# Support for significant evolutions of the user data model in ROOT files



2010 J. Phys.: Conf. Ser. 219 032004

(http://iopscience.iop.org/1742-6596/219/3/032004)

Journal of Physics: Conference Series **219** (2010) 032004 doi:10.1088/1742-6596/219/3/032004

# Support For Significant Evolutions of the User Data Model In ROOT Files

**Ph. Canal[1a], R. Brun[2], V. Fine[3], L. Janyst[2], J. Lauret[3], P. Russo[1]**

[1] Fermilab, Batavia, IL, United States of America

[2] CERN, Geneva, Switzerland

[3] Brookhaven National Laboratory, Upton, NY, United States of America

[a] E-mail: pcanal@fnal.gov

**Abstract**. One of the main strengths of ROOT input and output (I/O) is its inherent support for schema evolution. Two distinct modes are supported, one manual via a hand coded streamer function and one fully automatic via the ROOT StreamerInfo. One draw back of the streamer functions is that they are not usable by TTree objects in split mode. Until now, the user could not customize the automatic schema evolution mechanism and the only mechanism to go beyond the default rules was to revert to using the streamer function. In ROOT 5.22/00, we introduced a new mechanism which allows user provided extensions of the automatic schema evolution that can be used in object-wise, member-wise and split modes. This paper will describe the many possibilities ranging from the simple assignment of transient members to the complex reorganization of the user's object model.

## 1. Introduction

One of the strengths of the ***ROOT*** Input/Output package is its support for schema evolution. Two distinct modes have been supported so far, one manual via a hand coded streamer function and one fully automatic via the ROOT StreamerInfo. The fully automated simple schema evolution lets the developers trivially make small changes to their data model. For example a data member that used to be a float can become a double while still being able to be read from old files. The second technique requires the developer to write a C++ function handling explicitly the schema evolution; we call this technique "hand coded schema evolution." This technique offers almost complete freedom to the developer but requires specific coding for each variant of the class layout. It is also a bit awkward for very complex class layout reshuffling, for example in the case of moving the value of a data member to or from a sub-object.

***ROOT*** I/O now supports a third mechanism for schema evolution. This paper covers in details this new technique: "Customizable Automatic Schema Evolution". This technique expands on the existing infrastructure developed for the fully automated simple schema evolution and opens it to

 

user provided customization. It allows the implementation of very complex schema evolution of the class layouts in a simple and efficient manner.

## 2. Historical Perspective

***ROOT*** is a C++ framework for data processing, created at CERN, at the heart of the research on High-Energy and Nuclear Physics (HENP). Every day, thousands of physicists use ***ROOT*** based applications to analyze and visualize their data. ***ROOT*** started in 1995 [1]. It grew from a private project of two developers to the officially supported LHC analysis toolkit. It is currently developed by a small team with members from several laboratories across the world.

One of the core functionalities of ***ROOT*** since it inception is its ability to read and write large amount of data. One of the initial requirements is the ability to read back data that has been written several years back within impeding change in the user's class layout. And thus ***ROOT*** has been supporting schema evolution since its very first release.

### 2.1. Early days

At first, the I/O was driven by streamer class functions that were to be written explicitly by the user for each class. The purpose of the streamer functions is to iterate through each of the class members and to read or write their values to an I/O buffer. When the I/O buffer has been filled, it is optionally compressed and it is written to the ***ROOT*** file on disk.

Relying on the user to explicitly write these streamer functions was a very redundant, labor intensive and error prone practice. As early as release version 0.9 of ***ROOT***, we introduced the ability to automatically generate these streamer functions using a new utility called **rootcint**. **rootcint** relies on a C++ interpreter (CINT [2][3][4]) to be able to generate a dictionary file that describes the class layouts being used. Using these descriptions, **rootcint** is also able to generate the streamer function (See Figure 1).

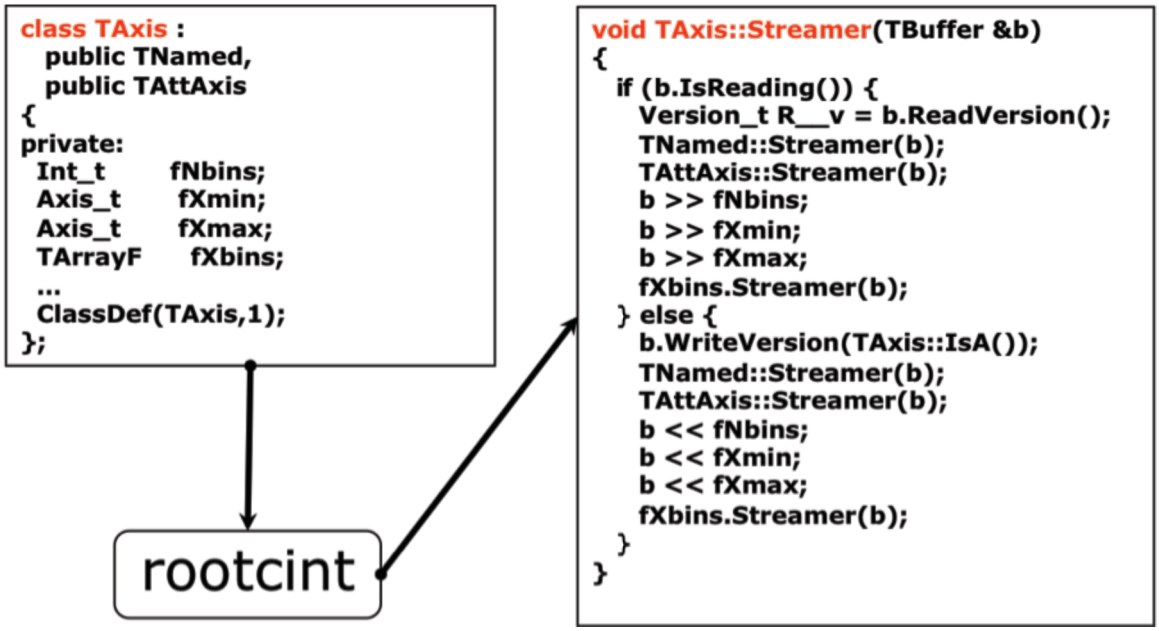


Figure 1: Streamers in ***ROOT*** version 0.90

To be able to support reading a file written with an older class layout, the streamer stores along side the data a class version number. This class version number can be used to write code handling schema evolution. For example, in Figure 2, the class *TAxis* was modified to add 2 new data

members. To support reading both the old and new class layout the user had to write explicit code; See Figure 2, in particular the code starting at "*if (R__v > 3)*".

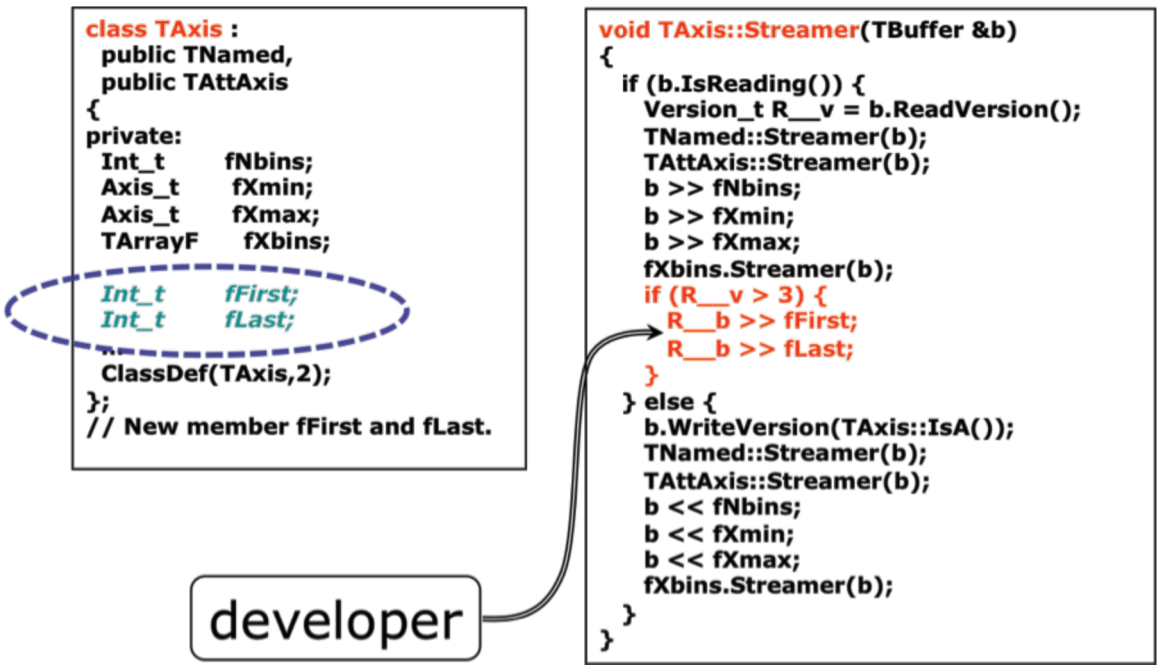


Figure 2: Hand coded schema evolution

### 2.2. Automatic Schema Evolution

With the increasing number of legacy class layouts to support, manually maintained streamer functions become quickly extremely complex and hence more vulnerable to the introduction of deficiencies. In addition any seemingly minor changes to the class layout, like adding a new member or changing the type of a member from *short* to *int*, requires manual attention.

In order to alleviate these issues, we introduced a new streaming mechanism that leverages the availability of the class layout dictionary. The new mechanism relies on StreamerInfo objects that describe the name and type of the data member to be stored for one specific class. These StreamerInfo objects are stored along side the data in the ***ROOT*** file.

Thanks to the addition of these StreamerInfo objects in the file, the ***ROOT*** file became self-describing and support for forward compatibility and reading without the original shared library became possible. Forward compatibility is the ability to read with an older version of the software a file that was written with a newer version of the software.

The self-describing capability also allows other frameworks [7] to be able to read ***ROOT*** files since they can interrogate the file to learn how to interpret the file's content.

When loading a StreamerInfo description from a file, the framework can lookup the memory offsets of the data members by simply matching the data member names from the StreamerInfo with the data member names lists in the dictionary. This is however only meaningful 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 the data members differ between disk and memory and the type is a basic type then the framework will attempt to do the conversion.

### 2.3. Object-wise and member-wise streaming

***ROOT***'s *TTree* is a collection that is optimized also for I/O. When a branch containing objects of a given class is split, the description of the class' members is used to create sub-branches. This can be

done recursively. This storage scheme is called a member-wise or column-wise layout (CWS; Figure 3), as opposed to object-wise or row-wise storage (RWS) that is usually found in databases.

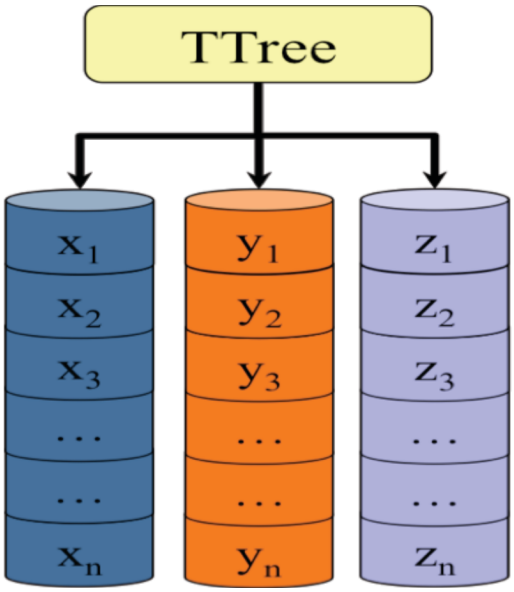


Figure 3: Column-wise storage

In CWS, just like in RWS, a collection ("table") of similar objects ("rows") is assumed. However, RWS stores all data members of the first object, then all those of the second object, and so on, whereas CWS stores the first data member of all objects consecutively, then the second data member of all objects, etc. CWS allows reducing the input operations and the amount of transferred data, because it makes it possible to only read the needed parts of each object. All other members of the object keep the values defined by the class default constructor. When iterating through the collection, data members that need to be read are consecutive on the storage medium in the case of CWS. This allows block-wise reading of the data for several entries in one go, something massively favored by all modern operating systems and storage media.

### 2.4. Limitation of the automatic schema evolution

Thanks to the StreamerInfo, ***ROOT*** is able to automatically support simple class layout changes, in particular:

- Changing the order of the members
- Changing simple data type (*float* to *int*)
- Adding or removing data members, base classes
- Migrating a member to base class

However it did not allow for any customization beyond these simple transformations. In particular it did not support change in the type of a data member from or to a class type and did not support change in semantic of the member (for example change in units). Whenever such changes were required the developer had to revert to using a manually maintained streamer function. However, those functions cannot be used during member-wise streaming.

## 3. Customizable Automatic Schema Evolution

In order to lift these restrictions while leveraging the existing infrastructure, we introduced a mechanism to customize the StreamerInfo.

### 3.1. Description

The mains goal of this new customizing framework for the automatic schema evolution is to support the following class layout changes:

- Assign values to transient data members
- Rename classes
- Rename data members
- Change in the hierarchy within the data structures or convert one class structure to another
- Change the semantics of data members (for example change in units)
- Allow accessing the *TBuffer* directly when needed
- Ensure that the objects in collections are handled in the same way as the ones stored separately
- Transform data before writing

The implementation also needs to support the use of the customization even without access to the original user shared library (‘*bare **ROOT**’.)*

Users can now supply a function to convert individual data members from disk to memory and a rule defining when to apply the transformation.

A schema evolution rule is composed of:

- ***sourceClass***, ***version***, ***checksum***: identifier of the on disk class
- ***targetClass***: name of the class in memory
- ***source***: list of type and name of the on disk data members needed for the rule
- ***target:*** list of in memory data members modified by the rule
- ***include***: list of header files needed to compile the conversion function
- ***code***: function or code snippet to be executed for the rule

Rules can be registered either via a LinkDef.h file, a **XML** selection file, direct calls to the C++ API or in a ***ROOT*** file. At the time of writing, a class layout can be given an easy to use version number by either using the ClassDef macro with the class definition or using the RootClassVersion macro outside of the class definition. When a class version number is not provided the way only to refer to a class layout is to use the ‘checksum’ that ROOT calculate for it based on the data member name and type that it contains.

### 3.2. Examples

When using **rootcint** to generate the dictionary, the LinkDef.h file can be instrumented with rules like:

```
#pragma read sourceClass=”oldname” version="[1-]" checksum=“[12345,23456]” \
    source=”type1 var; type2 var2" \
    targetClass=“newname” target=”var3" \
    include=“<cmath> <myhelper>” \
    code=”{ … ‘code calculating var3 from var1 and var2’ … }"
```

When using **genreflex** to generate the dictionary, the **XML** selection can be instrumented with rules like:

```
<read sourceClass=”oldname” version="[4-5,7,9,12-]” checksum="[12345,123456]”
        source=”type1 var; type2 var2”
        targetClass=”newname” target=”var3”
        include=“<cmath> <myhelper>”
<![CDATA[
  … ‘code calculating var3 from var1 and var2’ …
]]> </read>
```

A similar string based interface is also available in C++. In all 3 cases, the code to be executed can be passed either explicitly in the string argument or via a function name or pointer.

3.2.1. *Setting a transient member*

One of the main weaknesses of the fully automated schema evolution is its handling of transient members. When streaming the information from disk into a previously used object (most commonly used pattern when reading information from a *TTree*), the transient members were untouched and retained their previous values. For example:

*MyClass.h*

```
class MyClass {
private:
   Type fComplexData;
   Double_t fValue; //! Calculated from fComplexData
   Bool_t fCached;  //! True if fValue has been calculated
public:
   double GetValue() { if (!fCached) { fValue = ... ; }; return fValue; }
};
```

In the past when re-using an object of type *MyClass*, the value of the data members '*fValue*' and '*fCached*' would be unchanged. Consequently a call to *GetValue* would return the information from the previous incarnation of the object rather than the newly loaded information. Using the new schema evolution rules mechanism, for example with:

*MyClassLinkDef.h*

```
#pragma read sourceClass="MyClass" version="[1-]" source=""
  targetClass="MyClass" \
  target="fCached" \
  code="{ fCached = false; }"
```

the value of '*fCached*' will now be updated using the provided code every time an object of type MyClass is being read into memory. In this example, ***sourceClass***="*MyClass*" indicates that this rule apply to objects of type *MyClass*. The text ***version***="[1-]" indicates that this rule applies to all revisions of the class layout; ***source***="" indicates that the calculation of 'fCached' does not require any input; ***target***="*fCached*" indicates that this rule will update the value of the data member named '*fCached*' and thus needs to be executed whenever the data member *fCached* might be accessed.

3.2.2. *Merging several data members*

A classical example of complex schema evolution is the case where two or more data members are merged together in order to save space in memory. For example:

```
class MyClass {
private:
   int fX;
   int fY; // Values between 0 and 99
   int fZ; // Values between 0 and 9
public:
   int GetX() { return fX; }
   int GetY() { return fY; }
   ClassDef(MyClass,8);
};
```

```
class MyClass {
private:
   long fValues; // Merging of fX, fY and fZ
public:
   int GetX() { return fValues / 1000; }
   int GetY() { return (fValues%1000)/10; }
   int GetZ() { return fValues % 10; }
   ClassDef(MyClass,9);
};
```

Without the new infrastructure, supporting this case required the implementation of a streamer function that would read explicitly the values of *fX*, *fY* and *fZ* into temporary variables and assign the merged value into the data member *fValues*. Because the streamer function can only extract information from a single buffer, the streamer function cannot be used when splitting a TTree or during member-wise streaming. Now, the following schema evolution rule:

```
#pragma read sourceClass="MyClass" version="[8]" targetClass="MyClass " \
  source="int fX; int fY; int fZ" target="fValues" \
  code="{ fValues = onfile.fX*1000 + onfile.fY*10 + onfile.fZ; }"
```
*MyClassLinkDef.h*

allows for the seamless support for both the separated and the merged class layout even when the object has been split or streamed member-wise. In this example, ***sourceClass***="*MyClass*" indicates that this rule applies to objects of type *MyClass*. The text ***version***="[8]" indicates that this rule applies only when reading object written with the 8th class layout of the class *MyClass*. ***target***="*fValues*" indicates that this rule will update the value of the data member named '*fValues*' and thus needs to be executed whenever the data member *fValues* might be accessed.

The text ***source***="*int fX; int fY; int fZ*" indicates the types and names of the original members and tells the I/O system which members will need to be read from the input before executing the rule. The types are necessary in order for the rule to be precompiled even though it does not have direct access to the exact shape of the 8th layout of the class *MyClass* since this information is not in the header file (however it is stored in the ***ROOT*** files and thus if the types are not provided the rule can be compiled just in time when actually reading the file).

In the ***code="..."*** section, the notation *onfile.Varname* gives direct access to the input value of the data member named '*Varname*' as extracted from the *TBuffer* before it undergoes any conversion. The variable *onfile* is a placeholder containing all the data members that have been listed in the ***source="..."*** section and is filled before the rule is run.

### 3.2.3. *Renaming a class*

One of the most requested new features for the schema evolution has been the ability to rename classes. In this example we show not only how to rename a class but also how to change the class layout at the same time.

```
class MyClass {
private:
  int fX;
  int fY; // Values between 0 and 99
  int fZ; // Values between 0 and 9
public:
  int GetX() { return fX; }
  int GetY() { return fY; }
  ClassDef(MyClass,8);
};
```

```
class Properties {
private:
  long fValues; // Merging of fX, fY and fZ

public:
  int GetX() { return fValues / 1000; }
  int GetY() { return (fValues%1000)/10; }
  int GetZ() { return fValues % 10; }
  ClassDef(Properties,2);
};
```

```
#pragma read sourceClass="MyClass" version="[9]" targetClass="Properties"
#pragma read sourceClass="MyClass" version="[8]" targetClass="Properties" \
    source="int fX; int fY; int fZ" target="fValues" \
    code="{ fValues = onfile.fX*1000 + onfile.fY*10 + onfile.fZ; }"
```
*MyClassLinkDef.h*

The first rule simply indicates that when attempting to read an object of type *MyClass* (the ***sourceClass***) with the 9th layout of the class *MyClass* (***version***="[9]") into an object of type *Properties*, the I/O system should succeed and apply the automatic schema evolution rules.

The second rule indicates that when reading the 8th layout of the class *MyClass* (***version***="[8]") into an object of type *Properties*, the I/O system should also apply the rules merging *fX*, *fY* and *fZ* into *fValues* (see 3.2.2).

### 3.2.4. *Nested Objects*

The new infrastructure also opens up new possibilities in term of schema evolution. A streamer function only has access to the version number of the class layout for the current object being read and can only (short of very complex coding) use sub-objects after they have been read from the input.

Let's take the example of a class named *Event* whose 2nd version contains an "extended *Track*" object. Over time, the layout of the class Event did not change but the "extended *Track*" went through many changes. Now we would like to modify the class *Event* (its 3rd version) where we replace the "extended *Track*" with a more compact version of the *Track* class, in addition, we are moving some of the information that was stored in the "extended *Track*" directly into the *Event* object.

```
#pragma read sourceClass="Event" version="[2]" targetClass="Event" \
   source="Track fTrack" target="fId; fCompactTrack" \
   code="{ if( onfile.fTrack->GetVersion() == 3 ) \
        { \
          fId = onfile.fTrack->GetMember<double>( id_fTrack_fB) + \
                onfile.fTrack->GetMember<double>( id_fTrack_fC ); \
          onfile.fTrack->Load( fCompactTrack ); \
        } \
        else if ( onfile.fTrack->GetVersion() == 4 ) \
        { \
           fId = onfile.fTrack->GetMember<double>( id_fTrack_fB); \
           onfile.fTrack->Load( fCompactTrack ); \
        };
   }"
```

In the code section, the old Track object can be accessed (via *onfile.fTrack*) as it was when it was written. This is used to extract the *fB* and *fC* data members. Then the registered (as well as the automatic) schema evolution rules are used to load the "extended *Track*" into the more compact representation (*onfile.fTrack->Load( fCompactTrack )* ).

## 3.3. Forward and backward compatibility of analysis code

The new customizable schema evolution rules can enable the re-use of an old script to read new data. For example, let's take the case where our collaboration writes a class *MyClass* that contains *fPx*, *fPy*, *fPz* data members expressing some location in x,y,z coordinates and writes a ***ROOT*** file (t1.root) using this class layout. We can then develop a script (work1.C) relying on this class layout. Later on, our collaboration upgrades the class *MyClass* to use ρ,θ,φ coordinate (*fR*,*fT*,*fP*) and writes new file (t2.root) using this new class layout.

If we want to be able to process this new file, using the old infrastructure we had no choice but to upgrade our script to be able to handle the new class layout (and rely on a streamer function

written by the collaboration). With the new customizable schema evolution rule, we now have two more options:

- We can write a backward compatibility rule that transforms the *fPx*,*fPy*,*fPz* layout into the *fR*,*fT*,*fP* layout and upgrade our script to use *fR*,*fT*,*fP*. Our new script would then be able to read both t1.root and t2.root (even if the collaboration did not provide a streamer function or a schema evolution rule).

- We could also write a rule that transforms the *fR*,*fT*,*fP* layout into the *fPx*,*fPy*,*fPz* layout and leave our script as is. With this forward compatibility rule, our old script would still be able to read the old t1.root file but also be able to read the new t2.root file.

This powerful feature allows for the long-term support of users' analysis packages involving macros even if the experimental software and I/O schema has changed.

## 4. Conclusion

The new customization framework significantly enhances the capabilities of the ***ROOT*** schema evolution infrastructure. It increases flexibility when reading old files, since it now supports complex schema when using member-wise streaming. Thanks to its dependency tracking it also allows for these complex rules to be run after reading only the minimal data set required. It supports both forward and backward compatibility. It even gives the possibility to perform complex evolution even without user classes since the rule definition can be saved in the ***ROOT*** file.

## Acknowledgements

This work was supported by the HENP Divisions of the Office of Science of the U.S. DOE.

## References

[1] R. Brun and F. Rademakers, "ROOT – An Object Oriented Data Analysis Framework", Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nuclear Instruments and Methods in Physics Research A 389 (1997) 81-86. See also http://root.cern.ch/
[2] M. Goto, "C++ Interpreter – CINT", CQ publishing, ISBN4-789-3085-3 (Japanese)
[3] M. Goto BEOS and ROOT, Interface Magazine September 1997, CQ publishing (Japanese)
[4] R. Brun and F. Rademakers, "ROOT: An object oriented data analysis framework", Linux Journal 1998 July Issue 51, Metro Link Inc.
[5] The ROOT Team, "ROOT User's Guide", http://root.cern.ch/drupal/content-users-guide
[6] A. Naumann and Ph. Canal, "The Role of Interpreters in High Performance computing", Proceedings of ACAT 2008, PoS(ACAT08)065, http://pos.sissa.it/archive/conference/070/065/ACAT08_065.pdf
[7] http://java.freehep.org/lib/freehep/doc/root/rootjas.html, retrieved April 13, 2009.
[8] J.W. Harris and the STAR Collaboration: The STAR Experiment at the Relativistic Heavy Ion Collider, Nucl. Phys A566 (1994) 277c.