Selected aspects of the persistent Event Data Model for ATLAS experiment at LHC

Łukasz Janyst

Supervisor:
prof. dr hab. Elżbieta Richter-Wąs

Reviewer:
dr hab. Piotr Białas

Consultant:
dr Markus Elsing (CERN)

Faculty of Physics, Astronomy and Applied Computer Science
Jagiellonian University

Abstract

This document describes the data persistence facilities exploited by the offline reconstruction software of the ATLAS detector. The first part describes the infrastructure employed to manage, serialize and deserialize the data objects whereas the second part explains the enhancements and extensions that have been proposed and prototyped within the pursuit of the author's final project. The new functionality that has been introduced is part of two distinct domains: it is a storage scheme for some of the most commonly used data structures as well as an extension supporting the data model schema evolution. The first enhancement makes it possible to efficiently store the STL collections of polymorphic pointers, reducing the amount of metadata and time required to recreate them. The second one proposes a framework for loading the data serialized using the old data model libraries into the most recent in-memory shapes of the data objects even in the situation when the class hierarchies involved changed significantly over time.

To my mother.

Acknowledgements

Over the years it has been my good fortune to encounter many people who have given me their time, companionship, professional and personal help, and above all: patience. Without them this work would have never been possible. I would like to thank prof. Elżbieta Richter-Wąs for her invaluable help and encouragement since the very beginning of my studies; Markus Elsing for providing me with the opportunity to work at CERN on this project; Rene Brun and Philippe Canal for their readiness to share their knowledge, valuable comments, and ideas that have been incorporated into the design proposal of the schema evolution extention and eventually implemented in the prototype; Anna Kaczmarska for supporting my work on Tau EDM project; Daniel Froidevaux for supporting my first attempts to work with the ATLAS data model; Jennifer Godfrey for checking and fixing my lousy English writing; Armin Nairz and Luc Goossens for countless heavy yet enlightening conversations about the meaning of life and for keeping me sane during the pursuit of this project.

Contents

1 Introduction

This document concentrates on the description of the software used to handle the results of the measurements performed by the ATLAS detector. It also reports the enhancements that were designed and prototyped to improve the software.

This chapter begins with the goal statement, followed by some background information about the organizational environment in which this work has been done. It introduces some definitions, acronyms and abbreviations used throughout the document. Finally, it presents the outline of the remaining chapters.

1.1 The goal of the project

The aim of the project described in this document was to improve the abilities of the input-output system used by the offline reconstruction software of the ATLAS experiment. The facilities mentioned provide the means of managing object stores that have abilities to persistify and recreate the state of the objects being owned by them.

The project started with the development and maintenance of the event data model for representing the hadronic decays of tau leptons in the offline software of the ATLAS detector. It has been designed by the author and undergone constant improvement for over two years. This part (described in section 2.3.4) has been released with the ATLAS software framework and still is a part of it. The next part, done to better recognize the data persistence issues, was to implement the persistent classes within the T/P framework (described in section 2.6) for the tau data model and the tracking data model, this code is also a part of the standard ATLAS software releases. The final step was to make the two improvements to the core IO subsystem that are now a part of the ROOT framework.

The improvements that have been proposed influence two distinct areas. The first goal was to improve the storage scheme for the STL collections of polymorphic pointers. The new functionality enables the system to store these in the column-wise mode, thus reducing the amount of metadata that needs to be kept in order to recreate the exact in-memory shape. This part of the work has been described in section 3.5 and the code is in the ROOT release since version 5.20.

The second goal of the project was to propose the design and implement the prototype of a data model schema evolution extension. This subsystem provides the ability to recreate in memory the state of the objects serialized with the old versions of the data model libraries even in the situation when the class hierarchies involved changed significantly over time. The details are described in section 3.6 and the prototype implemented has been incorporated into ROOT development release 5.22.

The work was supported by the *European Organization for Nuclear Research*.

1.2 The organizational environment

1.2.1 The European Organization for Nuclear Research

The *European Organization for Nuclear Research*, often referred to as CERN, is the world's largest particle physics laboratory. It was established by twelve European countries in 1954 with the aim to provide the infrastructure to carry out complex high energy physics experiments. Following its mandate laid out in the CERN Convention, the organization "has no concern with work for military requirements and the results of it's experimental and theoretical work are published or otherwise made generally available" [18].

CERN is located in the northwest suburbs of Geneva and its sites and facilities are scattered on both sides of the Franco-Swiss border. Presently the organization has twenty member states and eight additional international organizations or countries that have so called observer status. It is the workplace of nearly 12000 scientists and engineers [6].

Several important achievements in particle physics have been made during experiments at CERN, two of which were awarded with the Nobel Prize. As of the time of this document, the preparations for a set of the brand new experiments are nearing their completion. Their primary goal is to discover the Higgs boson, a hypothetical particle that would help explain how otherwise massless elementary particles still manage to construct mass in matter.

1.2.2 The accelerator complex

In order to achieve the energies needed for these experiments, the world's largest system of particle accelerators has been built (see Figure 1). These devices use electric fields to accelerate two counter-rotating beams of electrically-charged particles to very high energies and then collide them together in so called interaction points at which detectors are being constructed. The Large Hadron Collider will be used to collide protons at total collision energy of 14 TeV[1] and Pb (lead) nuclei at total collision energy of 1150 TeV.

The LHC reuses the 27 km circumference underground tunnel that was built for the previous accelerator, LEP, which was dismantled in 2000. Within the accelerator, particles circulate in a vacuum tube and are manipulated by various kinds of superconducting magnets. Dipole magnets keep the particles in their nearly circular orbits and quadrupole magnets focus the beam in the interaction points. The magnets use niobium-titanium (NbTi) cables which become superconducting[2] below a temperature of 10 K. The LHC will operate at an even lower temperature, as it is in a 1.9 K (-271.3°C) bath of superfluid helium. To avoid collisions with gas molecules the internal pressure at the LHC will be as low as 10^{-13} atm [7].

Prior to being injected into the main accelerator, the particles are prepared by a series of systems that successively increase their energy. The first system is the linear accelera-

[1]tera electron volts; electron volt - the amount of energy gained by a single unbound electron when it accelerates through an electrostatic potential difference of one volt

[2]superconductivity - a phenomenon occurring in certain materials at very low temperatures, characterized by exactly zero electrical resistance

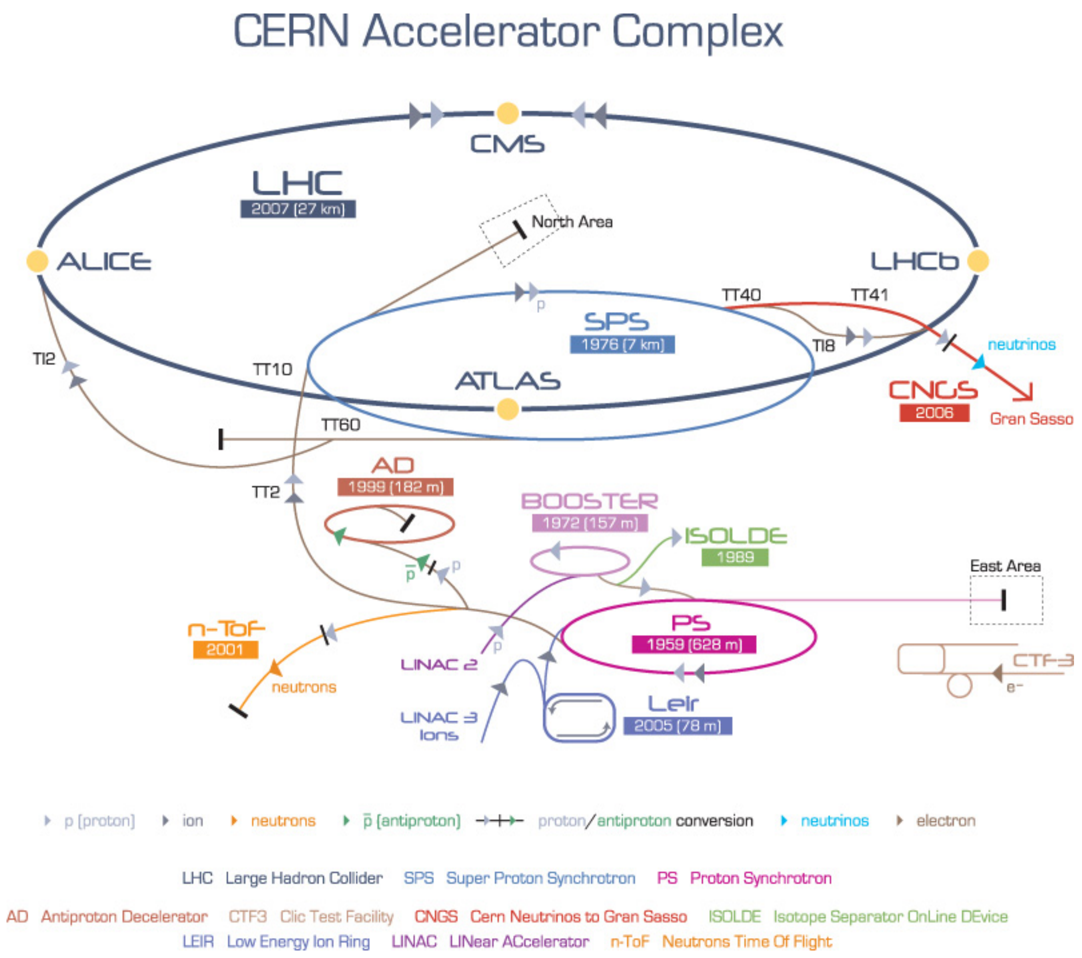

Figure 1: *The CERN accelerator complex*

tor Linac 2, which generates 50 MeV protons and sends them into the Proton Synchrotron Booster (PSB). There the protons are accelerated to 1.4 GeV and injected into the Proton Synchrotron (PS), where they are accelerated to 26 GeV. Finally, the Super Proton Synchrotron (SPS) increases their energy to 450 GeV before they are at last injected into the main ring, where proton bunches are accumulated, accelerated to their peak 7 TeV energy, and rotate for many hours while collisions occur at the four intersection points.

The Pb ions will be first accelerated by the linear accelerator Linac 3 and the Low-Energy Injector Ring (LEIR) will be used as an ion storage and cooler unit. The ions then will be further accelerated by the Proton Synchrotron (PS) and Super Proton Synchrotron (SPS) before being injected into LHC ring, where they will reach an energy of 2.76 TeV per nucleon.

There are six experiments installed at the LHC: A Large Ion Collider Experiment (ALICE), ATLAS, the Compact Muon Solenoid (CMS), the Large Hadron Collider beauty (LHCb) experiment, the Large Hadron Collider forward (LHCf) and TOTEM. ALICE, ATLAS, CMS and LHCb are installed in huge underground caverns built around four collision points of the LHC beams, whereas TOTEM and LHCf share caverns with CMS and ATLAS respectively.

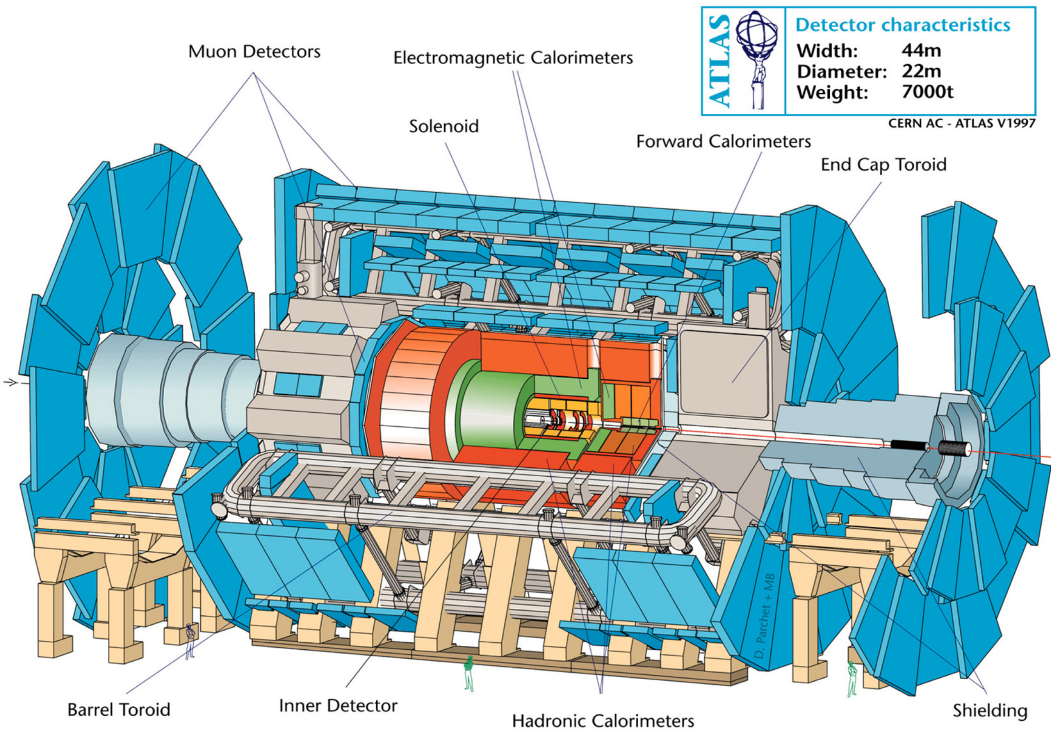

Figure 2: *The ATLAS detector*

1.2.3 A Toroidal LHC Apparatus - ATLAS

ATLAS (see Figure 2) is one of the detectors employed to study the head-on collisions of two beams of protons accelerated by the LHC. Along with CMS, ATLAS is a general purpose detector designed to cover the widest possible range of physics phenomena that can be observed in such circumstances. The detector is 46 meters long and 25 meters high and consists of concentric layers of sensors, each layer providing part of the required information on particle identity, energy and direction.

The innermost part of the detector, the tracker, measures the momentum of each charged particle. It is surrounded by the calorimeters that stop most of the particles, absorbing their energy and producing signals proportional to those energies. High-energy muons are the only observable particles able to traverse the full thickness of the calorimeters without being stopped. The muon spectrometer, surrounding the calorimeters, identifies muons and measures their properties.

Out of nearly 1 billion proton-proton collisions per second, only a few will have the special characteristics that might lead to new discoveries. For example the Higgs particle may be produced in a detectable form in only one collision out of a trillion. To avoid storing immense amounts of uninteresting information, only those few events whose characteristics make them potential candidates for interesting physics are selected[3].

One of the principal challenges for the ATLAS collaboration is to manage a storage infrastructure able to handle the data generated by the experiment. The detector will produce around 25 Megabytes of raw data per event (1.6 MB after zero suppression) and there will be around 40 millions collisions per second occurring within it. This results with the total amount of 1 Petabyte of data per second to be filtered by the trigger systems. After the filtering is done there will be about 200 events (320 MB) per second left to be stored. The raw data will be then processed by the reconstruction software producing Event Summary Data (ESD) (0.5 MB/event) and Analysis Object Data (AOD) (0.1 MB/event) files which are the main subject being described in this document and are discussed in more details later. The core technology exploited to process those files is common to all of the experiments, so all of them are influenced by the enhancements described in Chapter 3 of this document.

1.3 Chapter outlines

Chapter 2 provides a description of a framework within which the ATLAS reconstruction software is being developed. It gives a basic architecture overview and provides a more detailed description of the tools used for data management and serialization. Chapter 3 provides detailed description of the data storage layer that is being used to actually serialize and deserialize the data objects. It discusses in details how the memory chunks are being grabbed and organized in the files and how the data model evolution functionality was implemented.

2 The ATLAS software

This chapter starts with a short introduction to the ATLAS software framework, called Athena by briefly describing basic architecture components and methods used to manage the code, then discusses the object stores and the infrastructure exploited to serialize them and finally concludes with a description of one of the attempts to provide the schema evolution functionality, describing it's strengths and weaknesses.

2.1 Gaudi and Athena

The Athena framework is an enhanced version of the Gaudi framework that was originally developed by the LHCb experiment. Athena and Gaudi are concrete realizations of a component-based architecture (also called Gaudi) which was designed for a wide range of physics data-processing applications. They provide a set of common interface abstractions, dynamic library loading and configuration facilities which allow new components to be plugged in easily during runtime. Also, a clear distinction between the algorithms and data is made. By separating complex algorithmic code that is responsible for creating physics objects based on the raw data from the entities representing the physics objects themselves, the dependencies between producers and consumers of those objects are drastically reduced.

Major framework components are:

- *Application Manager.* The application manager is the overall driving intelligence that manages and coordinates the activity of all other components within the application. There is one instance of the application manager and it is common to all applications.
- *Algorithms.* Algorithms share a common interface and provide the basic per-event processing capability of the framework. Each Algorithm performs a well defined but configurable operation on some input data, in many cases producing some output data.
- *Tools.* A tool is similar to an algorithm in that it operates on input data and can generate output data, but differs in that it can be executed multiple times per event. In contrast to algorithms, tools do not normally share a common interface and are therefore more specialized in their manipulation. Each instance of a tool is owned, either by an algorithm, a service, or by default by the AlgToolSvc.

- *Transient Data Stores.* The data objects accessed by algorithms are organized in various transient data stores depending on their characteristics and lifetimes. The event data itself is managed by one store instance, detector conditions data, such as the geometry and alignment, by another store, etc.
- *Services.* A service provides services needed by the algorithms. In general these are high-level, designed to support the needs of the physicist.
- *Selectors.* These components perform selection. For example, the Event Selector provides functionality to the Application Manager for selecting the input events that the application will process. Other types of selectors permit the selection of objects within the transient data stores.
- *Converters.* These are responsible for converting data from one representation to another. One example is the transformation of an object from its transient form to its persistent form and vice versa.
- *Properties.* All components of the architecture can have adjustable properties that modify the operation of the component.
- *Utilities.* These are C++ classes that provide general support for other components. [4]

The framework also provides a set of basic services offering the minimal functionality needed for constructing applications. The *message service* is used to send and format messages generated in the code, with an associated severity that is used for filtering and dispatching them. The *job options* service allows the configuration of the application by end users assigning values to properties defined within the code. The *random numbers service* makes available several random number distributions via a standard interface, and ensures that applications use a unique random number engine in a reproducible fashion. The *chrono service* offers the functionality for measuring elapsed time and job execution statistics. *Auditors* and *AuditorSvc* provide monitoring of various characteristics of the execution of algorithms. The *incident service* provides a synchronization between objects within the application by using named incidents that are communicated to listener clients. The *tools service*, manages tools [16].

Typically, to run an application, a user creates a python[3] script that is responsible for loading selected components and configuring them according to the user specific needs. This python script is then fed to Athena executable that takes care of running the application according to specified configuration. It is also possible, however, to run the Athena in interactive mode. In that case the python interpreter is invoked and set up allowing the user to control the program flow by typing commands to the terminal.

2.1.1 The "Hello world" example of the algorithm

The following example illustrates a very simple concrete example of an algorithm as being used in Gaudi/Athena environment. The argument does nothing more than notifying the user of initialization and finalization and for every input event prints "Hello World" (or whatever the property "Whom" was set to).

[3]http://python.org/

Listing 1: *The algorithm class declaration*

```
 1 class HelloWorld: public Algorithm
 2 {
 3   public:
 4     HelloWorld( const std::string &name, ISvcLocator *pSvcLocator );
 5     ~HelloWorld();
 6
 7     StatusCode initialize();
 8     StatusCode execute();
 9     StatusCode finalize();
10
11   private:
12     std::string m_whom;
13 };
```

As shown on the Listing 1, the example algorithm class has to inherit from the base class (interface) called *Algorithm* and defined in *Gaudi/Algorithm.h* header. It is obligatory to implement the following virtual methods: *initialize* - called before the event processing commences, typically used for initializing the services exploited during the execution; *execute* - called for every input event, in most cases used to retrieve some objects from the StoreGate (described in more details in section 2.3.1) and modify them or create new ones; *finalize* - called after the event processing is finished to free the resources that were reserved in previous stages. In line 12 a data member that will be used as a property is declared.

Listing 2: *The algorithm class definition*

```
 1 HelloWorld::HelloWorld( const std::string& name,
 2                         ISvcLocator* pSvcLocator )
 3            :Algorithm( name, pSvcLocator )
 4 {
 5   declareProperty( "Whom", m_whom = "World" );
 6 }
 7
 8 StatusCode HelloWorld::initialize()
 9 {
10   MsgStream log( messageService(), name() );
11   log << MSG::WARNING << "Initializing␣the␣algorithm" << endreq;
12   return StatusCode::SUCCESS;
13 }
14
15 StatusCode HelloWorld::finalize()
16 {
17   MsgStream log( messageService(), name() );
18   log << MSG::WARNING << "Finalizing␣the␣algorithm" << endreq;
19   return StatusCode::SUCCESS;
20 }
21
22 StatusCode HelloWorld::execute()
23 {
24   MsgStream  log( messageService(), name() );
25   log << MSG::WARNING << "Executing:␣saying␣hello␣to␣" << m_whom << endreq;
26   return StatusCode::SUCCESS;
27 }
```

Listing 2 shows the implementation. First in the constructor (line 5) the property is declared and will be set to the value specified by the user in the job option file, or to "World" if nothing is specified. At the beginning of each function a message stream object is created to communicate with the outer world. Afterwards, the object can be used as an ordinary C++ ostream, with two minor differences. First the message level is be specified (*DEBUG*, *INFO*, *WARNING*, *ERROR*, and *FATAL*) to signalize the severity of the transmitted message, the message itself and the conclusion *endreq* to signalize the end of the notification.

Listing 3: *The python configuration file*

```
1  include( "AthenaPoolCnvSvc/ReadAthenaPool_jobOptions.py" )
2  include( "AthenaSealSvc/AthenaSealSvc_joboptions.py" )
3  include( "EventInfo/EventInfoDict_joboptions.py" )
4
5  theApp.TopAlg    += [ "HelloWorld" ]
6  hwAlg = Algorithm( "HelloWorld" )
7  hwAlg.Whom = "Lukasz"
8
9  from AthenaCommon.AppMgr import ServiceMgr
10 import AthenaPoolCnvSvc.ReadAthenaPool
11 ServiceMgr.EventSelector.InputCollections = [ "ESD.pool.root" ]
12
13 ServiceMgr.MessageSvc.OutputLevel = 3
14 ServiceMgr.MessageSvc.defaultLimit = 9999999
```

To be able to run the algorithm defined above the user must provide a job option file that defines the environment in which the run should take place (Listing 3). If the algorithm is supposed to process file, then the data persistence services must be configured (lines 1-3) and the event selector must be informed which file to use (line 11). The actual algorithm configuration is performed in lines 5 to 7.

2.2 The code management

The code of the ATLAS software is divided among several projects grouping the components of similar functionality. For instance:

- *AtlasCore* groups all the basic functionality required by the rest of the software (data persistence services, basic run-time libraries, python kernel and so on)
- *AtlasEvent* contains all the components describing the event data model (tracks, calorimeter cells, electrons, photons and so on)
- *AtlasReconstruction* consists of the algorithms responsible for constructing the "high-level" physics objects such as electrons, photons, taus or muons from the data obtained from the detector
- *AtlasAnalysis* provides functionality enabling users to analyze those objects

2.2.1 The code repository

To manage the code a revision control system is being used. The current implementation is based on the Concurrent Version System[4] however a substantial effort is being made to migrate to something less antique with Subversion [5] being the most probable replacement.

[4] http://www.nongnu.org/cvs/
[5] http://subversion.tigris.org/

CVS allows for decomposition of the codebase into packages usually mapping to concrete Athena components. Each package is accessible to all the developers for reading and requires permissions to check-in the changes.

2.2.2 The package manager

Build and run environment management for the projects is being done by the CMT[6] tool. The tool provides an environment for Linux systems capable of configuring requested packages, dealing with their dependencies and carrying out the (re)compilation process. In order to be handled by CMT, packages must be organized in a specific way and must provide a *requirements* file in which the required external components and the compilation and code generation rules are declared.

Each package is contained in a directory named with the package name and is divided into the following subdirectories:

- *cmt* - contains all of the administrative files used to configure and define the package; typically contains only the *requirements* file (Listing 4) and is a directory where the makefiles and other temporary files are created
- *src* - contains all implementation files and private headers
- *share* - usually contains python or other configuration files
- <*package name*> - contains all the header files that are to be exported (visible for other packages)

Listing 4: *An example requirements file*

```
package HelloWorld
author Lukasz Janyst <ljanyst@cern.ch>

use AtlasPolicy            AtlasPolicy-*
use GaudiInterface         GaudiInterface-*        External
use StoreGate              StoreGate-*             Control
use AthenaPoolCnvSvc       AthenaPoolCnvSvc-*      Database/AthenaPOOL
use AthenaPoolUtilities    AthenaPoolUtilities-*   Database/AthenaPOOL
use AthenaPoolServices     AthenaPoolServices-*    Database/AthenaPOOL

library HelloWorld *.cxx components/*.cxx
apply_pattern component_library
```

The listing 4 presents the *requirements* file used for the package containing the Hello World example algorithm described in section 2.1.1. At the beginning the package name and author are declared, then the packages that are required to build and run the HelloWorld package are listed and finally the component library is declared to be created from all the files having *cxx* suffix. The source files are compiled and linked into a form of dynamic library which can then be scheduled to be loaded by the python configuration file and provide access to all the components implemented by it.

[6] http://www.cmtsite.org/

2.3 The event data model

The purpose of the event data model classes is to publish results computed by arbitrarily complex algorithms. The results are then registered with an in-memory database (a blackboard, more in Section 2.3.1) and made available to other algorithms or persistified. Decoupling data objects from the algorithms producing them has proven to be very scalable since the algorithmic code is usually volatile and the data object provide a stable interface for the clients. The ATLAS persistence model does not require any arbitrary interface to be implemented by the data objects in order to be managed by the blackboard system, generally everything that fulfils *STL Assignable* concept can be correctly handled. In most cases however the objects are grouped into STL-like collections, the *DataVectors*, before being published. This section starts with a description of some basic concepts that affect the design and implementation of the data model and then provides a description of a typical data model class hierarchy, using tau leptons as an example, as well as the most complicated one responsible for describing the tracks that the particles followed in the detector.

2.3.1 The data object blackboard

As mentioned before, the ATLAS reconstruction software uses a kind of blackboard to pass the data objects between the algorithms and, eventually, persistify some of them if instructed to do so. Essentially, the blackboard holds key-object pairs. To retrieve a stored object, the user has to know a key and a type of the desired object, symlinking and retrieval using a base class is also supported. This functionality is provided by an Athena service called *StoreGateSvc*, which exposes the object store functionality (described in more details in section 2.4) to algorithm developers.

The only requirement for a class to be registered with the *StoreGate*, besides fulfilling of an *STL Assignable* concept, is to provide a specialization of the *ClassID_traits* template done with the *CLASS_DEF* macro.

Listing 5: *A basic usage of StoreGate*

```
1 StoreGateSvc *storeGate;
2 StatusCode sc = service( "StoreGateSvc", storeGate );
3
4 sc = storeGate->record( container, "ContainerName" );
5 sc = storeGate->setConst( container );
6
7 const Rec::TrackParticleContainer *trackContainer;
8 sc = storeGate->retrieve( trackContainer, trackContainerName );
```

Listing 5 shows the basic usage of the ATLAS data object blackboard. Lines 1 and 2 show how the service handle can be retrieved. This is usually done at the algorithm initialization time. Lines 4 and 5 show how a container can be registered with a given string key, lines 7 and 9 show how a container can be retrieved provided that the user knows the associated key.

2.3.2 DataVectors

Typically all of the collections are either instances of the *DataVector* class templated for the contained elements or classes deriving from *DataVector* with some added functionality.

A *DataVector<T>* acts like a *std::vector<T*>* with few exceptions. The most important are:

- Optionally, depending on how it is constructed, it can manage the memory of the objects it contains. The ownership policy is set on construction time by passing the *SG::VIEW_ELEMENTS* or *SG::OWN_ELEMENTS* parameter to the constuctor. By default, *DataVector* owns its elements so that the elements are deleted either when any method removes them from the container or when the container itself is deleted.
- Methods that return a reference in *std::vector* return *ElementProxy* objects instead. This is done to ensure proper handling of the elements being owned by the container. That is, when an element is assigned to a new object, the old element is deleted and the container takes the ownership over the new one.
- Due to ownership issues, standard algorithms that alter the range that they operate on (such as data*sort* or *remove_if* do not work properly on *DataVector*s. Hence, *DataVector* specific specializations are provided.
- *DataVector*s may inherit from one another. If a contained class *A* inherits from class *B* then *DataVector<A>* may also derive from *DataVector<B>*, as set by the user. This is done with *DATAVECTOR_BASE* macro which provides a template specialization for the *DataVectorBase* template used in the *DataVector* class definition (see Listings 6 and 7).

Listing 6: *The* DataVectorBase *template*

```
template <class T>
struct DataVectorBase
{
  typedef DataVector_detail::NoBase Base;
};
```

Listing 7: *The* DataVector *class definition*

```
template <class T, class BASE = typename DataVectorBase<T>::Base>
class DataVector : public BASE
```

2.3.3 Persistent object pointers

It is often required to link together two objects registered with *StoreGate*. Ffurthermore in most cases it is necessary to provide a link to particular element of the registered collection. To fulfill these requirements the concept of persistent pointers was introduced and the functionality is provided by two classes. An object of the *DataLink* class provides a connection to a "top level" object registered with *StoreGate*. An *ElementLink* is a means of pointing to an element of a "top level" collection.

Both classes are based on policies [2] (Listings 8 and 9). *DataLink* and *ElementLink* both use *StoragePolicy* set to *DataProxyStorage<STORABLE>* by default which knows how to retrieve the requested object from the database. The *ElementLink* not only needs to retrieve a "top level" object but also must return a pointer to one of its constituents. The *IndexingPolicy* provides the means to decompose the storable object and pick the right element. In this case the *GenerateIndexingPolicy* template checks if the container fulfils the

STL concept of *Sequence*, ie. has a *ForwardIterator*, if so the *ForwardIndexingPolicy* is used, otherwise the user must provide his own specialization of *DefaultIndexingPolicy* which can cope with his custom container.

Listing 8: *The* DataLink *class definition*

```
template <typename STORABLE,
  class StoragePolicy=DataProxyStorage<STORABLE> >
class DataLink:
  public StoragePolicy
```

Listing 9: *The* ElementLink *class definition*

```
template <typename STORABLE,
  class StoragePolicy=DataProxyStorage<STORABLE>,
  class IndexingPolicy=typename SG::GenerateIndexingPolicy<STORABLE>::type >
class ElementLink:
  public StoragePolicy,
  public IndexingPolicy
```

2.3.4 The Tau Event Data Model

The Tau Event Data Model (EDM) is a moderately complicated case among the data model class hierarchies used in ATLAS software, still it is affected by most of the IO perfomrance and data model evolution issues. It has been designed and maintain for over two years by the author of this document and more detailed description of the contents and the evolution over time of the class hierarchies can be found in the following documents: [14] [19] [10] [11] [15]. The EDM was designed to meet the needs of two reconstruction algorithms, a simulation algorithm (ATLFAST), the High Level Trigger system and the data analysis framework. It consists of the main class, called *TauJet* and various detail classes, some of which are stored in files depending on the level of details required for given file type [12].

The *TauJet* class (Figure 3) is the trunk of the data model. It implements the *INavigable4Momentum* interface providing generic access to its constituents and holds the basic information about the reconstructed particle and links to objects holding more detailed information:

- name of the algorithm which reconstructed the object
- discriminant scores the object got from cuts, neural networks, likelihood algorithms, boosted decision trees and so on
- *ElemenLinks* to objects that were used to produce the TauJet (Tracks, CaloClusters, Jets)
- *ElementLinks* to *TauDetails* objects providing more detailed, algorithm specific information

The *ParticleBase* base class holds the information about the charge of the particle and the vertex of its origin while *P4EEtaPhiM* manages the 4-Momentum information.

The *TauDetails* classes hold the algorithm specific information. Each reconstruction algorithm stores the results of its computations in either **Details* or **ExtraDetails* class depending on the importance of the specific information for the user performing the analysis. All of the details classes are stored in the *ESD* (Event Summary Data) files while the *AOD* (Analysis Object Data) files store only **Details* classes (**ExtraDetails* are omitted).

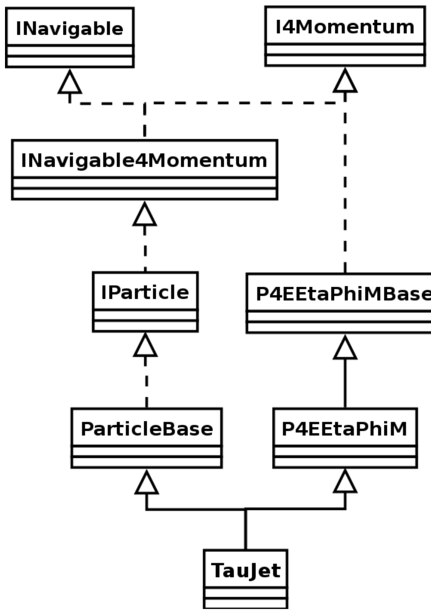

Figure 3: *The inheritance hierarchy of TauJet*

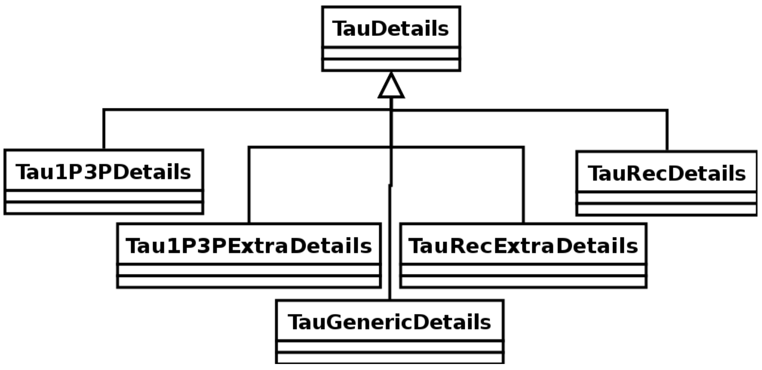

Figure 4: *The inheritance hierarchy of TauDetails*

2.3.5 The Tracking Event Data Model

The Tracking Event Data Model describes the tracks followed by the particles passing through the detector. It is generic enough to handle data produced both by the Inner Detector and the Muon Spectrometer. Consisting of around a hundred classes it is by far the most complicated case within the ATLAS software. A detailed description can be found in [1]. The description provided by this section is very brief and concentrates on the class hierarchy as only this information is relevant to the discussion of the IO performance and the data model evolution issues.

The code is scattered between many packages in the *Tracking*, *InnerDetector* and *MuonSpectrometer* projects. The main class is called *Trk::Track*, it contains a *Trk::TrackSummary*, *Trk::FitQuality* objects, a vector of *Trk::TrackStateOnSurface*s, as well as some vectors of cached objects which are marked as “transient” and are irrelevant from the stand point of the serialization system. The *Trk::TrackStateOnSurface* class consists of pointers to objects deriving from *Trk::ParametersT<Charged>*, *Trk::MaterialEffectsBase* and

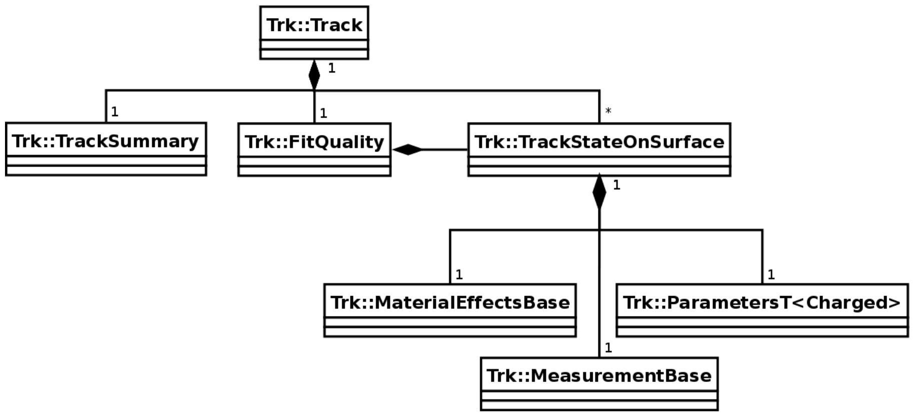

Figure 5: *The contents of Trk::Track*

Trk::MeasurementBase (see Figure 5). The inheritance hierarchies of these classes are rather complex and often involve multiple inheritance, the classes on all levels of the hierarchy contain other complex objects. As an example, Figure 6 presents the inheritance hierarchy for the *Trk::TrackParameters* class.

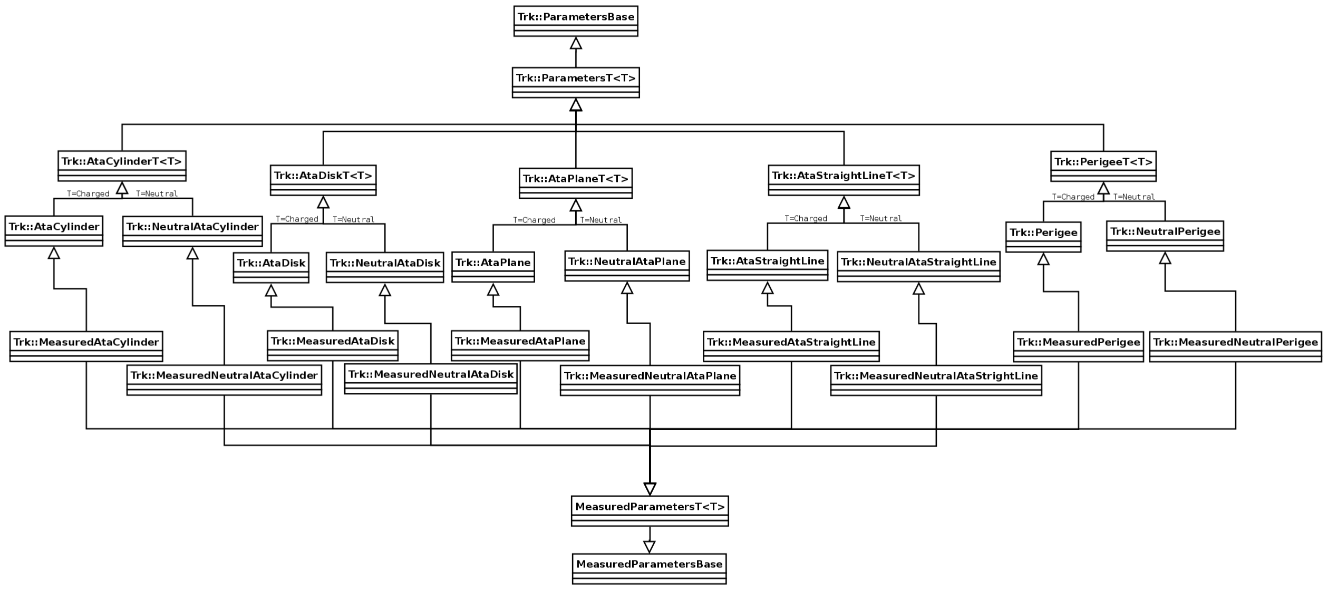

Figure 6: *The inheritance hierarchy of TrackParameters*

In total, the EDM contains around a 80 classes (around 55 in *Tracking*, 15 in *InnerDetector* and 10 in *MuonSpectrometer*) [13].

2.4 The object store

Before the user object can be handled by an object store it has to be wrapped into a *DataBucket* object. The *DataBucket* objects provide additional information about their contents such as their inheritace hierarchy and equip the system with ability to cast the object being held to one of its base types by knowing the CLID or typeinfo of the target type. The object has to be wrapped again, this time in the *DataProxy* object that holds the information necessary to store and retrieve the data from POOL (see Section 2.7). The data store itself is implemented as a set of maps associating names and CLIDs with the actual data proxy objects.

2.5 The streaming algorithms and event selectors

The writing process is performed by an ordinary algorithm which is run every event after all other algorithms are completed. It takes all the objects from the object store and calls the appropriate POOL converter. Additional information that cannot be stored with the object data is put into a structure called *DataHeader* that is also stored in the file.

The reading is initiated by an event selector that reads the data header for particular event and decides whether the event can be processed. If it can, the event selector processes the contents of the data header and recreates the *DataProxy* objects from the information stored in it and puts these objects in the object store. Later, when a user algorithm requests an object (via *StoreGateSvc::retrieve*) then the right proxy object is found and the POOL converter is called to recreate the object's in-memory state. The recreated object is then returned to the user.

2.6 The transient-persistent separation

The converters are used to modify the objects before they are passed to POOL prior to writing and after they are retrieved from POOL durring the reading process. By default the converters are generated by a CMT macro and do not modify the objects that are being passed through them. This ability to change the objects has been used to implement the ATLAS system handling the schema evolution and modifying the objects so that they may easily be handled by ROOT IO.

The main idea is to have two different class hierarchies holding the same information: the transient one that is being used by the algorithms and the persistent one that is being passed to the ROOT IO system via POOL (see Section 2.7). At any given moment in time there may be many persistent representations that were used to represent the changing transient data model and that were serialized to files at different times. The persistent class has the same name as the transient one with *_px* suffix, where x is the current persistent version number. There can be only one transient representation: the current one. There is also a set of converters that are capable of translating any persistent representation to the current transient one and vice versa.

Listing 10 shows an example POOL converter for a *TauJetContainer* object holding a set of *TauJet* objects described in section 2.3.4. The conversion to a persistent object (as performed prior ro writing) picks the most recent T/P converter and converts the current transient data model to the most recent persistent data model, in this case *Analysis::TauJetContainer* to *TauJetContainer_PERS* which is a typedef to *TauJetContainer_p3*. The persistent to transient conversion, performed after reading, involves checking which persistent version had been written to a file by comparing the POOL identifiers and picking the right converter for the job.

Listing 10: *A POOL Transient-Persistent converter for TauJetContainer*

```
TauJetContainer_PERS*
TauJetContainerCnv::createPersistent(Analysis::TauJetContainer *transCont)
```

```
{
   MsgStream msg( msgSvc(), "TauJetContainerCnv" );
   TauJetContainerCnv_p3 cnv;
   TauJetContainer_PERS *persObj = cnv.createPersistent( transCont, msg );
   return persObj;
}

Analysis::TauJetContainer *TauJetContainerCnv::createTransient()
{
   MsgStream msg( msgSvc(), "TauJetContainerCnv" );
   Analysis :: TauJetContainer *transObj = 0;

   static pool::Guid p1_guid( "AD52E539-5A69-493A-B33C-7BE558348EBA" );
   static pool::Guid p2_guid( "3F9C4AF7-1B48-4DBC-BA24-F7CF658E7820" );
   static pool::Guid p3_guid( "3B6CC0D5-D033-45A6-9440-0276EE55B4C5" );

   if( compareClassGuid( p3_guid ) ){
      std :: auto_ptr<TauJetContainer_p3>
            persObj( poolReadObject<TauJetContainer_p3>() );
      TauJetContainerCnv_p3 cnv;
      transObj = cnv.createTransient( persObj.get(), msg );
   }
   else if( compareClassGuid( p2_guid ) ){
      std :: auto_ptr<TauJetContainer_p2>
           persObj( poolReadObject<TauJetContainer_p2>() );
      TauJetContainerCnv_p2 cnv;
      transObj = cnv.createTransient( persObj.get(), msg );
   }
   else if( compareClassGuid( p1_guid ) ){
      std :: auto_ptr<TauJetContainer_p1>
            persObj( poolReadObject<TauJetContainer_p1>() );
      TauJetContainerCnv_p1 cnv;
      transObj = cnv.createTransient( persObj.get(), msg );
   }
   else
      throw
      std::runtime_error("Unsupported␣persistent␣version␣of␣TauJetContainer");

   return transObj;
}
```

The persistent object is a mirror of the transient one. It usually contains the same members and the base classes are contained as data members, not inherited. The converter usually does an assignment for the data members being basic types or calls appropriate T/P converters of the complex types (see Listings 11 and 12).

Listing 11: *One of the persisten object representing transient* TauJet *class*

```
class TauJet_p3 {
   friend class TauJetCnv_p3;
   public:
      TauJet_p3(): m_flags( 0 ), m_vetoFlags( 0 ),
                   m_isTauFlags( 0 ), m_roiWord(0) {};
      ~TauJet_p3() {};

   private:
      P4EEtaPhiMFloat_p2       m_momentum;
```

```
    ParticleBase_p1          m_particleBase;
    ElementLinkInt_p1        m_cluster;
    ElementLinkInt_p1        m_cellCluster;
    ElementLinkInt_p1        m_jet;
    ElementLinkIntVector_p1  m_tracks;
    ElementLinkIntVector_p1  m_tauDetails;
    unsigned char            m_flags; // 0 bit - has TauPID object
                                      // 1 bit - is Tau // obsolete
                                      // 2 bit - author TauRec
                                      // 3 bit - author Tau1P3P
    unsigned long            m_vetoFlags;
    unsigned long            m_isTauFlags;
    unsigned long            m_numberOfTracks;
    unsigned int             m_roiWord;  // requested by trigger
    std::vector<std::pair<int, float> > m_params;
};
```

Listing 12: *Parts of a transient-persistent converter*

```
static P4EEtaPhiMCnv_p2                                    momCnv;
static ParticleBaseCnv_p1                                  partBaseCnv;
static ElementLinkCnv_p1<ElementLink<CaloClusterContainer> > clusterCnv;
static ElementLinkCnv_p1<ElementLink<JetCollection> >      jetCnv;

void TauJetCnv_p3::persToTrans( const TauJet_p3     *pers,
                                Analysis::TauJet *trans,
                                MsgStream           &msg ) {

   momCnv.persToTrans( &pers->m_momentum, trans, msg );
   partBaseCnv.persToTrans( &pers->m_particleBase, trans, msg );
   clusterCnv.persToTrans( &pers->m_cluster, &trans->m_cluster, msg );
   clusterCnv.persToTrans( &pers->m_cellCluster, &trans->m_cellCluster, msg );
   jetCnv.persToTrans( &pers->m_jet, &trans->m_jet, msg );

   trans->m_numberOfTracks = pers->m_numberOfTracks;
   trans->m_roiWord = pers->m_roiWord;

   if( getBit( pers->m_flags, 2 ) )
      trans->setAuthor( TauJetParameters::tauRec );
   if( getBit( pers->m_flags, 3 ) )
      trans->setAuthor( TauJetParameters::tau1P3P );
}
```

This kind of approach ensures that a wide range of possible schema evolution scenarios can be handled. It also enables users to make some modifications to the data structures that can make them easier to handle for the underlying ROOT IO technology. Usually the pointers are preplaced with the actual objects and in some cases the data structures are flattened to ensure that the ROOT IO will be able to store the data model entirely in a split mode and to save the space and time spent in the IO operations. The flattening is done by having a persistent "top-level" object holding *std::vectors* containing all the possible objects present in the class hierarchy and the information on how to recreate the transient structure. Also, some compression is done by packing together the floating-point data members that do not need to be stored with full precission offered by the *double* type.

There is no way to generate the persistent versions automatically so the developer must do a lot of dull typing to actually implement the persistent classes and the converters. Since the persistent classes and the transient persistent converters contain explicit references to other persistent types, one can observe so called "chain effect" which occurs when one of the classes placed at the bottom of the hierarchy changes. In such a case, all other classes and converters referring to that class must also be updated. Given the fact that the ATLAS data model consists of hundreds of classes it, is a huge maintenance problem. Also the flattening and the floating-point numbers packing make the converters more sophisticated, and hence more difficult to maintain.

2.7 POOL - the LCG persistence framework

The aim of POOL is to follow a technology neutral approach to the storage systems. It provides a set of service APIs that isolate experiment framework user code from the details of a particular implementation technology. Even though POOL implements object streaming via ROOT-I/O and uses MySQL as an implementation for relational database services, there is no link time dependency on the ROOT or MySQL libraries. Back end component implementations are instead loaded at runtime.

The POOL system is based on a hybrid technology approach. It combines two main technologies with quite different features into a single consistent API and storage system. The first technology includes so-called object streaming packages (e.g. ROOT I/O described in Chapter 3) that deal with persistence for complex C++ objects, such as event data components. The second technology class provides Relational Database (RDBMS) services, such as distributed, transaction consistent, concurrent access to data that still can be updated.

POOL implements a distributed store with full support for navigation between individual data objects. References between objects are transparently resolved meaning that referred-to objects are brought into the application memory automatically by POOL as required by the application. References may connect objects in either the same file or spanning file and even technology boundaries. Physical details such as file names, host names, and the technology that holds a particular object are not exposed to reading user code. These parameters can therefore easily be changed, which allows for optimizing the computing fabric with minimal impact on existing applications[9].

3 The ROOT Framework

ROOT is an object-oriented framework aimed at solving the data analysis challenges of high-energy physics. It provides, among other tools, a C++ class introspection functionality and data serialization framework that are widely used in the software of the LHC based experiments. Virtually all of the reconstructed data will be stored and processed using them.

To give the reader a broader perspective on how these systems were implemented this chapter starts with a brief discussion of the memory layout of C++ classes as generated by the GCC. Next, it presents a sketch of the compiler independent introspection systems based on dictionaries. Thirdly the description of the file format and the serialization framework itself is presented. Finally, this chapter discusses the schema evolution facilities of which the foundations were layed out by the author of this document.

3.1 The memory layout of C++ objects

In order to serialize or read back the objects the system must be able to access the memory occupied by them in a generic way. That is it has to be able to list the data members and access or assign their values. The task is quite difficult to do in a portable way as the C++ standards do not enforce a common ABI [7] so that the compilers are free to organize the layout of the objects in a way that is optimal for a given system. This section shows an example of how the object memory is organized by the gcc compiler and can be accessed by the user. More detailed information on the C++ object model and memory access can be found in [17] and [5].

Listing 13: *A simple class hierarchy*

```
class Left {
  public:
    int m_a;
    double m_b;
};

class Right {
  public:
```

[7]Application Binary Interface - http://en.wikipedia.org/wiki/Application_binary_interface

```
    double m_c;
    double m_d;
};

class Bottom: public Left, public Right {
  public:
    int m_e;
};
```

Listing 13 defines a very simple class hierarchy where one class derives from two others without any additional complications in the form of virtual methods or virtual inheritance. In memory an object of class *Bottom* takes the form presented in the Figure 7. It occupies a consecutive chunk of memory with all of the data members packed into it. The contents of the object can be then accessed simply by knowing where the object starts and what is the offset of the required data member. As an example the value of the data member *Right::m_d* can be accessed or assigned by dereferencing the *m_d* pointer (Listing 14).

Data member name	**Offset**
Left::m_a	0
Left::m_b	4
Right::m_c	12
Right::m_d	20
Bottom::m_e	28

Figure 7: *The memory layout of the class defined in Listing 13*

Listing 14: *Data access example*

```
Bottom* b   = new Bottom();
double* m_d = (double*)(((char*)b) + 20);
```

The class hierarchy described above is rather simple and it is probable that all the compilers will represent it in the same way. Things get more complicated though, using polymorphism or virtual multiple inheritance. The compiler uses the virtual tables to figure out the offset of some of the data members and the pointers to the right virtual functions [8]. In general, however, the object always occupies a consecutive chunk of memory but the offsets to the same data member may differ from platform to platform.

3.2 The introspection systems

The introspection or reflexion systems are facilities by which a computer program can observe and modify its own structure and behavior. They are usually present in a high-level languages and provide on runtime a set of features such as an ability to discover and modify the code constructs, convert a string matching a function to actual function call, create and access an object of a class knowing just a string name of that class and so on. The C++ language does not provide any of these features but they can be emulated to some extent by a system of dictionaries. The ROOT framework currently supports two systems of this kind: it's native *CINT* as well as *Reflex* that has been originally developed by the LCG project[8] and can be interfaced to ROOT via subsystem called *CINTEX*. The next two

[8]http://lcg.web.cern.ch/lcg/

sections take a closer look at both of the systems concentrating on the areas relevant for the object persistence, ie. runtime discovery of the object shapes.

3.2.1 Reflex

Reflex is a system that is used by the ATLAS experiment to provide the introspection capabilities for *ATHENA* and the class shape information for the data persistence system. It comes with a well defined C++ interface and a generator that analyzes the source files to create the dictionary code. The dictionary generation process is based on a *GCCXML* application that uses GCC to parse a C++ program and generate its XML representation from GCC's internal data structures. The result is then picked up by a python program called *genreflex* and translated again to a C++ dictionary code. *Genreflex* can optionally take a selection.xml file that defines which parts of the input program should be present in the resulting dictionary. The dictionary code can be then compiled and either dynamically or statically linked to the application using the introspection system (see Figure 8).

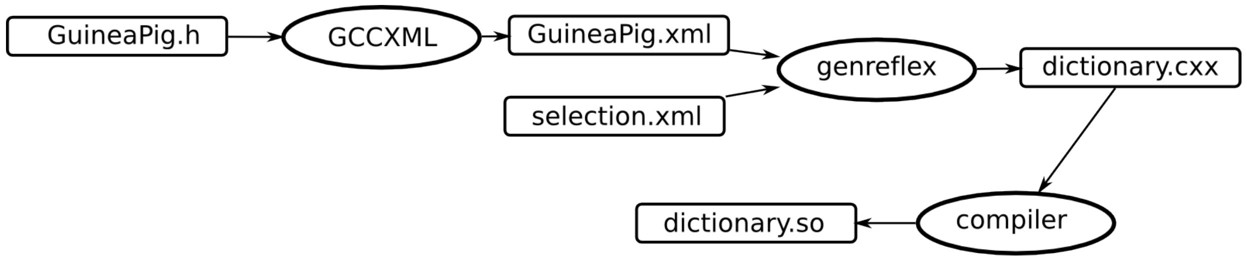

Figure 8: *Reflex dictionary generation process*

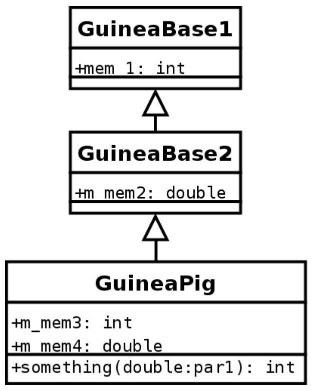

Figure 9: *GuineaPig class hierarchy used to test the introspection systems*

Listing 15: *Reflex overview*

```
Type    t = Type::ByName( "GuineaPig" );
Object  o = t.Construct();

for( size_t  i = 0; i < t1.FunctionMemberSize(); ++i ) {
  std::cout << "Method:␣" << t1.FunctionMemberAt( i ).Name() << std::endl;
}
```

```
7
8  for( size_t i = 0; i < t.DataMemberSize(); ++i ) {
9    std::cout << "Data␣Member:␣" << t.DataMemberAt( i ).TypeOf().Name();
10   std::cout << ",␣offset:␣" << t.DataMemberAt( i ).offset() << std::endl;
11 }
12
13 for( size_t i = 0; i < t.BaseSize(); ++i ) {
14   std::cout << "Base:" << t.BaseAt( i ).Name() << std::endl;
15 }
16
17 Type tbase1 = Type::ByName( "GuineaBase1" );
18 Type tbase2 = Type::ByName( "GuineaBase2" );
19 Object obase1 = o.CastObject( tbase1 );
20 Object obase2 = o.CastObject( tbase2 );
21
22 o.Set( "m_mem3", 43 );
23 o.Set( "m_mem4", 45.35 );
24
25 std::cout << "m_mem3␣=␣" << Object_Cast<int>( o.Get("m_mem3") ) << std::endl;
26
27 std::vector<void *> parVect;
28 double par1 = 12.32;
29 parVect.push_back( (void *)&par1 );
30 int ret1 = Object_Cast<int>( o.Invoke( "something", parVect ) );
31 std::cout << "Returned␣value:␣" << ret1 << std::endl;
32
33 o.Destruct();
```

Listing 15 briefly presents the most important aspects of the *Reflex* C++ API. The code deals with the class hierarchy shown on Figure 9. The first two lines deal with the type finding and the object construction; lines from 4 to 15 show how various pieces of information about the class hierarchy are retrieved; lines from 17 to 20 demonstrate casting to base classes; in lines 22-25 the data members are being accessed for reading and writing; lines from 27 to 31 demonstrate a method call and finally in line 33 the object is destructed.

Listing 16: *Reflex dictionary*

```
namespace __shadow__ {
#ifdef __GuineaPigSubSub1
#undef __GuineaPigSubSub1
#endif
class __GuineaPig : public ::GuineaBase2 {
  public:
  __GuineaPig();
  virtual ~__GuineaPig() throw();
  virtual int something(double) throw();
  int m_mem3;
  double m_mem4;
};
}

namespace {
static void* method_1491( void* o, const std::vector<void*>& arg, void*)
{
  static int ret;
  ret = (((::GuineaPig*)o)->something)(*(double*)arg[0]);
  return &ret;
21 }
22
23 void __GuineaPig_dict() {
24   ClassBuilder("GuineaPig", typeid(::GuineaPig), sizeof(::GuineaPig),
25                            PUBLIC | ARTIFICIAL | VIRTUAL, CLASS)
26   .AddBase(type_593, BaseOffset< ::GuineaBase1, ::GuineaPig >::Get(),
27                            PUBLIC)
28   .AddDataMember(type_28, "m_mem3", OffsetOf(__shadow__::__GuineaPig, m_mem3),
29                            PRIVATE)
30   .AddDataMember(type_1367, "m_mem4",
31                            OffsetOf(__shadow__::__GuineaPig, m_mem4), PRIVATE)
32   .AddFunctionMember(FunctionTypeBuilder(type_void), "~GuineaPig",
33                            destructor_1490, 0, 0, PUBLIC | VIRTUAL | DESTRUCTOR )
34   .AddFunctionMember(FunctionTypeBuilder(type_28, type_1367), "something",
35                            method_1491, 0, "arg1", PUBLIC )
36   .AddFunctionMember<void*(void)>("__getBasesTable", method_x0, 0, 0,
37                            PUBLIC | ARTIFICIAL);
38 }
39 }
40 namespace {
41   struct Dictionaries {
42     Dictionaries() {
43       __GuineaPig_dict();
44     }
45     ~Dictionaries() {
46       type_196.Unload(); // class GuineaPig
47     }
48   };
49   static Dictionaries instance;
50 }
```

In most cases class data members are declared private, but in order to dump the state of objects the system still needs to be able to access them. To cope with the visibility issues *REFLEX* creates "shadow" classes with exactly the same memory shape as the ones defined by users (Listing 16, lines 2-12), so that the right offsets can be obtained without causing compiler errors and without having to use preprocessor tricks (such as defining private to be public). Methods are wrapped into functions accepting object pointers and vectors of parameters so that they can always be called in the same way (lines 16-21). To build the full type information *Reflex* creates dictionary functions putting all the information together (lines 23 to 38) and finally a static data structure is declared to invoke the dictionary functions when the dictionary library is loaded dynamically and unload them when it is removed from the memory (lines 40 to 50).

3.2.2 ROOT CINT

The *CINT* dictionaries are natively supported by ROOT and extensively used throughout the framework. The data persistence system uses the information provided by them to dump and recreate the state of the in-memory objects. To generate the dictionaries a built-in C++ interpreter is used and an additional "LinkDef" file can be accepted to list the entities for which the dictionary should be generated and pass some additional parameters (Figure 10).

The API for manipulating object through the *CINT* dictionary system is somehow more raw the one provided by *Reflex*. It exposes the memory directly to the user via void pointers. *TClass* object is used to get the information about the shape of the class hierarchies. It knows

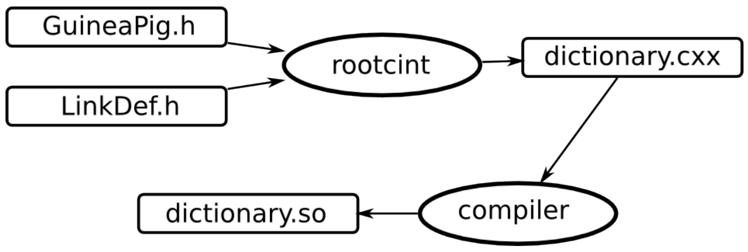

Figure 10: *ROOT CINT dictionary generation process*

how to construct and destruct the objects and holds all the information about the types and offsets of the base classes and data members associated with it.

Listing 17: *CINT Overview*

```
TClass*   cl  = TClass::GetClass( "GuineaPig" );
void*     obj = cl->New();

TIter next( cl->GetListOfBases() );
TBaseClass* base;
while( base = (TBaseClass*)next() ) {
  TClass* baseCl  = base->GetClassPointer();
  void*   basePtr = ((char*)obj) + cl->GetBaseClassOffset( baseCl );
}

TIter next( cl->GetListOfDataMembers() );
TDataMember* member;

while( member = (TDataMember*)next() ) {
  std::cout << member->GetTrueTypeName() << " " << member->GetName();

  TClass *mcl = TClass :: GetClass( member->GetTrueTypeName() );

  if( member->IsBasic() && !strcmp( member->GetTrueTypeName(), "int" ) ) {
    int* mem = (int *)(((char *)obj) + member->GetOffset());
  }
  else if( mcl ) {
    char *ptr = ((char *)obj) + member->GetOffset();
  }
}

cl->Destructor( obj );
```

The dictionary code is very similar to the one presented in Listing 16. Two major difference are that the dictionary code creates the lightweight *TGenericClassInfo* objects for each type (these objects are then converted to *TClass* objects on demand). The second difference is that sometimes there is no need to create the shadow classes, since the classes deriving from TObject can be instrumented with a *Streamer* function providing access to the class private content.

3.3 CINTEX

CINTEX is a subsystem capable of converting types and functions known to *Reflex* to their *CINT* counterparts. When enabled, it converts all the known *Type* objects to instances of the *TGenericClassInfo* class. It also registers a type listener with the *Reflex* API so that it is notified when new dictionary information is loaded and the conversion to *CINT* structures can be performed.

3.4 The ROOT files

3.4.1 The file format

A ROOT file consists of one ”file header”, one or more “data records,” and zero or more “free segments”. The file header is of fixed length and is always located at the beginning of the file, while the data records and free segments may in principle appear in any order. The file header contains some basic information about the file, including a “magic file identifier,” file version number, pointers to the beginning of the list of data records and so on.

A free segment is of variable length and it is a set of contiguous bytes that are unused and available to use for new or resized data records. The first four bytes of a free segment contain the negative of the number of bytes in the segment while the contents of the remaining part are irrelevant.

A data record represents either user data or data used internally by ROOT. All data records have two portions, a “key” portion and a “data” portion. The key portion precedes the data portion. The format of the key portion is the same for all data and corresponds to an object of the *TKey* class. The object name and they key cycle are together sufficient to uniquely determine the record within the file. All the data in a ROOT file is stored in machine independent format (ASCII, IEEE floating point, Big Endian byte ordering).

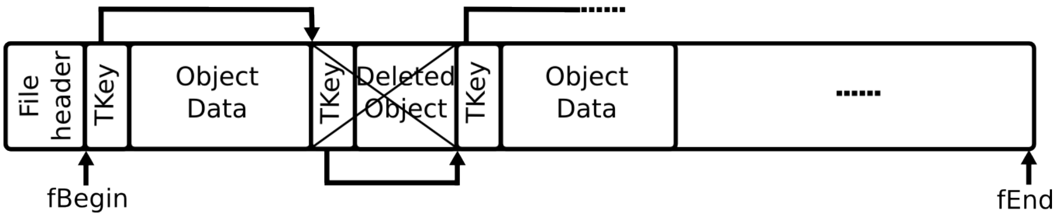

Figure 11: *ROOT File Structure*

There are several types of data records used internally by the IO system to support the storage of byte sequences. These record types are “TFile” for storing data relevant to the file as a whole, “TDirectory” for managing logical contents of the file, “KeysList” containing a list of the logical objects, and “FreeSegments” sotring pointers to the unused chunks of the file.

3.4.2 Logical structure of the ROOT file

Logically a ROOT file is like a UNIX directory, it can contain objects or other directories. The *TFile* class used for representing files provides ways of browsing and altering this directory structure as well as facilities for storing of retrieving objects.

3.4.3 Trees

The ROOT trees, represented by the *TTree* class, provide a sophisticated IO functionality based on top of the simple physical and logical structure described in sections 3.4.1 and 3.4.2. It is especially well suited for storing a large number of the same-class objects which is usually the case for the *LHC* experiments. The *TTree* class is optimized to reduce disk space and enhance the access speed and is usually stored as an object in a directory. The major difference between storing objects in trees instead of using directories directly is that in trees the objects are not written individually but rather collected and written bunch at a time.

A tree consists of objects called branches, which are objects of classes deriving from the *TBranch* class. The branches allow users to decompose the objects into portions that are stored in separate buffers. This kind of approach reduces the amount of storage space since the branch and its sub-branches always hold the objects of the same type needed to store the metadata which is required to restore the in-memory shape just once. Moreover, the portions of data that end up in the same sub-branch are usually very similar, since they refer to the same data member, so they can be better handled by the deflation algorithm used to compress the buffers.

Listing 18: *Example of Tree creation*

```
 1 const char* dictname = "./libGuineaPig_dictCINT.so";
 2 if( argc == 2 && !strcmp( argv[1], "reflex" ) ) {
 3    dictname = "./libGuineaPig_dictREFLEX.so";
 4    ROOT::Cintex::Cintex::Enable();
 5 }
 6
 7 gSystem->Load( dictname );   // or a coresponding dlopen call on posix systems
 8                              // for that matter
 9
10 GuineaPig*                oGP     = new GuineaPig();
11 std::vector<GuineaPig>*   vGP     = new std::vector<GuineaPig>();
12 std::vector<BuineaPig*>*  vGPStar = new std::vector<GuineaPig*>();
13
14 TFile* file = new TFile( "ARootFile.root", "RECREATE" );
15 TTree* tree = new TTree( "TestTree", "A␣ROOT␣Tree" );
16
17 tree->Branch( "GuineaPig", &oGP, 32000, 99 );
18 tree->Branch( "GuineaPigVector", &vGP, 32000, 99 );
19 tree->Branch( "GuineaPigVectorStar", &vGPStar, 32000, 99 );
20
21 tree->Fill();
22 file->Write();
23 file->Close();
```

Listing 18 shows how a tree can be created for a class hierarchy shown on Figure 13. It presents how to store a single object of this class, a standard vector of objects of this class and a standard vector of pointers, possibly polymorphic, to objects of this class. Before dealing with trees, the shape of the class hierarchy needs to be made known to the system. To do that either a *REFLEX* or *CINT* dictionary must be loaded (lines 1 to 8) and we also need to have some data objects to be stored (lines 10 to 13). The next step is to create a file and a tree objects (lines 14 and 15) and then create branches holding the objects (lines 17 to 19). The first argument to the procedure creating a branch is the name the new branch should be identified by. The second one is a pointer to the data object that should be stored in the branch, the third one is a size of a single buffer and a fourth one is a split level. The buffer size determines how many objects can be grouped together before the buffer is written down and a new one is created. The split level tells the system how long the objects should be split if it is possible. That is how many levels of sub-branches representing the data members can be created. In this case it is set to 99 so that there is one branch for each data member. The *TTree::Fill* procedure fills the appropriate branch buffers with the data held by the objects and creates a new *entry*. The state of the objects may change later and be dumped again with a call to *Fill*, thus creating another entry. At the end the file needs to be closed.

The object splitting is possible as long as the system does not encounter pointers as the structure of the tree is static and the contents of the memory pointed to by the pointers can only be determined on run-time (actually this was true for older versions of ROOT but the system was updated to adapt the structure of the trees in some cases due to the improvements done by this project).

Listing 19: *Example of reading back the objects*

```
1  TFile* file = new TFile( "ARootFile.root", "READ" );
2
3  TTree* tree = (TTree*)file->Get( "TestTree" );
4  tree->SetBranchAddress( "GuineaPig", &oGP );
5  tree->SetBranchAddress( "GuineaPigVector", &vGP );
6  tree->SetBranchAddress( "GuineaPigVectorStar", &vGPStar );
7
8  tree->GetEntry(0);
9
10 file->Close();
```

The read-back procedure is presented in Listing 19. Obviously the dictionary information must also be loaded before the objects are loaded as was shown on Listing 18. To read the objects an input file needs to be opened for reading (line 1) and the tree object needs to be retrieved (line 3). The next step is to tell the branch where in memory the data should be recreated (lines 4 to 6). The *TTree::GetEntry* methods loads the requested objects to the memory. After all the reading is done the file has to be closed.

3.4.4 Dealing with buffers

When the branch buffers are filled with data they are written to a file as ordinary records, as described in section 3.4.1. The tree object together with it's branches is handled the same way. The branches have all the information required to locate on file the right buffer records for the requested entry so that no linear search is required and the retrieval process is very efficient. Furthermore, if the contents of a buffer vary in size then the buffer itself contains a lookup table pointing to the offset of the right entry.

3.4.5 Ways of describing persistified data

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. As an example, the streamer info structure for the class presented in Figure 9 is shown on Figure 12. *TStreamerInfo* object representing the *GuineaPig* class holds three *TStreamerElement* objects representing the entities the class is consisted of. There is one concrete element of class *TStreamerBase* representing the base class and two elements of class *TStreamerBasicType* representing the data members. Before the read process starts this information is matched against the information provided by the dictionaries to handle simple changes in the in-memory class shape.

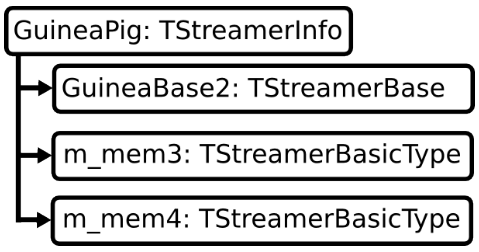
GuineaPig: TStreamerInfo
GuineaBase2: TStreamerBase
m_mem3: TStreamerBasicType
m_mem4: TStreamerBasicType

Figure 12: *Example TStreamerInfo structure*

3.4.6 Serialization and deserialization process

The actual procedures of serialization and deserialization are performed recursively by methods of the *TStreamerInfo* class desciribeing the input object. The serialization procedure dumps relevant part of the memory to an object of the *TBuffer* class whereas the deserialization procedure performs essentially the same operation in the reverse direction and taking into account the type change if one occurred and can be handled. The scope of the serialization procedure, that is whether the entire input object should be handled or just some parts of it, depends on the split level.

The STL collections are handled in special way, because while processing collections the system is not really concerned with the memory shape of the object representing the collection, but with the actual elements. Hence, the *TCollectionProxy* class was introduced to wrap the collection objects and provide a common interface to handle all of them. The collection proxies provide the means to determine how many elements are contained by the collections and define a *operator[](int)* method which returns the pointer to the indexed element.

If the current object is not supposed to be split, the serialization (and deserialization) is done by looping over the streamer elements and dumping the chunks of the object memory that the current streamer element describes to a buffer. The collections of objects (not pointers to objects) can be handled in two ways: they can be streamed either object-wise or member-wise. The object-wise streaming simply dumps the elements to the buffer one-by-one whereas the member-wise streaming processes the first data member of all elements in the collection, then the second one and so on.

If the object is split between many branches then the serialization is done by looping over the branches responsible for handling the parts of the input object. The addres of the relevant part is passed to the *TStreamerInfo* object together with the index of the streamer element that should be processed. The remaining part of the serialization process is essentially the same.

If the object being processed is not split then with every complex object in the hierarchy (a base class or a data member of non-basic type) a 32-bit bytecount is stored pointing to the position where the object ends. This enables the system to continue reading even if something goes wrong with the current object (a dictionary or a streamer info is missing). Furthermore, with every complex object a 16-bit class version is written or if such information is unavailable the systems assume that it is equal to zero and writes a cyclic redundancy checksum computed from the dictionary information about the data members. Behavior like that enables ROOT to recognize the mismatch between the in-memory shape of the class and the contents of the buffer and adopt to such situation.

This kind of behavior provides a flexible way to handle errors as well as means of recognizing differences between in-memory shape and the data that has been stored on disk. In particular if the objects involved are small the relative overhead is very significant.

In the split-mode the error recovery and versioning information is stored only once, with the appropriate branch, not with every object. This allows for huge space savings while dealing with big number of relatively small objects.

3.4.7 Serializing and deserializing pointers

Cases involving pointers must be handled in a special way as cycles are possible and one physical object can be pointed to by many pointers. Also, the pointer can point to any object deriving from the one that was declared or be a zero pointer.

To cope with this situation, every buffer contains an object map associating 32-bit tags with the addresses of the objects. The writing is performed as follows:

1. If the input pointer is a zero-pointer then *kNullTag* is written to the buffer and the writing algorithm ends

2. If the input pointer points to an address that was handled already (so that it has an entry in the object map) then the tag of the object is being stored and the algorithm ends

3. If the object was not stored before then:

    3.1 A space for 32-bit bytecount is reserved

    3.2 The actual class name is written in the following way:

        3.2.1 If the *TClass* object corresponding to the class has an entry in the object map (an object of this class was written in this buffer already) then the object tag (ored with *kClassMask* is stored

        3.2.2 If such class was not stored before then the *kNewClassTag* is stored and the string with the name of the class follows

    3.3 The actual object is streamed

    3.4 The bytecount is written in the slot reserved before

This kind of algorithm is able to handle every possible case of data structure containing pointers, including graphs with cycles. The flexibility is quite costly though as additional space is used for the tags and additional computation time is spent in map searches. If the class hierarchy being stored involves many small objects, it is heavily dependent on pointers. If one object is not pointed to more than once and there are no cycles (as in the case of ATLAS), then this kind of algorithm provides little gain for a high price.

3.5 The split storage mode for collections of pointers to polymorphic objects

During the pursuit of this project, a set of improvements were made to allow the tree structures to handle the STL collections of polymorphic pointers. The structure dynamically upgrades its shape creating new sub-branches when new objects are encountered.

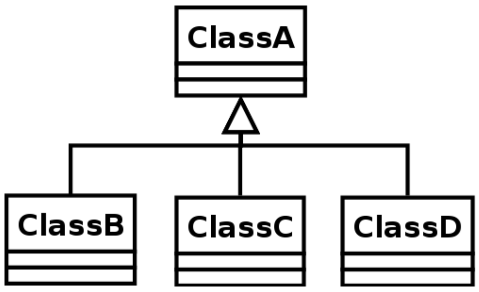

Figure 13: *Class hierarchy used for split collection example*

Normally the entire tree structure is created before any data is stored in the tree. An input object is split and branches are created for its base classes, data members and their data members as long as splitting is no longer possible either due to encountering a nested collection or pointers in any form. The splitting is performed during the *TTree::Branch* call (Listing 18) and done this way does not work for pointers since a pointer can point to any object deriving from the declared class or be a zero pointer. Therefore, the tree structure cannot be determined in advance. The branch creation algorithm was upgraded to create an object of *TBranchSTL* class when it encounters an STL collection of pointers and delegate the further splitting to its *Fill* procedure when the actual data is being processed.

The *TBranch::Fill* algorithm works in the following way:

1. A table for 8-bit indices capable of containing as many elements as the input collection is created

2. For every element in the input collection the following is performed:

   2.1 If the element being processed is a zero pointer, then zero is inserted in the position of the index table corresponding to the current position in the input collection and the loop proceeds to the next element.

   2.2 If the element is not a zero pointer and its concrete type hasn't been encountered yet then a new *TBranchElement* object is created (together with appropriate sub-branches) to represent an *std::vector* of pointers to this object. In this case splitting is possible even for pointers since we can safely assume that the collection will contain objects of the same concrete type and no zero pointers. The existing structures were updated to handle such a case as a collection of actual objects.

   2.3 If the concrete type was encountered before or operation 2.2 was performed, then it means that we have a sub-branch and a collection dedicated for this type elements. The element is inserted into this collection and the ID of the sub-branch is inserted in the position of the index table corresponding to the current position in the input collection.

3. The index table is stored in the buffers of the *TBranchSTL* object.

4. The *TBranchElement::Fill* method is called for all direct sub-branches if the *TBranchSTL* object.

Figure 14 shows the example *TBranchSTL* object with the immediate sub-branches created while storing an *std::vector<ClassA*>* filled with the objects presented on Figure 13.

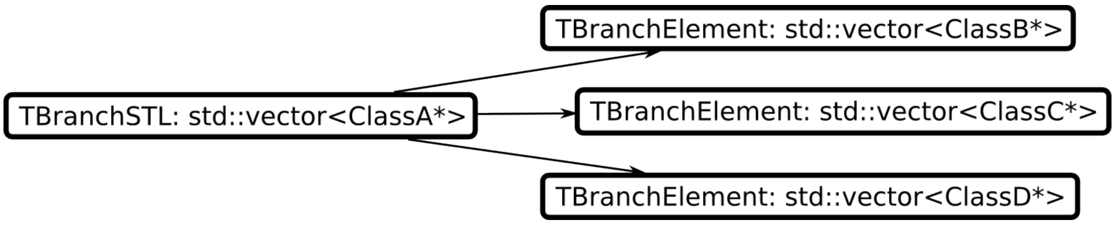
TBranchSTL: std::vector<ClassA*>
TBranchElement: std::vector<ClassB*>
TBranchElement: std::vector<ClassC*>
TBranchElement: std::vector<ClassD*>

Figure 14: *Branch hierarchy resulting from splitting of the class hierarchy presented on 13*

The deserialization algorithm reads the index table and loops over the indices. If a zero is encountered then a null pointer is inserted to the output collection. Otherwise, the appropriate branch data is read and the last unprocessed element in the vectored stored in it is inserted to the output collection.

The actual buffering is done by the same code that is invoked for a collection of objects. The collection of pointers that is being processed is passed to it by an adaptor class. Finally, many other minor improvements to other entities responsible for processing the trees were made.

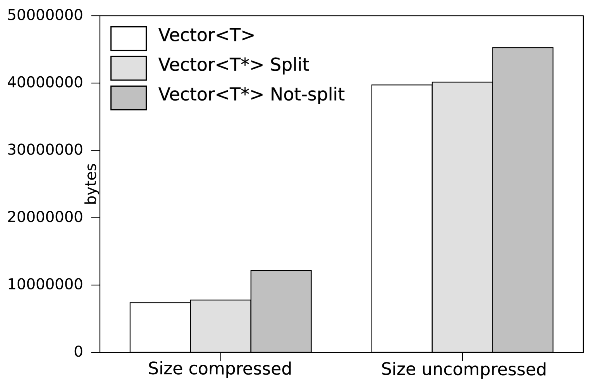

Figure 15: *Differences in size of the same collection stored in differend modes*

To check the behavior of the algorithm the ROOT IO test storing 400 entries with collections of 1000 hit objects in each (a hit object contain 3 floats and up to 31 ints) has been updated to also test a split store for the case of a collection of pointers. Figure 15 shows the differences in the amount of storage space occupied in the case of storing the same objects as a vector of objects, a vector of pointers to objects in a split mode and a vector of pointers in a non-split mode. The plots show that the storage space taken by a split vector of pointers is slightly larger than in the case of a vector of objects. The overhead is caused by the necessity of having an index table to recreate the collection in memory.

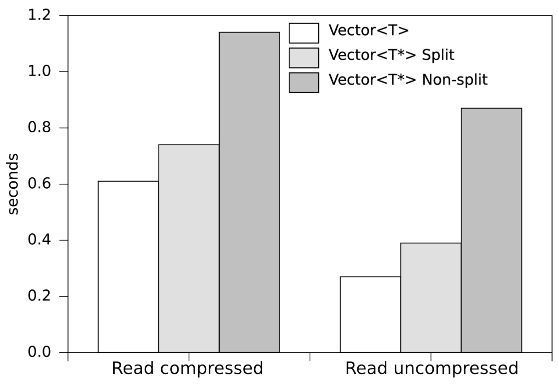

Figure 16: *Differences in reading time of the same collection stored in different modes*

Figure 16 shows the reading time for the three cases mentioned in the previous paragraph. A big difference between a vector of objects and a split vector of pointers is caused by a necessity to process the index table and to allocate the memory in less efficient way. In the former case the memory already allocated can be reused whereas in the later case only the memory for holding pointers is pre-allocated and the memory for the actual objects has to be allocated during the reading process. The amount of time used to read a non-split collection of pointers is even greater due to the necessity to inspect the object map to recreate the pointer structure (as described in 3.4.7) also the amount of the storage space ocupied in this case is larger so more time may be spent in the operating system IO calls and the decompression algorithm.

3.6 The data model evolution

The reconstruction and data analysis algorithms used by the experiments are constantly improved. From time to time there is a need to change the data model to better support the extended algorithms and the new features offered by them. Sometimes the changes are trivial, such as adding a new data member to handle a new property of the object computed by algorithm and sometimes the class hierarchy needs to be changed significantly. As mentioned before the experiments produce huge amounts of data and the updated algorithm must be able to process the data files written down using old class hierarchies. Hence the persistence systems have to provide facilities for loading back the old data structures to new in-memory representations.

3.6.1 Automatic conversion capabilities

In the previous versions of the system the ability to load older versions of class shapes into memory worked 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 were 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 with a uniform interface. These facilities are helpful but insufficient to meet all the needs of the experiments. An effort has been made to extend the system to handle these needs more, and this section describes the proposal of the extensions and presents the implemented prototype.

3.6.2 How the automatic conversion is done

As it was mentioned in Section 3.4.6 the actual reading of an object is done by the *TStreamerInfo* structure associated with the class of the object. If the difference between in-memory class and on-disk class is detected (the checksums or the version numbers differ) then the *TStreamerInfo::BuildOld* method tries to adjust the offsets to the new values or disables the streamer elements if the data members associated with them are not present in the current in-memory object.

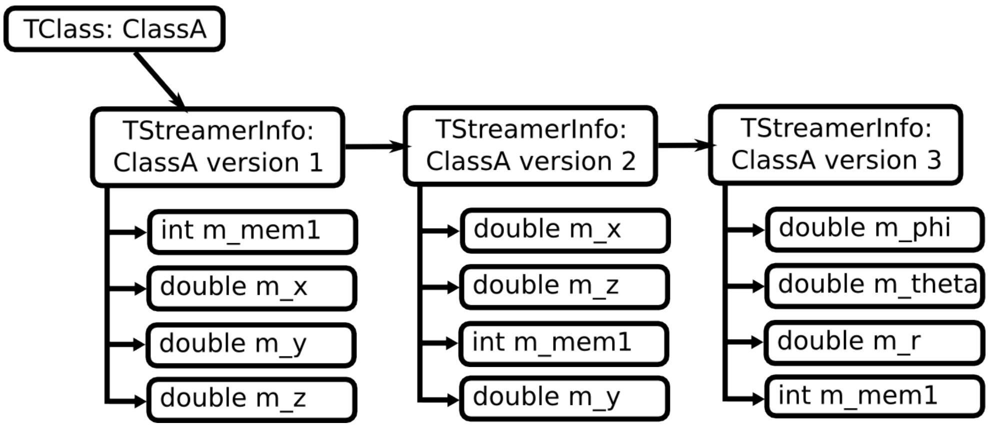

Figure 17: *The* TStreamernfo *objects associated with* TClass *for a sample hierarchy*

Figure 17 shows an example *TClass* object holding a list of streamer info structures corresponding to different versions of a class. If the objects in memory are the object of class *ClassA* in version 2 and the read of the version 1 is attempted, then the system could easily adjust the streamer info for version 1 to read data to the new shape. If, however, the in-memory shape was that of version 3 and the reading of version 1 or 2 was attempted, then the adjustments could not be made because there is no sufficient information to perform the appropriate conversions. The extension that has been proposed attempts to cope with that.

3.6.3 The requirements for the extension

This section lists and discusses the requirements that the system has to meet to be usable by the experiments and acceptable for other users not concerned by the schema evolution issues.

Backward and forward compatibility The newly implemented functionality cannot cause any backward compatibility problems. The system must be able to process the files written with the old versions of the software. Also the forward compatibility has to be preserved in the sense that the files written with the new system have to be readable with the old versions of ROOT.

No performance penalty for the users that do not wish to use the new facilities The newly implemented functionality cannot impose any significant costs on the users who do not wish to use it.

Transparency The conversion hints must always be declared in the same way regardless the storage mode used to handle the objects.

Minimization of the chain effects The chain effects described in the sections dedicated to the T/P separation (2.6) have to be avoided as much as it is possible.

Possibility of storing the conversion rules in the files The system has to provide the possibility of serializing the conversion rules so that they could be used in a bare ROOT mode when the dictionary information for the serialized objects is not available and the system tries to emulate the in-memory shape.

Ability to set the values of the transient data members The system has to provide the functionality allowing users to set the values of the transient data members.

Handling the data member and class name changes The system must provide the functionality to easily cope with the situation in which the names of the data members or the classes involved in the structure change but there is no other modification.

Ability to cache the on-disk data and convert it to the new shape The system must be able to load the data stored on disk to the memory cache and provide access to it in an efficient way for a user-specified conversion function. The system also has to provide the metadata describing the cached data including the information about the types of objects and their version numbers.

Ability to directly access the buffer In some rare occasions (ie. when some internal ROOT writing algorithms change) there is a need to have a direct access to a buffer instead of the cached data. The system has to provide facilities for handling situations like that.

3.6.4 The general idea for the extension

Simple data model changes can easily be handled by the system by matching the names of the streamer elements against the dictionary information or disabling the streamer elements that do not correspond to any data members in the in-memory class. This approach could be easily extended to data member renaming by providing the system with the information that on-disk data member *m_mem1* is an in-memory data member *m_mem2* and the conversion can be done either automatically or using other rules specified by the user. The same situation appears when we need to rename a class. If the system encounters on-disk *ClassA* and the user requested to load it to in-memory *ClassB* the system could be instructed to match the streamer info for *ClassA* against the dictionary information of *ClassB*. The situation is more complicated when some of the data members cannot be matched, such as when an on-disc class stores some information about coordinates using cartesian system and the in-memory class requests to have the same information in spherical system, other data members match just fine. In such a situation a conversion function needs to be provided, in this case a conversion function which takes cached information encoded in cartesian system and converts it to a spherical system. A conversion function take as input an arbitrary sub-set of the data members of the object on file and sets an arbitrary sub-set of the data members of the in-memory object.

The general idea for the schema evolution extension is to upgrade the *BuildOld* procedure to cope with more sophisticated changes by inserting "artificial" streamer elements into the streamer info structure that are handled in a special way by the reading routine. The artificial streamer elements do not represent any data member neither in on-disk class nor in in-memory class. When encountered by the reading procedure they instruct it to perform operations other that putting the currently processed chunk of data from the buffer to a memory location that corresponds to the data member represented by the element.

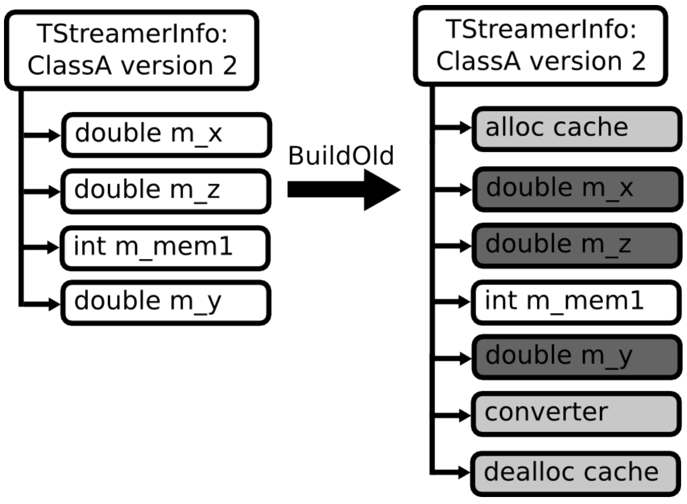

Figure 18: *The upgrades to the* BuildOld *method*

Figure 18 is an example how the streamer info structure for class *ClassA* in version 2 could be upgraded to read this kind of objects to the in-memory version 3. First an artificial element instructing the read procedure to allocate the cache is inserted, then the members that are required as input to one of the rules are marked as needing to be stored in the cache. After these data members are put into the cache the conversion can be performed and a streamer info holding the information about the conversion function to be called is inserted. The final step is to clean up the cache. Note that there is no need to do anything with data member "m_mem1" since it is present both on-disk and in memory so it is processed the same way it was done before.

3.6.5 The syntax of the rules defined in the dictionaries

The system provides two ways of defining the conversion rules. One way is to specify the rule in the dictionary helper file (LinkDef.h in *CINT* or selection.xml in *REFLEX*). The rule is processed by the dictionary generator and converted to C++ code in the dictionary file. The information is then passed to the introspection system and can be modified or extended from the C++ code by the API described in section 3.6.7. It is more convenient than defining the rule directly in the C++ code because the dictionary generator can do some preprocessing of the code snippet provided by the user defining some helper variables which makes the code that user has to write a lot simpler.

Listing 20: *Normal schema evolution rule defined in CINT LinkDef file*

```
#pragma read                                              \
    sourceClass="ClassA"                                  \
    source="double m_a; double m_b; double m_c"           \
    version="[4-5,7,9,12-]"                               \
    checksum="[12345,123456]"                             \
    targetClass="ClassB"                                  \
    target="m_x"                                          \
    targetType="int"                                      \
    embed="true"                                          \
    include="iostream; cstdlib"                           \
    code="{m_x = onfile.m_a * onfile.m_b * onfile.m_c;}"  \
```

Listing 21: *Normal schema evolution rule defined in REFLEX selection XML*

```
<ioread sourceClass="ClassA"
        source="double m_a; double m_b; double m_c"
        version="[4-5,7,9,12-]"
        checksum="[12345,123456]"
        targetClass="ClassB"
        target="m_x"
        targetType="int"
        embed="true"
        include="iostream; cstdlib">
<![CDATA[
  m_x = onfile.m_a * onfile.m_b * onfile.m_c;
]]>
</ioread>
```

Listings 20 and 20 present the syntax for defining the rules that depend on the buffered code. They consist of a set properties and a code snippet.

- **sourceClass** - The field defines the on-disk class that is the input for the rule.
- **source** - A semicolon-separated list of values defining the source class data members that need to be cached and accessible via object proxy when the rule is executed. The values are either the names of the data members or the type-name pairs (separated by a space). If a type is specified then the *ondisk* structure can be generated and used in the code snippet defined by the user.
- **version** - A list of versions of the source class that can be an input for this rule. The list has to be enclosed in a square bracket and be a comma-separated list of versions or version ranges. The version is an integer number, whereas the version range is one of the following:
  - "a-b" - a and b are integers and the expression means all the numbers between and including a and b
  - "-a" - a is an integer and the expression means all the version numbers smaller than or equal to a

  - "a-" - a is an integer and the expression means all the version numbers greater than or equal to a

- **checksum** - A list of checksums of the source class that can be an input for this rule. The list has to be enclosed in a square brackets and is a comma-separated list of integers.
- **targetClass** - The field is obligatory and defines the name of the in-memory class that this rule can be applied to.
- **target** - A semicolon-separated list of target class data member names that this rule is capable of calculating.
- **targetType** - This property is meant to define the target type if used without dictionaries (when the rule was serialized to a file), it is ignored otherwise.
- **embed** - This property tells the system if the rule should be written in the output file is some objects of this class are serialized.
- **include** - A list of header files that should be included in order to provide the functionality used in the code snippet.
- **code** - An user specified code snippet, see 3.6.6.

Listing 22: *Raw schema evolution rule defined in CINT LinkDef file*

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

Listing 23: *Raw schema evolution rule defined in REFLEX selection XML*

```
<ioreadraw  sourceClass="TAxis"
            source="fXbins"
            targetClass="TAxis"
            target="fXbins"
            version="[-5]"
            include="TAxis.h">
<![CDATA[
   Float_t *xbins = 0;
   Int_t n = buffer.ReadArray( xbins );
   fXbins.Set( xbins );
]]>
</ioreadraw>
```

Listings 22 and 23 present the syntax for defining a raw conversion rule. A raw conversion rule differs from a normal conversion rule by the type of the input data. A normal conversion rule accepts as its input a cached data that can be accessed via *TVirtualObject* whereas a raw rule accepts an object of *TBuffer* class that contains the data member that was declared in a source property of the rule. Since ROOT supports many storage modes and data members of the same object may end up in different buffers the raw rule can only declare one data member as it's input.

3.6.6 The code snippets in the rules declared in dictionaries

The user provided code snippets have to consist of valid C++ code. The system can do some preprocessing before wrapping the code into function calls and declare some variables to facilitate the rule definitions. The user can expect the following variables being predeclared:

- **newObj** - variable representing the target in-memory object, it's type is that of the target object
- **oldObj** - in normal conversion rules, an object of *TVirtualObject* class (details are presented in 3.6.8) representing the input data, guaranteed to hold the data members declared in the source property of the rule
- **buffer** - in raw conversion rules, an object of *TBuffer* class holding the data member declared in source property of the rule
- names of the data members of the target object declared in the target property of the rule declared to be the appropriate type
- onfile.xxx - in normal conversion rules, names of the variables of basic types declared in the source property of the rule

3.6.7 The C++ API concerning handling the rules

The C++ API, as presented on Figure 19, is a key part of the new system. It allows users to define the rules that have to be applied to convert the old on-disk data to the new versions of the data model classes from the C++ code. It also allows for the modifications or removal of the rules that were declared in the dictionaries. All conversion rules are associated with the target in-memory class and are being held by the *TClass* object corresponding to this class. Each *TClass* object can have exactly one *TSchemaRuleSet* object associated with it. The rule set object holds and facilitates manipulations and querying for the rules and can also be serialized to persistify the schema evolution information in the new files that are produced. Objects of the *TSchemaRule* class are being owned by the *TSchemaRuleSet*s and they cannot be deleted if they are returned outside in the *TObjArray*s or *TSchemaMatch* objects. The rule sets also perform consistency checks of the rules contained in them. The user cannot, for example, define two rules for the same input class version producing the same output data member.

The *TSchemaRule* object corresponds to a schema rule declared in a dictionary. Most of the setter methods accept strings in exactly the same format as the ones described in section 3.6.5 and they provide convenient ways of getting the values they hold. that is the target data members are returned as a linked lists of string names. The class also provides methods for testing if the rule represented by it should be applied to a class of given version or checksum. There are also two types of “special” rules. First of all, an alias rule (a rule having only a *targetClass* and a *sourceClass* defined) tells the system that it is possible to do a conversion between the classes mentioned. Secondly a renaming rule with *sourceClass*, *targetClass*, *source* and *target* properties defined tells the system that the data members described by the rule have the same meaning (either only the name changed or the conversion can be performed using other rules). In the C++ code it is possible to pass the code as a snippet but the user can also provide a function pointer doing the conversion. A normal conversion function accepts a void pointer that is set to a target object being reconstructed and a *TVirtaulObject* pointer pointing to an object containing cached information. A raw conversion function also accepts a void pointer to a target object but the second parameters is a pointer to *TBuffer* object.

TStreamerInfo::BuildOld method has been updated to take the conversion rules into account and to insert the artificial streamer elements that are then processed by the reading procedure. The streamer infos that were updated to result with current in-memory shape for old on-disk input data are cached because the adjustment procedure takes much time and would have to be repeated very frequently.

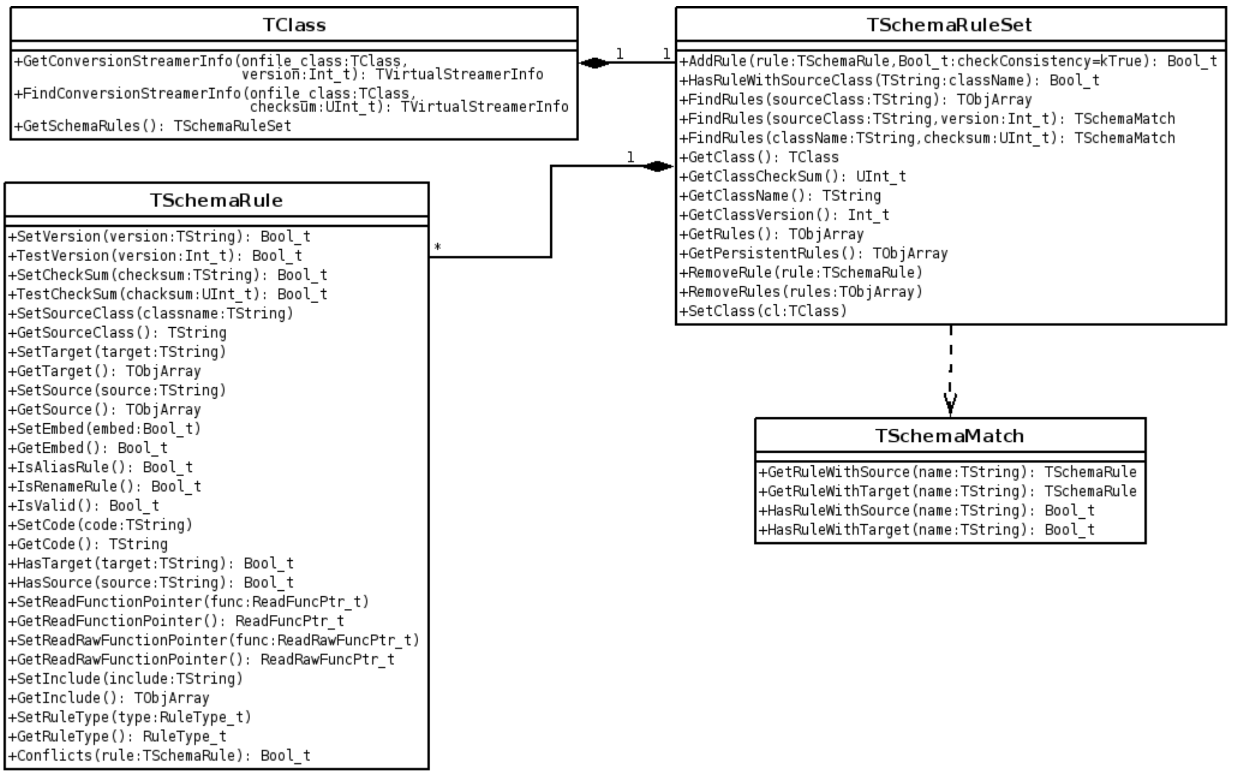
TClass
+GetConversionStreamerInfo(onfile_class:TClass, version:Int_t): TVirtualStreamerInfo
+FindConversionStreamerInfo(onfile_class:TClass, checksum:UInt_t): TVirtualStreamerInfo
+GetSchemaRules(): TSchemaRuleSet
1
1
TSchemaRuleSet
+AddRule(rule:TSchemaRule,Bool_t:checkConsistency=kTrue): Bool_t
+HasRuleWithSourceClass(TString:className): Bool_t
+FindRules(sourceClass:TString): TObjArray
+FindRules(sourceClass:TString,version:Int_t): TSchemaMatch
+FindRules(className:TString,checksum:UInt_t): TSchemaMatch
+GetClass(): TClass
+GetClassCheckSum(): UInt_t
+GetClassName(): TString
+GetClassVersion(): Int_t
+GetRules(): TObjArray
+GetPersistentRules(): TObjArray
+RemoveRule(rule:TSchemaRule)
+RemoveRules(rules:TObjArray)
+SetClass(cl:TClass)
1
*
TSchemaRule
+SetVersion(version:TString): Bool_t
+TestVersion(version:Int_t): Bool_t
+SetCheckSum(checksum:TString): Bool_t
+TestCheckSum(chacksum:UInt_t): Bool_t
+SetSourceClass(classname:TString)
+GetSourceClass(): TString
+SetTarget(target:TString)
+GetTarget(): TObjArray
+SetSource(source:TString)
+GetSource(): TObjArray
+SetEmbed(embed:Bool_t)
+GetEmbed(): Bool_t
+IsAliasRule(): Bool_t
+IsRenameRule(): Bool_t
+IsValid(): Bool_t
+SetCode(code:TString)
+GetCode(): TString
+HasTarget(target:TString): Bool_t
+HasSource(source:TString): Bool_t
+SetReadFunctionPointer(func:ReadFuncPtr_t)
+GetReadFunctionPointer(): ReadFuncPtr_t
+SetReadRawFunctionPointer(func:ReadRawFuncPtr_t)
+GetReadRawFunctionPointer(): ReadRawFuncPtr_t
+SetInclude(include:TString)
+GetInclude(): TObjArray
+SetRuleType(type:RuleType_t)
+GetRuleType(): RuleType_t
+Conflicts(rule:TSchemaRule): Bool_t
TSchemaMatch
+GetRuleWithSource(name:TString): TSchemaRule
+GetRuleWithTarget(name:TString): TSchemaRule
+HasRuleWithSource(name:TString): Bool_t
+HasRuleWithTarget(name:TString): Bool_t

Figure 19: *The C++ API for defining the conversion rules*

3.6.8 The C++ API concerning handling the buffering

Normal conversion functions accept as their input a proxified pieces of the on-disk data that were specified in the source properties. If the piece of data is of complex type its shape should be the same as the one on disk (before performing any conversion on this piece if the conversion rules for its type were defined). This is necessary since there may exist conversion rules converting a given source class to many different target classes and it may not be possible to determine the right target type automatically.

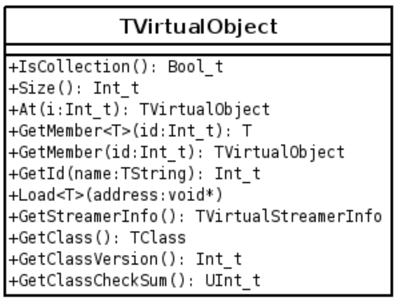
TVirtualObject
+IsCollection(): Bool_t
+Size(): Int_t
+At(i:Int_t): TVirtualObject
+GetMember<T>(id:Int_t): T
+GetMember(id:Int_t): TVirtualObject
+GetId(name:TString): Int_t
+Load<T>(address:void*)
+GetStreamerInfo(): TVirtualStreamerInfo
+GetClass(): TClass
+GetClassVersion(): Int_t
+GetClassCheckSum(): UInt_t

Figure 20: *An UML diagram of TVirtualObject class*

Figure 20 shows an UML diagram of *TVirtualObject* class. It has to be able to support the case when the input to the conversion function is a collection. Hence, it has *IsCollection*, *Size* and *At* methods. Data members or base classes could be accessed using *GetMember* method. The templated one is meant to retrieve basic types, since they can be accessed by just casting a memory cell to a given type. The one returning *TVirtualObject* should to be used with the concrete types since their memory printout cannot be safely casted to any type. The *GetMember* methods should get the integer id as the input parameters instead of the string names of the data members or the base classes for performance reasons (to avoid costly string comparisons). The ids should be class-wise and should not depend on the version of the class being read (ie. data member *m_mem1* should correspond to id 1, no matter if we are reading class *ClassA* in version 1 or 2 even though the class shape could change dramatically from one version to the other). The *Load* method is supposed to load the proxified data to the memory shape of an object of some existing type is it is possible. Other methods provide information about what hides behind the proxy.

3.6.9 The prototype

The actual prototype that has been implemented by the author of this document and by Philippe Canal provides a large part of the functionality described in the previous sections. The facilities for calling custom conversion methods are in present, but some of them are still missing in the split-mode. The buffering is done in a simplified way: ids are replaced with offsets of the data members and the *Load* facility that was meant to ultimately break the chain-effects as seen in the T/P separation is missing.

The results of the tests that has been done show that the time overhead caused by the new code for the case when the new schema evolution is not used is negligible and amounts to around 0.5%.

3.7 Outlook

The author's work with ROOT has shown that the system is able to successfully support the persistence of the data models used by the ATLAS experiment but that there are still many fields where the IO system can be tweaked to do the job in a more optimal way.

The algorithm used to handle the pointers is very general and able to handle the broadest spectrum of cases but, given the ATLAS specifics, it could be updated/extended to let the user choose if he wants to track the pointers pointing to the same objects or the pointer structures with cycles. Compression of the floating point numbers could also be enhanced, the data members of floating point types within one class could be grouped together so that only a user specified amount of bits in the base and the exponent could be stored in the optimal way. For reading back maps the insertion with hinting could be used when possible instead of doing searches every time when a new element is read back and the *std::bitset* class could be supported. These are all relatively small but necessary drops in the sea of the work that already has already been done.

4 Conclusions

As the reader could observe, the ATLAS data persistence system is very sophisticated. A lot of effort has been done to make it capable of handling very complex data structures. There are still some problems to be solved and the attempts are being made to provide the right solutions. The ATLAS solution to the schema evolution problem is fully operational and, as a side effect, makes the IO system working in more efficient way. All those improvements come for very high price though. It involves a lot of effort, mindless typing and arduous debugging resulting with a huge amount of code containing many redundant blocks.

This project has shown that the ROOT framework provides facilities on which a system providing similar functionality to that of the T/P separation can be built. In most of the cases such a system is or would be capable of performing the same or equivalent optimizations automatically. In the cases when the user code is required the system is flexible enough to make this code less redundant since one conversion rule can be applied to many versions of the input class.

There is still a lot of effort to be made to make the system functioning in the optimal way by enhancing the storage technology (the ROOT IO) to do what it is expected to do instead of trying to apply unmaintainable workarounds elsewhere that may lead to serious maintenance troubles in the future.

References

[1] F. Akesson, T. Atkinson, M.J. Costa, M. Elsing, S. Fleischmann, A. Gaponenko, W. Liebig, E. Moyse, A. Salzburger, and M. Siebel. ATLAS Tracking Event Data Model. http://cdsweb.cern.ch/record/973401/files/soft-pub-2006-004.pdf.

[2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional, 2001.

[3] ATLAS. Atlas Brochure. http://atlasexperiment.org/atlas_brochures_pdf/brochure_american.pdf.

[4] ATLAS. ATLAS computing: Technical Design Report. http://cdsweb.cern.ch/record/837738/files/lhcc-2005-022.pdf.

[5] Jonathan Bartlett. Programming from the ground up. http://download.savannah.gnu.org/releases/pgubook/ProgrammingGroundUp-1-0-booksize.pdf.

[6] CERN. CERN personnel statistics. https://webh12.cern.ch/hr-info/stats/persstats/CERNPersonnelStatistics2007.pdf.

[7] CERN. LHC - The Large Hadron Collider booklet. http://visits.web.cern.ch/visits/guides/tools/presentation/LHC_booklet-2.pdf.

[8] Edsko de Vries. Memory layout for multiple and virtual inheritance. http://www.phpcompiler.org/articles/virtualinheritance.html.

[9] The POOL Developers. Pool 2.6.0 user guide. http://pool.cern.ch/currentReleaseDoc/UserGuide.pdf.

[10] P. Bechtle et al. (L. Janyst). Identification of hadronic tau decays with atlas detector. ATL-COM-PHYS-2006-029.

[11] L. Janyst and E. Richter-Was. Hadronic tau identification with track based approach: optimisation with multi-variate method. ATL-COM-PHYS-2005-028.

[12] Lukasz Janyst. New EDM for taus. http://indico.cern.ch/getFile.py/access?contribId=2&resId=0&materialId=slides&confId=11467.

[13] Lukasz Janyst. T/P separation of Tracking. http://indico.cern.ch/getFile.py/access?contribId=8&resId=0&materialId=slides&confId=5121.

[14] A. Kaczmarska, E. Richter-Was, and L. Janyst. The track-based algorithm tau1p3p: integration with taurec and performance for release 12.0.5. ATL-COM-PHYS-2007-010.

[15] A. Kaczmarska, E. Richter-Was, M. Wolter, and L. Janyst. Performance of the tau1p3p algorithm for hadronic tau decays identification with release 12.0.6. ATL-PHYS-INT-2008-004 ATL-COM-PHYS-2007-039.

[16] LHCb. The GUADI Users Guide. http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/Gaudi/Gaudi_v9/GUG/GUG.pdf.

[17] Stanley Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[18] Conseil Européen pour la Recherche Nucléaire. Convention for the establishment of a European Organization for Nuclear Research. http://dsu.web.cern.ch/dsu/ls/conventionE.htm.

[19] E. Richter-Was, L. Janyst, and T. Szymocha. The tau1p3p algorithm: implementation in athena and performance with csc data samples. ATL-COM-PHYS-2006-029.