td>
</tr>
<tr>
<td>UC #6</td>
<td>A class was converted to collection and vice versa</td>
</tr>
<tr>
<td></td>
<td>We convert the collection to normal class having associated streamer info, possibly containing a collection or collections. Reconstruct objects if possible, put proxy if impossible.</td>
</tr>
</tbody>
</table>
2. Design proposal

2.1 Conversions on member-wise level

Assigning values of transient members (UC#1) and carrying out member-wise conversions (UC#4) is done using "artificial" streamer elements added to streamer info objects. The word artificial means that those streamer element is not directly associated with any information stored in buffers but it will pass this information to a supplied conversion function and assign the result to data member of transient object associated to the streamer element. Assigning a value to transient data member is just a special case of this functionality where the conversion function does not need any input parameters to compute the result.

This approach requires only a slight modification to the streamer element class and introduction of one new concrete type of streamer element.

![Drawing 2: New data fields in TStreamerElement and new concrete streamer element class](image)

fElementType member of TStreamerElement class is supposed to provide 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 data member is 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.

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.

fDeps member is a list of the streamer elements that this one depends on and provides the information about the order in which the input parameters should be passed to the conversion function.

fFunc is a conversion pointer to the conversion function defined in the following way:
Target parameter is an address of the data member that should be assigned to some value and params parameter is a vector of addresses where the buffered parameters can be found. Params vector is ordered in the same way as fDeps array which in turn reflects the order specified by the user (described later in this document).

fValue member is used to assign a predefined value to transient data member. Again, similarly to the name change handling, this case could be dealt with using a conversion function taking zero parameters and returning a constant, but this approach has time and memory penalties.

While reading the persistent information stored in split mode we do not loop over the streamer elements but over the branches so some part of this functionality has to be doubled in the code of TBranchElement.

```cpp
void func( void *target, const std::vector<const void *> &params )
{
}
```

![Diagram](image)

**Drawing 3: 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 v2 (see Drawing 1). The green element representing the m_mem1 is marked as real, because its 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 processing 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.

### 2.2 Conversions on object-wise level

The approach described above cannot be used when the difference between transient and persistent information is substantial, i.e. when transient classes have completely different shape then the persistent ones, but they contain logically the same information. To handle this issue the conversion has to be performed on the object level. This raises the problem of accessing data members of transient objects,
because in most cases they are private. The problem can be overcome however by accessing the data members using proxy objects because all required information is provided by introspection mechanisms and streamer infos.

**TVirtualObjectProxy**

- IsCollection() : bool
- Size()
- At(int i) : TVirtualObjectProxy*
- IsEmulated() : bool
- GetObject<T>(int i) : TVirtualObjectProxy*
- GetObject(id : int i) : TVirtualObjectProxy*
- GetId(name : TString) : int i
- Load(address : void *) : bool
- GetAddress() : void *
- GetStreamerInfo()
- GetClass()

**Drawing 4: Proxy interface for accessing transient and persistent objects during objectwise conversion**

TVirtualObjectProxy is an interface for accessing data members objects that should be implemented by several classes optimizing access to underlying data, which can be a transient object, an "emulated" object reconstructed in memory basing on information from streamer infos or branches in split mode, or the data stored directly in the buffer.

First three methods are meant to be used when the object is a collection, since in this case we do not really care about the data members of the collection class but about the access to particular data members.

IsEmulated method determines if the underlying object has an user class associated with it or if it is emulated in any way.

GetObject methods return object's constituents either as a reference or const reference or pointer to (const) TVirtualObjectProxy object representing given data member or base class. The parameter passed to those methods is an identifier that enables user to retrieve the information efficiently. It can be an offset, an index to streamer element or whatever is the most convenient for given implementation.

GetId method takes a string description (a data member name or a base class name) and generates the ID to be used by GetObject.

Load method tries to load the data represented by the proxy to given memory address according to whatever schema evolution rules different that object-wise conversion.

GetAddress method returns an address of the underlying object is it is placed in memory.

GetStreamerInfo and GetClass methods return information about the streamer info and the TClass object associated to underlying data if this kind of information is available.

The objects of this type are passed to the conversion function defined in the following way:

```cpp
void func( TVirtualObjectProxy *target, const TVirtualObjectProxy *source )
{
}
```
2.3 Dealing with situations where class name changes

The system should always try to match the data member names even if the class names of persistent and transient classes do not match and use the functionality described in section 2.1.

2.4 Conversion rules as strings

User can supply the system with the conversion rules in the form of strings containing chunks of code, that are processed and compiled using ROOT's built-in on-demand compiler. Processing means converting the rules to the form of function described in section 2.1. Conversions provided this way cannot depend on user's custom code.

2.5 Handling conversion rules

The schema evolution rules should be handled together with TClass objects. This section describes a helper class named TSchemaRuleSet that manages the conversion rules and the converters. An object of this class is owned by every TClass object.

Drawing 5: User interface for handling conversion rules

Every object of TSchemaRuleSet class contains three arrays of objects: fRules, fPersistentRules and fConvertors. The first two arrays contain TSchemaRule objects, one of those arrays contains only the rules that should be attached to TStreamerInfo object and persistified to a file. fConvertors array contains TSchemaConvertor objects.

TSchemaRule contains the following information:
- fSourceClass - name of the persistent class that the rule applies to
- fSource versions and fSourceChecksums - versions and checksums of persistent class that the rule applies to.
fDependencies - list of string names of the persistent data members this rule depends on
fTarget - name of the data member in transient class this rule should be applied to
fCode - string code representation if applicable
fFunc - function pointer in the format described in section 2.1 obtained either from user or as a result of processing and compiling contents of fCode
fValue - a value that should be assigned to transient data member - for the case we know it
fValueSize - size in bytes of the above, for persistency reasons

TSchemaConvertor contains the following information:
fSource, fVersion, fChecksum - the same as for TSchemaRule
fFunc - pointer to the conversion function in format described in section 2.2

The TSchemaRuleSet objects have to be able to provide the conversion rules for given class name, version or class name, checksum pairs in time efficient way.

2.6 The ways of supplying the system with conversion rules

The conversion rules can be supplied to the system in three different ways. Directly from the C++ code compiled into a shared library and from selection xml file converted during the compilation time to C++ code and then compiled. Some of the rules supplied this way can be associated with the streamer info objects and serialized together with them. While reading those files in bare ROOT mode, when no other schema evolution information is supplied, the system will parse and compile the serialized code chunks.

**Drawing 6:** An example showing processing stages of the conversion rules when supplied from user’s code

**Drawing 7:** An example showing how the conversion rules are supplied from ROOT file while running in bare ROOT mode
3. Examples

3.1 Member-wise conversion

Conversion rule specification in C++ code:

```cpp
TClass *cl = TClass :: GetClass( "ClassA" );
TSchemaRuleSet = cl->GetSchemaRuleSet();
cl->AddConversionRule("m_phi", "ClassA", "[1,2]", "m_x,m_y,m_z",
                    "arctan(m_y/m_x)" )
cl->AddConversionRule("m_theta", "ClassA", "[1,2]", "m_x,m_y,m_z",
                    "arctan(sqrt(m_x*m_x+m_y*m_y)/m_z)" )
cl->AddConversionRule("m_r", "ClassA", "[1,2]", "m_x,m_y,m_z",
                    "sqrt(m_x*m_x+m_y*m_y+m_z*m_z)" )
```

Conversion rule specification in selection.xml:

```xml
<class name="ClassA">
  <conversion target="m_phi" sourceClass="ClassA" version="[1,2]"
               dependencies="m_x,m_y,m_z">
    arctan(m_y/m_x)
  </conversion>
  <conversion target="m_theta" sourceClass="ClassA" version="[1,2]"
              dependencies="m_x,m_y,m_z">
    arctan(sqrt(m_x*m_x+m_y*m_y)/m_z)
  </conversion>
  <conversion target="m_r" sourceClass="ClassA" version="[1,2]"
              dependencies="m_x,m_y,m_z">
    sqrt(m_x*m_x+m_y*m_y+m_z*m_z)
  </conversion>
</class>
```