DISSERTATION

Event Data Modelling for the LHCb Experiment at CERN

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften unter Leitung von

a. o. Univ. Prof. Dr. Silvia Miksch
Institut für Softwaretechnik und Interaktive Systeme (E188)

in Zusammenarbeit mit

Dr. Pere Mato Vila
CERN, EP Division, CH-1211 Genf 23

eingereicht an der Technischen Universität Wien
Technisch Naturwissenschaftliche Fakultät

von

DI Stefan Roiser
Matrikelnummer 9125881
A-3424 Wolfpassing, Schubertg. 7

Genf, am 30. Sept. 2003

Abstract

In this thesis a new technique for event data modelling for the LHCb experiment at CERN will be presented.

The event model of a high energy physics experiment is one central ingredient of the experiment specific software. A coherent event model has to be shared between algorithms which will use it to process the data in several iterations from the raw data, obtained from the detector, to reconstructed analysable physics quantities. All software applications built on top of the experiment specific software will depend on a proper functioning of the event model.

It is expected that with the new Large Hadron Collider accelerator, currently in construction at CERN, an unprecedented amount of data, in the order of peta bytes per year, will be retrieved from the different experiments. This data has to be stored according to the event model. Due to the long lifetime of experiments over several decades the consistency and maintenance of a coherent event model has to be guaranteed.

Requirements for the description of the event model derive from implementation language constraints, such as object oriented programming techniques, or constraints setup by the experiment itself, like programming rules. In addition it has to be taken into account that new programming techniques or new programming languages for the implementation of the event model may have to be supported in the future.

To meet all the requirements a system for event object description was designed. This system requires designers of event objects to provide a single file with the description of their event objects in a high level language. From this description it will be possible to derive the actual implementations of the event objects and other information like their reflection information or documentation about the objects. Several possible outputs for different object oriented implementation languages will be discussed.

A high level language for the description of the event model will be proposed. The description language has to be defined as such, that it will be flexible enough to enable future enhancements but also strict enough to be able to constrain the language to a subset of the current features of object oriented programming, as not all of them are required for the description of the event model and may also not be provided in all implementation languages. A discussion of different possibilities for definition languages will elaborate the advantages and shortcomings of the most popular ones.

The features of the chosen description language together with its syntax will be presented. When using the tools, several generic and implementation specific rules will be applied, which will be explained.

As a practical example for the extension of the event data modelling tools, a software package which supports reflection in C++ was developed. This package is used for tasks like persistence of event data or interactive usage and will be discussed in depth.

The tools that have been developed as part of this research work have been evaluated both in terms of description language and usage in the experiment and their integration into the LHCb software build procedure will be explained. The event data modelling tools have been used successfully for several iterations of the description of the LHCb event model.

Kurzfassung

In dieser Doktorarbeit wird eine neue Technik der Datenmodellierung für das LHCb Experiment am CERN vorgestellt.

Das Event Modell eines Hochenergiephysik Experiments ist einer der zentralen Bestandteile der experimentspezifischen Software. Ein einheitliches Event Modell wird von Algorithmen benutzt um Daten in mehreren Iterationen von den Rohdaten aus dem Detektor in rekonstruierte und analysierbare physikalische Mengen überzuführen. Alle Softwareapplikationen eines Experiments bauen auf dem Event Modell auf und basieren auf dessem reibungslosen Funktionieren.

Es wird erwartet, dass mit dem Large Hadron Collider Beschleuniger, der zur Zeit am CERN gebaut wird, eine noch nie dagewesene Menge von Daten, in der Größenordung von Peta Bytes pro Jahr, von den einzelnen Experimenten produziert werden wird. Diese Daten werden gemäß dem Event Modell gespeichert. Mit der langen Laufzeit der Experimente, über mehrere Jahrzehnte hinweg, muss die Konsistenz und Wartbarkeit eines koherenten Event Modells garantiert sein.

Anforderungen an die Datenbeschreibung des Event Modells kommen von Beschränkungen in der Implementationssprache, zum Beispiel objektorientierte Programmiertechniken, oder vom Experiment selbst, wie eigene Regeln für die Codeimplementierung. Des weiteren müssen zukünftige Programmiertechniken und Implementationssprachen unterstützt werden.

Um alle diese Anforderungen zu erfüllen wurde ein System zur Beschreibung von Event Objekten entwickelt. Dieses System erfordert von den Entwicklern der Event Objekte ein einzelnes File mit der Beschreibung der Objekte in einer höheren Sprache. Von dieser Beschreibung ist es möglich Implementationen der Event Objekte in verschieden Implementationssprachen und weitere Informationen, wie Reflection oder Dokumentation, abzuleiten. Einige Möglichkeiten der Implementierung in objektorientierten Sprachen werden beschrieben.

Eine höhere Sprache zur Beschreibung des Event Modells wird vorgestellt. Diese Beschreibungssprache muss derart definiert werden, sodass sie für künftige Erweiterungen flexible genug ist aber auch Möglichkeiten bietet die Sprache einzugrenzen, da nur ein Teil der Charakteristika von objektorientierter Programmierung zur Implementation des Event Modells nötig ist und auch nicht alle Eigenschaften objektorientierter Programmierung in allen Implementationssprachen vorhanden sind. Die populärsten Definitionssprachen mit ihren Vor- und Nachteilen werden besprochen.

Die Eigenschaften der gewählten Definitionssprache zusammen mit ihrer Syntax werden präsentiert. Bei der Verwendung des Systems werden allgemeine und implementationsspezifische Regeln aus LHCb angewandt, die genau besprochen werden.

Als praktisches Beispiel für die Erweiterung des Systems wurde ein spezielles Softwarepaket für Reflection in C++ entwickelt, welches für Anwendungen wie Datenspeicherung und interaktive Analyse verwendet. Das Paket wird im Detail beschrieben.

Es erfolgte eine Evaluierung der im Rahmen dieser wissenschaftlichen Arbeit entwickelten Tools, sowohl was die Beschreibungssprache als auch deren Verwendung im Experiment angeht. Die Integration in die LHCb Software Build Prozedur wird erklärt. Das System wurde erfolgreich für mehrere Iterationen zur Beschreibung des LHCb Event Modells verwendet.

Acknowledgements

Contents

Chapter 1

Introduction

This thesis will describe a new technique for the description of the event model of the LHCb experiment at CERN. The event model of a high energy physics experiment is one of its central ingredients. Algorithms will use the event model to process data in several steps from raw data to reconstructed physics quantities. Furthermore the event model has to be maintained over a long time in a consistent and coherent way.

In chapter 2 an overview of software architectures in high energy physics will be presented, explaining where the event model fits into the overall design. Chapter 3 discusses the requirements of a language that will be adequate for developing the event model. The design of a system for describing the event model and the technical choices for such a system will be explained in chapters 4 and 5. The thesis will be concluded by a discussion of the actual implementation and an evaluation of the system in chapters 6 and 7.

The rest of this chapter will give a short introduction into the world of high energy physics (HEP) and especially to the European Organisation for Nuclear Research (CERN). In the first section some details about CERN and its relation to other HEP institutes will be presented, followed by a description of the experiments in preparation and especially the LHCb detector.

1.1 CERN

The European Organisation for Nuclear Research (Conseil Europénne pour la Recherche Nucléaire) is the worlds largest particle physics laboratory. It is located in Geneva, Switzerland and France. CERN operates with a yearly budget of 1.000 Mio. swiss francs which is raised by its 20 european member states. Furthermore CERN employs 2500 people (out of which around 1000 are scientific personnel) and collaborates with 500 universities and institutes and 6500 people world wide.

Together with laboratories like SLAC[1] or Fermilab[2] in the USA and DESY[3] in Germany, CERN is one of the most important laboratories for particle physics.

---

[1] http://www.slac.stanford.edu
[2] http://www.fnal.gov
[3] http://www.desy.de

1

Figure 1.1: Sketch of the LHC ring
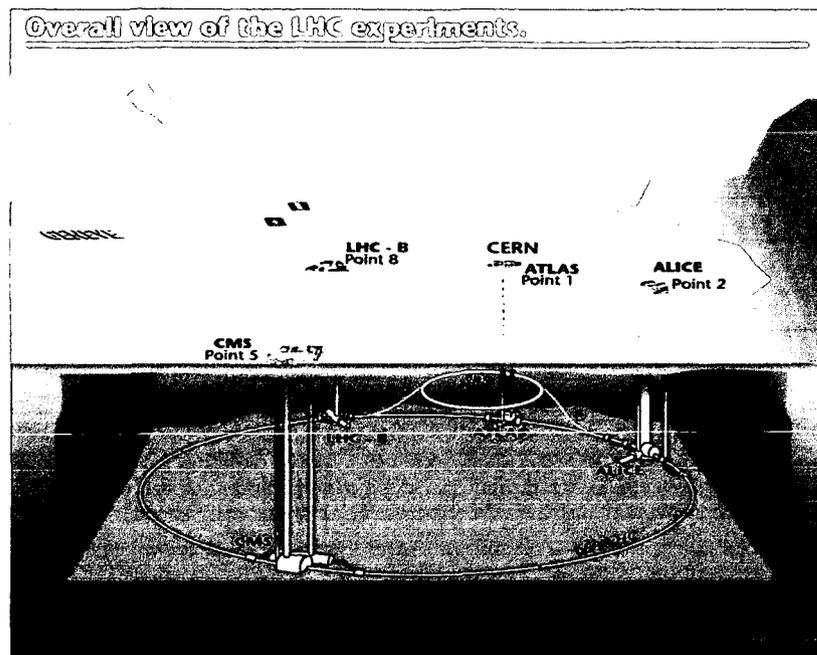
CERN provides accelerators and detectors for sub-atomic particles for high
energy physics experiments to physicists around the world. The experiments are
used to study the building blocks of matter or conditions which have happened
shortly after the big bang. Besides physics another important discovery at
CERN for daily life was the invention the HTTP [30] protocol and the HTML
[29] language by Tim Berners Lee in 1989 which led to the world wide web.

1.2   The Large Hadron Collider

The particle accelerator currently in preparation at CERN is called Large
Hadron Collider (LHC) (see Figure 1.1). It will be installed in a tunnel of 27 km
length which was constructed in both Switzerland and France and lies between
50 to 175 m below surface. The LHC will accelerate hadrons, i.e. particles
which are part of the atomic nucleus, nearly to the speed of light and collisions
of these particles will happen in 4 different detectors, namely LHCb[4], ALICE[5],
ATLAS[6] and CMS[7] (see Figure 1.2). The footprints of these 4 detectors can be
seen in Table 1.1.

---

[4] http://cern.ch/lhcb

[5] http://cern.ch/alice

[6] http://cern.ch/atlas

[7] http://cmsinfo.cern.ch/Welcome.html

Figure 1.2: The 4 LHC experiments
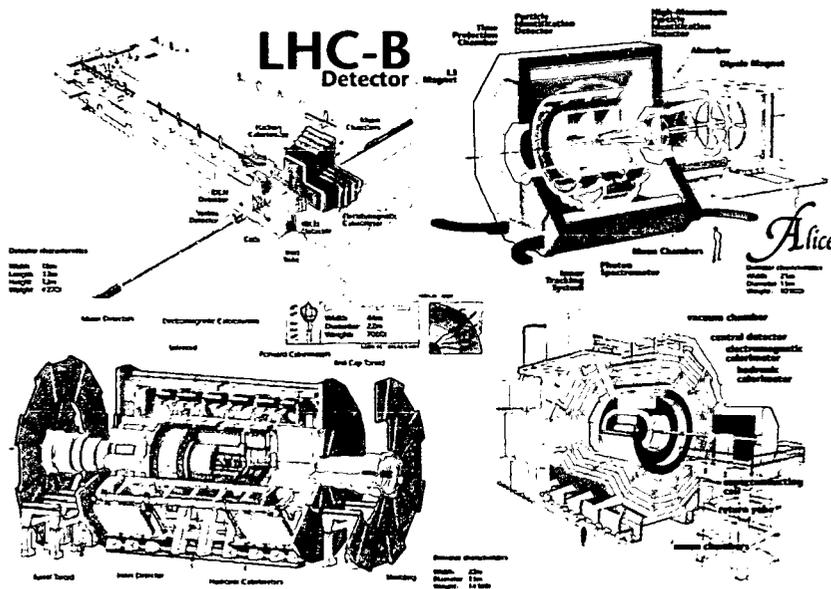
Table 1.1: Footprints of the four LHC experiments

	Width	Diameter	Weight
LHCb	18 m	12 x 12m	4.270 t
ALICE	25 m	15 m	10.000 t
ATLAS	44 m	22 m	7.000 t
CMS	22 m	15 m	14.500 t

After a beam crossing, i.e. a collision of particles, their decays into other particles will be measured in the detectors. The data from one of these collisions is called an event. Collisions of particles in the different detectors will take place at a rate of 40 MHz (i.e. every 25 ns). Due to this high frequency and the expected size of data per event coming from the detectors, the volume of data that has to be processed will be very high. It is expected that, after all processing and filtering of data is done, the four LHC experiments will generate several peta bytes of data per year which have to be stored onto a persistent medium.

The design of the LHC and the four related experiments started in the 1990's. The preparation of the accelerator machine and the 4 detectors should be finished in 2007. From the point of the inauguration of the LHC and its detectors the data taking phase should last for around 15 years.

Figure 1.3: The LHCb Detector
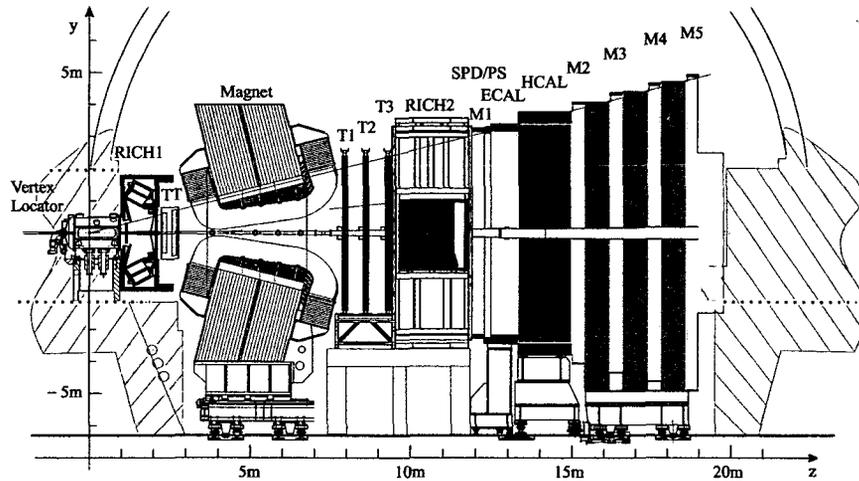
1.3   The LHCb Detector

LHCb (see Figure 1.3) is a detector designed to study rare decays of beauty
quarks. Due to peculiarities of the decays of beauty quarks, LHCb will measure
particle decays only into one direction while the other 3 detectors with the LHC,
which are designed for other purposes, will measure decays into every direction.

The LHCb detector is divided into several parts which are responsible for
different tasks.

- The Vertex Locator will measure the point of interaction between the two
  colliding particles.

- The magnet will bend the tracks of the charged particles which will ease
  the task of measuring the momentum of tracks.

- The remaining parts are sub-detectors specialised to the finding and lo-
  calisation of special particles which will be part of the events. These are:

    - The ring imaging cherenkov light detectors (RICH1, RICH2)
    - The tracking stations (T1 - T3)
    - The pre-shower station (PS)
    - The calorimeter stations (ECAL, HCAL)
    - The muon stations (M1 - M5)

The main challenges of LHCb are a good trigger performance which has to
filter from some 10 MHz of collisions to a few 10 Hz of fully reconstructible
decays. For this task a good particle identification will be required where the
RICH sub-detector will play a major role.

Chapter 2

Software Architecture

This chapter will give a short overview of software frameworks for HEP experiments in general and the software framework used for the LHCb experiment in detail. After this introduction the role of the event model for these frameworks will be discussed and it will be explained why the event model is one of the central parts of the framework software.

2.1 Software Layers in HEP Experiments

Software in an experiment will be divided into several parts. As an example for the different layers, the pieces of software in LHCb will be explained in more detail (see Figure 2.1):

Figure 2.1: LHCb Software Architecture
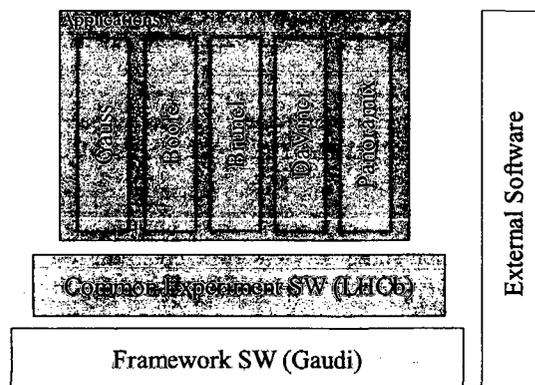
- The framework software (*Gaudi*) at the very bottom will provide basic functionality and services to the layers above.

- Physicists will then build the experiment specific software on top of it (*LHCb*). Examples for this kind of software are the event model or the detector description.

5

- On top of the experiment specific software the different applications are built. In LHCb these are:

  - The simulation software, *Gauss*
  - The digitisation software, *Boole*
  - The reconstruction software, *Brunel*
  - The analysis software, *DaVinci*
  - The visualisation software, *Panoramix*

- On every of these layers, developers may make use of external software. This external software provides functionality which is integrated into the framework and so does not need to be re-implemented. Examples for external software packages are:

  - *Boost* [10] for general purposes like implementations of smart pointers or call-back functions
  - *Xerces* [11] for XML parsing
  - *GSL* [27] for mathematical functions

2.2 Software Frameworks

For experiments which are as big as the four new ones for the LHC machine it will be necessary to provide a proper framework for the underlying software, providing some common functionalities and services, which will ease the writing of code for the developers of the different sub-detectors.

Software frameworks should perform the following tasks:

- Provide interfaces to basic functionalities. Interfaces will work as an abstraction layer which the developers will use to write their code. The implementation of an interface may change without the developer having to notice it.

- Insulate the programmer from underlying functionality of the software. Developers should not be bothered e.g. which current visualisation library or I/O library is used.

- Provide basic services, like error reporters or plug-in managers. Developers should be able to use these services in a coherent way throughout the software. The services should be easy to use and facilitate the writing of the current code.

- Provide base classes from which implementation classes may be derived. For example a base class called "DataObject" with some basic functionality for event data objects like streaming will provide a coherent behaviour of the event model of an experiment.

Also documentation and education is a major role of the software framework developers. Through the framework software and the people involved, developers from different sub-detectors should be able to learn new programming techniques and get up to speed with the software provided.

2.3 Gaudi

For the LHCb experiment the software framework is called Gaudi [13, 36]. In Figure 2.2 an overview of the architecture of the framework can be seen. In addition to the features listed above, the following design decisions were taken for the Gaudi framework.

Figure 2.2: The Gaudi Architecture Overview
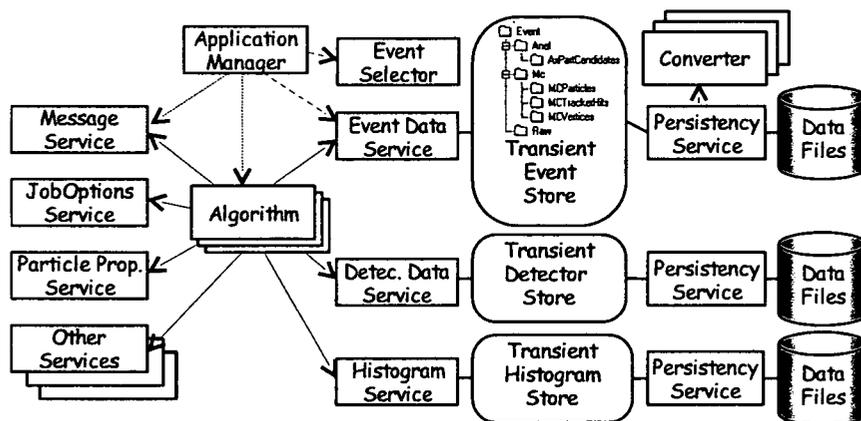
2.3.1 Separation between Data and Algorithms

Data and algorithms will be clearly separated in Gaudi. The functionality of data objects such as particles or tracks will be limited to manipulations of internal members.

Three different types of data were identified for Gaudi:

- *Event data* will store all kinds of data that comes from simulation or the detector to reconstruction and analysis data (see section 2.4.2). Each of these data types will be kept in the event data store.

- *Detector data* will describe the layout of the detector. This kind of data is more static than event data. It will be read once for a run. Detector data will be needed in all different stages of the processing of event data.

- *Statistical data* is all kind of data that provides information about the event or detector data. Examples for statistical data are histograms of energy distributions or other information of reconstructed or analysed data.

An algorithm will process data objects of a given type and may produce new data objects.

2.3.2   Separation of Transient and Persistent Data

Algorithms dealing with objects should only use the transient store of the framework and not have access to the persistent representation of the data. This decision was taken for several reasons:

- Most of the physics code should be independent of the storage technology for objects.

- Optimisation criterias for transient and persistent stores are different. While I/O performance and data size are important for persistence of data, optimisation of execution time is the major goal for transient objects.

The separation of transient and persistent data should be invisible for the user. A machinery for retrieving and storing data will provide the data no matter whether the user just fetched it from a persistent medium (e.g. tape) or the transient store in memory.

2.3.3   Data Store Centered Architectural Style

Data stores are an important architectural unit of the framework. The flow of data between algorithms will proceed via the transient stores. This will minimise the coupling between algorithms and allow their independent development.

The event data store of a software framework is responsible for storing all information about data that is acquired while simulating or running the detector. In general the major parts of the event data store are:

- *Simulation data* will contain all the data that is related to the software simulation of the detector.

- *Raw data* is either the output of simulation software or in real life the output of the detector.

- *Reconstruction data* contains the reconstructed tracks of particles and hints what kind of particles were involved in an event.

- *Analysis data* will contain the finally reconstructed particles which were part in the event and statistical data about them.

2.3.4   Encapsulation of User Code

Physics and sub-detector specific code shall be added to the framework in two main places:

Algorithms

The way how data will be read and stored from and to the data stores is done through algorithms. Algorithms are encapsulated pieces of software responsible for a special task.

Algorithms will retrieve the data from the transient data store process it
accordingly and write new data back into the store. Algorithms are the usual
way of interaction of physicists software with the data when processing it.

The event data store is the central piece through which algorithms may com-
municate with each other. Because algorithms should encapsulate some func-
tionality they may also be concatenated to a chain which will process the data
in several stages (see Figure 2.3).

Data in the Gaudi event store is available to everybody using the framework.
For this reason data that was once written for an event to the store should not
be removed or changed anymore, because other users of the event store may
rely on it.

Figure 2.3: Usage of Algorithms in Gaudi
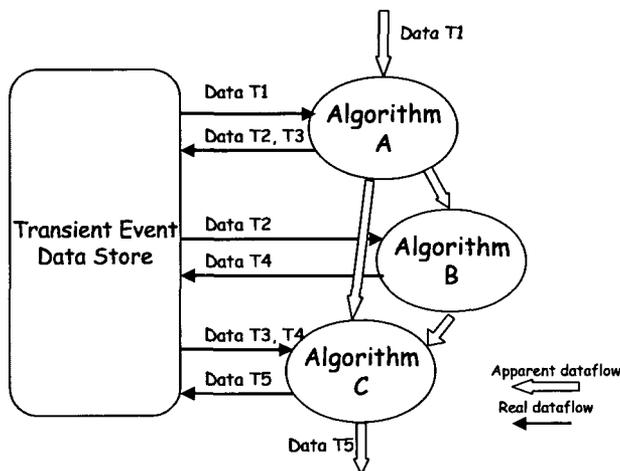
Converters

The second possibility for physicists to provide code is when writing convert-
ers. Converters are pieces of software that provide the functionality of storing
and retrieving data objects between the transient and persistent stores in the
Gaudi framework.

In most cases an automatic translation between the transient and persistent
representation of the data will be provided by the framework software, never-
theless in some specific cases these default converters will need to be overwritten
and specific ones have to be provided.

In case of the event data the serialising of one object is not done by an external
converter but the converter (i.e. serialising/deserialising method) is provided
as part of the object. When an event object needs to be made persistent or

transient the serialising/deserialising method of this object will be called which will do the needful to either write the object onto external medium or read it back from there.

2.3.5 Generic Component Interfaces

Each component of the framework architecture will implement a set of interfaces. An interface provides some generic functionality of a component which is independent of its actual implementation. A generic interface model should provide support for interface versioning, dynamic interface discovery and generic component factories which will allow run-time loading of components.

2.4 Event Data

The Gaudi framework distinguishes three different kinds of data, one of them being the event data. This data will hold all information which is related to particles and their way through the detector.

2.4.1 Past Practices for Event Data Description

In some previous experiments the event data models were not described directly in code but with means of a higher level description. E.g. in ALEPH, a previous LEP experiment, the event data model was described with a software called ADAMO [1, 44]. This software was not only responsible for the description of the event data model but also partly for its processing and provided bindings to other programming languages and applications like Microsoft Access or Matlab.

A part of a description of an event object with the DDL language used in ADAMO can be found in Listing 2.1. The syntax for the description of event objects was especially designed for this purpose. With a specially written parser the files were processed and the event model was filled. Having the advantage of an especially for this domain designed language the disadvantage was that every tool for the processing of the data had to be written by the developers of ADAMO and they could not use already existing tools like e.g. language parsers.

ADAMO was not only used in the ALEPH experiment at CERN but also in several other high energy physics experiments for data definition.

2.4.2 From Simulation to Reconstruction in LHCb

The relation between the different types of particles contained in one event may be seen in Figure 2.4. The left part of the figure shows the simulation part of the data while the right part shows the particles which will be used both in simulation and real data taking. The processing of the information from the simulation to the reconstructed data will be done in these steps:

Listing 2.1: DDL Description of Aleph Muon Sub-Detector

```
SUBSCHEMA MuonRAWGALJULPOT

AUTHOR    'G. Capon, G. Taylor'
REVIEWER  'F.Loverre'
VERSION   '4.0'
DATE      '28/03/95'

DEFINE ESET

PHMA
         :        'Hcal Muon tracks Association Data (POT)\
                  Number of words/associated track\
                  Number of associated tracks'
              STATIC

= (MultHits  = INTE [0,30]        : 'number of clusters in last\
                                      ten planes',
   IGeomflag = INTE [-1,6]        : 'flag of possible dead zone',
   EnerDep   = REAL [0.0,100.]           : 'energy deposit in nearest\
                                      Hcal storey',
   ChiSquare = REAL [0.00,200.00] : 'chisquare',
   NumbDeg   = INTE [0,23]        : 'number of degrees of freedom',
   IExpbmapInHcal  = INTE [0,11000000] : 'expected bit map in Hcal',
   ITruebmapInHcal = INTE [0,11000000] : 'observed bit map in Hcal',
   IdenFlag  = INTE [-1,1]        : 'preliminary identification flag\
                                      1=muon,0=not classif,-1=hadron',
   TrackNo   = INTE [1,999]       : 'index of associated track ')
      ;

MUHT
         :        'MUon detector HiTs.
                  NR=0. (GAL) \
                  Number of words / hit\
                  Number of hits'
              STATIC

   = (TrackNum   = INTE [1,999]   : 'track # that generates the hit',
      Electronics = INTE [1,99]   : 'electronics module #',
      StripPlane  = INTE [1,4]    : 'strip plane # in this electr. module',
      StripAddres = INTE [0,700]  : 'strip address in this strip plane\
                                      ( start from 0 )')
      ;

END ESET

END SUBSCHEMA
```

- *GenParticles* are the output of the simulation of the primary collision of particles. This simulation will be done with special software. The most widely used software packages for this task are called Herwig [28] and Pythia [42].

- The results of this first step will then be fed into a software which simulates the reaction of particles originating from this primary collision with the materials of the detector. Particles flying through the detector in this step are called *MCParticles*. The output of this simulation software is called *MCHit*. An MCHit is the response of a simulated sub-detector to an MCParticle traversing it. A popular software for this task is called GEANT4 [24].

- The next step of the processing of the event data is called digitisation. This will simulate the response of the detector hardware to the MCHits. An intermediate step here is called *MCDeposit* which resembles the amount of energy that a particle has left while traversing a sub-detector. The output of this step is called *MCDigit*. MCDigits are the last step of the simulated event.

- A step that is not mentioned in the picture is the trigger step. In real life triggers will be used to reduce the amount of data that comes out of the detector and to filter those events which are interesting for further

Figure 2.4: Relations of Event Data Types
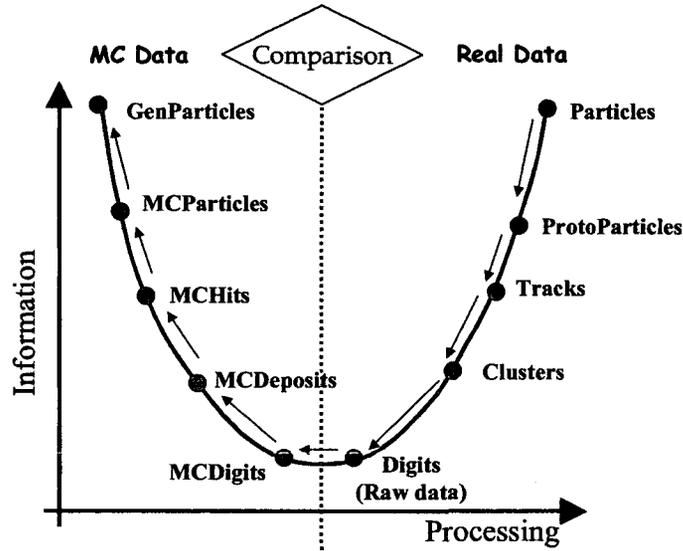
investigation. Triggers will work with digitised data and also return digits. There are 3 different kinds of triggers. While the *level 0 trigger* is implemented in hardware, *level 1 trigger* and the *high level trigger* are implemented in software.

- From this step on, called *Digits*, the processing of the simulated and real data will be the same. Digits are the same as MCDigits but they are needed to function as a starting point for the subsequent steps. In real life Digits will be the output of the different trigger levels of the experiment. A synonym for Digits is *Raw Data*.

- The *Raw Data* will be fed into a reconstruction program which will use the information that comes from the high level trigger to reconstruct the tracks of the different particles that were part of a given event. An intermediate step in this reconstruction of the tracks is called *clustering*. Major problems that can occur in this step is the possibility that more than two particles were involved in the primary collision, which is called pile up, or that slow particles from a previous collision are overtaken by fast ones from a subsequent one, which is called spillover.

- The reconstruction program will not only try to reconstruct the *tracks* of the particles but will also give the a 'tag' which will be a first hint of the type of the particle. The output of the tagging step is called *ProtoParticle*.

- In the end an analysis software will use the information that was provided by the reconstruction program to determine the types of *particles* that were involved in the interaction. Now that the analysis software knows

what kinds of interactions took place during a given event it may put this event into a pool of events which only stores interactions of a given type.

With each of the steps on the right side of the curve, special algorithms are involved which will do specific tasks for a sub-detector. For example for the reconstruction step there are specific algorithms of the different tracking stations which will process the raw data and provide their information to the overall finding of a track of a particle from a collision.

While simulating the experiment it is also crucial for the event model that there will be no connection from the simulated data (on the left side of the figure) to the 'real' data (on the right side). The only connection when processing the data is allowed from MCDigit to Digit to ensure that the reconstruction and analysis of the simulated data will resemble conditions of real data taking.

2.4.3 The Transient LHCb Event Store

For the transient event store (TES) in LHCb it is important that the data may be accessed quickly and can be shared between different users of the store. The TES is treelike organised (see Figure 2.5). Every algorithm in LHCb consuming or writing event data will pick it up from and put it to the TES. Under the top Event there are several subtrees:

Figure 2.5: Structure of the LHCb Transient Event Store
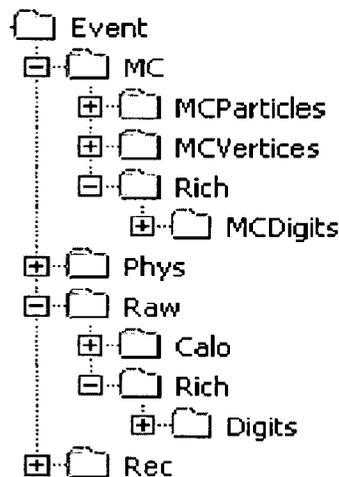
- *Gen* will contain the output from the event generators. During the simulation phase of the detector this subtree contains the output of the software which produces simulated events.

- The *MC* subtree will contain the output of the simulation of the particles running through the detector with GEANT4. This tree will for example

contain the MCParticles and MCVertices which are the output from the simulation software.

- The *Raw* subtree contains the raw information of the detector output or its equivalent of the simulated data.

- The *Trig* subtree will contain the data after the Trigger has processed them and accepted. Several levels of trigger decision are involved when processing the data.

- The *Rec* subtree will contain the data after the reconstruction step. A reconstruction program will try to reconstruct the events that where accepted after the trigger step. Data that this program produces will contain the track of the particles and a proposal for their type.

- Finally the *Phys* tree contains the data that was processed by the analysis software. The analysis software will read the reconstructed events and also perform some statistical processing on this data.

2.4.4 The Persistent LHCb Event Store

For the persistent event store the major problem is how to store the data in an efficient way. Storage should be efficient both in terms of compacting information and the time needed to write and retrieve the data from the external medium.

A problem that occurs when handling the persistent event store, is that most of the particle informations are tiny objects which need to be grouped together accordingly. When making data persistent a clever mechanism of compacting the information which is stored in lists or vectors in the transient world, needs to be found.

In LHCb a mechanism for implementing these features of compacting small objects was implemented. If needed the objects will be serialised into binary blobs which are passed to the database system which is responsible for the storage of the data. Together with some meta information about the objects stored in the database it will be possible to restore the objects properly into memory when they are read back.

The current persistent technology for Gaudi is called ROOT [9, 48] but also other persistent back ends will be possible like database systems (e.g. Oracle [40]).

Chapter 3

Data Modelling Requirements

In this chapter all the necessary requirements for the description of event objects will be listed. Requirements arise from the people and the organisation where it is used, from the constraints of programming languages used in the experiments and philosophies and conventions developed by people in the experiment.

3.1 Long Lifetime

The first plans for the LHCb experiment were made in the mid 1990's [34], data taking should start in 2007 and will last for approximately 15 years. Including the planning and construction phase, the LHCb experiment is supposed to run for several decades. Even the preparation phase of the experiment will last for more than a decade. In this respect the durability of the described data is important. For the whole period of preparation and data taking the coherence and the maintainability of the data described in the event model has to be guaranteed.

3.2 Schema Evolution

In order to guarantee a coherent event model through time, a mechanism for versioning of documents has to be implemented. This mechanism will be outside the scope of this thesis but the object descriptions in the event model will have to cooperate with it. This means that every document which describes an event object will also have to include some information which will make it possible to trace the version of it and with this information provide means to compare it to other versions of the same object description. Schema evolution will be important when data, that has been written several years ago with different software and a different layout should be read back into another program and fill objects in this program appropriately.

15

3.3   Programming Language Independence

As the experiment software will also continue to evolve when the experiments are running, new programming languages are likely to come up which are more adequate with better functionality and suit better for given tasks. In order not to re-implement the event model every time a new language is introduced it would be important to describe the event model with a higher level language from which concrete implementations can be derived.

This description of the event model has to be done on a higher level, as future languages may not use features which are part of the implementation of the model today or may implement new features. For this reason the description has to be essentially limited and reduced to the main functionality of the model.

It has also to be kept in mind that not only the implementation language of the event model but also the implementations of the underlying framework software and the algorithms directly using it, may change at the same time, and they may also evolve at a different speed.

3.4   External Dependencies

The event model per se will not be self contained. It will depend on other pieces of software which are either provided by the framework software or are completely external and provided by third parties. For this reason it will be useful if the tools for handling the event model will have some knowledge about the existence of software which is not part of the model itself and include it at the proper places. For this purpose a database with the information about all externally available software will have to be built and provided when the tools for handling the event objects are executed.

3.5   Flexibility of the Software

Changing the software for the description of the event model should be possible in a flexible way. Due to the long lifetime of the experiment many different people will be involved to maintain the functionality of the implementation of the event objects. Changing parts of the model or implementing new features should not be too difficult and easily adaptable.

3.5.1   Changing Implementation

During the lifetime of the experiment it will certainly occur that different implementation techniques will evolve which for example allow a faster or more efficient implementation of the event model. Instead of changing the implementation of each object it would be useful if the overall design of the implementation was stored in only one place were it could be changed easily. After having done these changes one only needs to regenerate the whole model with the new functionality to obtain the new features. This could be useful for example when new techniques for object containers, which are used frequently in the model, are designed.

In this sense the object description techniques will function like a style sheet which stores the overall design of the model decoupling the implementation details from the list of objects describing the model itself.

3.5.2 Evolution to New Languages

Not only the implementation details for a given language may change but also new implementation languages for event objects may be used in the experiment software. In this case it should be easy to generate the object descriptions also with these new languages which may support different features than previous ones.

3.6 Easiness of Design

Physicists describing event data should not be bothered with complex implementation languages which are difficult to understand. The goal is to either create a language which is easy to understand and to learn with a simple syntax or use a language that users are already familiar with. Also the syntax with which the event model is described should be straight forward and reflect the layout and relations of the objects involved.

3.7 Short Descriptions

Data descriptions in concrete implementations are often verbose and so error prone to implement. E.g. in C++ in most cases the implementation of a data member in a class also requires declaration and definition of a set- and a get-method. In addition information of this member may also appear in other places of the implementation of a class, e.g. the streaming functions.

Listing 3.1 shows an object description in C++ for two data members. For describing these two members a file of at least 120 lines will be needed. What is not shown in this listing are all additional lines which typically also appear in a C++ header file like the description of the class, include files, header guards or cvs information.

Listing 3.1: Example class containing two members

```
class ExampleClass : public KeyedObject<int>
{

public:

    /// Default Constructor
    ExampleClass() : m_momentum(0.0, 0.0, 0.0, 0.0) {}

    /// Retrieve 4-momentum-vector
    const HepLorentzVector& momentum() const;

    /// Retrieve 4-momentum-vector (non-const)
    HepLorentzVector& momentum();

    /// Update 4-momentum-vector
    void setMomentum(const HepLorentzVector& value);

    /// Retrieve Vector of pointers to decay vertices (const)
    const SmartRefVector<MCVertex>& endVertices() const;

    /// Retrieve Vector of pointers to decay vertices (non-const)
    SmartRefVector<MCVertex>& endVertices();

    /// Update Vector of pointers to decay vertices
```

```
   void setEndVertices(const SmartRefVector<MCVertex>& value);

   /// Add Vector of pointers to decay vertices
   void addToEndVertices(const SmartRef<MCVertex>& value);

   /// Remove Vector of pointers to decay vertices
   void removeFromEndVertices(const SmartRef<MCVertex>& value);

   /// Clear Vector of pointers to decay vertices
   void clearEndVertices();

   /// Serialize the object for writing
   virtual StreamBuffer& serialize(StreamBuffer& s) const;

   /// Serialize the object for reading
   virtual StreamBuffer& serialize(StreamBuffer& s);

   /// Fill the ASCII output stream
   virtual std::ostream& fillStream(std::ostream& s) const;

private:

   HepLorentzVector         m_momentum;      ///< 4-momentum-vector
   SmartRefVector<MCVertex> m_endVertices;   ///< Vector of pointers to decay vrtcs

};


inline const HepLorentzVector& ExampleClass::momentum() const
{
  return m_momentum;
}

inline HepLorentzVector& ExampleClass::momentum()
{
  return m_momentum;
}

inline void ExampleClass::setMomentum(const HepLorentzVector& value)
{
  m_momentum = value;
}

inline const SmartRefVector<MCVertex>& ExampleClass::endVertices() const
{
   return m_endVertices;
}

inline SmartRefVector<MCVertex>& ExampleClass::endVertices()
{
   return m_endVertices;
}

inline void ExampleClass::setEndVertices(const SmartRefVector<MCVertex>& value)
{
   m_endVertices = value;
}

inline void ExampleClass::addToEndVertices(const SmartRef<MCVertex>& value)
{
   m_endVertices.push_back(value);
}

inline void ExampleClass::removeFromEndVertices(const SmartRef<MCVertex>& value)
{
   SmartRefVector<MCVertex>::iterator iter =
     std::remove(m_endVertices.begin(), m_endVertices.end(), value);
   m_endVertices.erase(iter, m_endVertices.end());
}

inline void ExampleClass::clearEndVertices()
{
   m_endVertices.clear();
}

inline StreamBuffer& ExampleClass::serialize(StreamBuffer& s) const
{
   KeyedObject<int>::serialize(s);
   s << m_momentum
     << m_endVertices(this);
   return s;
}

inline StreamBuffer& ExampleClass::serialize(StreamBuffer& s)
{
   KeyedObject<int>::serialize(s);
   s >> m_momentum
     >> m_endVertices(this);
   return s;
}

inline std::ostream& ExampleClass::fillStream(std::ostream& s) const
{
   s << "{ "
     << " momentum:\t" << m_momentum << std::endl;
   return s;
}
```

3.8 Ability for Constraining the Language

As the LHCb software framework is written in object oriented style, the event model should also be capable of reflecting these concepts. But not all capabilities of current programming languages need to be reflected in such a description language. While concepts like inheritance would be essential to implement, other concepts like abstract interfaces are perhaps not necessary for a data model.

In this respect it will be useful if the description of objects can be constrained so that not all possibilities of a language can be used.

3.9 Modelling Relations

There are data modelling features that are not reflected in current object oriented programming languages directly, for example the distinction between data members which are holding some data of an object and relations which link to other parts of the event model. E.g. in C++ there is no distinction between relations between objects and members of the object.

Modelling relations may be further divided into 1 to 1 and 1 to many relations. This distinction is important for the design as different implementation techniques for these two kinds of relations may be used.

In the object oriented world the distinction between 1 to 1 and 1 to many relations will be done by either pointing to one instance of an object directly or by holding a list or vector of pointers to instances, where the choice of using what kind of container is an implementation detail.

- A 1 to 1 relation for example would be the vertex to mother particle relation. Each vertex can only be related to one mother particle.

- An example for 1 to many relations is a vertex to decay particles relation, where from a given decay many different particles may evolve. There will be no further distinction on the cardinality of relating objects (i.e. it is not important how many decay particles evolve from a given vertex as long as they are more than one and they can be followed when traversing the model).

In principal it should also be possible to model other kinds of relations like many to many relations, but for the description of the event model the 1-1 and 1-many relation are enough.

The modelling of relations in the event model is a crucial part. When using the model through algorithms or interactively, it will be important to traverse the event structure quickly and in a transparent way. Relations between objects will also play a special role for persistence of objects (see section 3.10).

3.10 Persistence

It is expected that the detectors of the LHC machine will produce several peta bytes of data every year. In this respect it will also be important to store the intermediate data at various steps of the processing pipeline onto persistent media, e.g. tapes. Storing the content of the event model bears several problems:

- The different objects of the event model are related to each other. It will be crucial to store these relations between the different instances of the objects so these relations may be restored once the data will be read back into memory. Only relations which can be made persistent have to be used when implementing the event model.

- Due to the huge amount of data it will also be necessary to compact the data as much as possible once it is written onto the persistent medium. Most of this work will be done by the mechanism which is responsible for the persistence of the data but the layout of the event model may also support this.

- Also due to the high amount of data it will be necessary to access and write the data quickly. There are two main aspects for speed when accessing data. Once the underlying hardware will be responsible for the basic speed of the operation. On the other hand data structures may also optimise and facilitate the access to the data. Implementing the event model with data structures which allow fast access will be important.

3.11 Data Modelling Constructs

The means for describing the event model do not have to be sophisticated as the underlying model is also not complex. Also concerning portability to other languages it will be an advantage if the descriptions themselves are not bound to specific features of a given implementation language. Principles of data modelling for objects are described in [7, 50].

3.11.1 Objects

An object is the basic entity of a data model. An object is used for representing a concept or abstraction of a set of functionalities and states which are inherently bound together. The way how objects are organised and grouped together will always depend on the problem at hand. It will not make sense to split an object into smaller pieces. Objects provide an abstraction layer of a real world problem and the practical basis for a computer implementation.

Objects always have an identity and are distinguishable. This means that they will be distinguished by their existence and not the properties they have. Object may be used for either denoting a single instance or a group of similar objects. In case of a single instance they will be called object instances in case of a group of similar objects they will be called object class.

The concept of objects is fundamental for a the event data modelling and so should be part of the description language.

3.11.2 Data Members

Each object may contain a certain number of properties which are called data members of the object. One way to distinguish data members is whether they are part of the object or point to other objects.

- *Composite data members* are part of the object and the object also controls their lifetime. They are initialised when an instance of an object is constructed and deleted when the instance goes out of scope.

  In most cases composite members are not accessible from outside the object. Access is only possible through special set- and get-functions.

- *Aggregate data members* are not controlled by the object itself. A relation between the object and the aggregated member exists, but when the object will be deleted the aggregated member will not be deleted as well.

Most object oriented programming languages will not make a distinction between composite and aggregate members. The difference between these two types of members is crucial for the design of the event model, because it allows to introduce means to treat composite and aggregate members differently.

Another way how to distinguish data members is in terms of complexity.

- *Basic types* are most times supported inherently by the language itself. Examples are integer, floating point, boolean or character types. Usually these types will be stored within the class and the class will also be responsible for the lifetime of the object.

- *Complex types* are more sophisticated than basic types, which also provide more or special functionality. In general they are not provided by the implementation language itself but will be built by the developers of the event model. As these types are usually larger in size than basic types, if used as object members, they will be stored most times outside the scope of a class and also the class will not control their lifetime.

Data members will store the information contained in the event model, they are another fundamental part of the description language.

3.11.3 Member Methods

Member methods provide the functionality of an object. In case of the event model they will be used in most cases for the manipulation of data members. In few cases methods will also be used for other tasks like serialisation of objects or output of information about the current instance.

Many of the methods can be generated automatically by the tools supporting the data description language. In some cases special descriptions for methods will be needed. For these special cases it will be necessary to describe these methods in a generic way with all its arguments, return value and special modifiers, e.g. the constness of the method or the ability to modify the implementation of the object in subsequent classes.

Although not always necessary in some special cases also the implementation code of the method should be provided. This could be useful because certain languages, e.g. C++, provide the concept of inlining of code, where the compiler will replace the function call with the implementation code of the method, saving a stack call and so speeding up the execution. The implementation code is of course bound to a specific implementation language and needs to be replicated once a new implementation language is introduced. This is one of the few cases of the description of the event model were a binding to specific implementations languages makes sense.

Member methods are another basic concept for the implementation of the event model and should be part of the description language.

3.11.4   Bit Fields

To encode a set of flags which can be either true or false, bit fields should be used. Bit fields are an efficient means to store binary information in a single data member because the binary information is compacted to the bitwise information it really contains. If the information was stored with other types like boolean or integer the storage of this information would take much more space. Bit fields also provide efficient means to get and set the information they are containing.

As there will be a lot of small information in the event model, the concept of bit fields should be represented in the description language.

3.11.5   Enumerations

Enumerations are types that can hold a set of values specified by the user. This concept was also introduced to make the code better readable by humans. Enumerations will set aliases of strings to numbers. These strings may be used in the code and will be replaced to the actual numbers by the preprocessor. An example for an enumeration in C++ can be seen in Listing 3.2.

Listing 3.2: Example Enumeration

```
1    enum Quark {down=1, up, strange , charm , bottom , top};
```

The concept of enumerations and constraining a type to a given range of values is useful when implementing data models. For this reason it should be a requirement for the event modelling language.

3.11.6   Typedefs

To increase the readability of the code it will also be desirable to have some means of typedef'ing complicated constructs of code to some aliases. These aliases will be replaced by the compiler but will increase the readability of the code which is crucial for large software frameworks where many different developers are involved. Example for a typedefs in C++ can be seen in Listing 3.3.

Listing 3.3: Example Typedef

```
1    typedef std::vector<std::pair<int,double> > PIDInfoVector;
2    typedef std::pair<int,double> PIDInfoPair;
```

For the reason of better readability of source code, the concept of typedefs should also be part of the description language.

3.11.7 Encapsulation

Encapsulation of members (i.e. data members and member methods) is a usual concept of object oriented languages. For example hiding the values of member variables in a private section of the object and only setting and getting them through public functions or constructors is supported by every modern object oriented language and so should also be part of the description language.

In general data members of an object may be encapsulated in three different parts of the object:

- *Private*: Everything that is put in the private area may be accessed only by the object itself. This means that for example data members in the private section may only the retrieved, changed or set by methods which are part of the object. In general most data members of an object are hidden in its private section. In this case it is also usual that for each data member there is also one public get and set function provided by the object containing it.

- *Protected*: Members in the protected section may be accessed by the object itself and all objects which are derived from it. This concept provides more access to the functions and members but still restricts it to objects which are related to each other.

- *Public*: Members in the public section are accessible from outside the object. They are not protected by the object anymore. The functionality of an object through its member functions is usually put into its public section.

Encapsulation is part of every object oriented language. It increases the security of the implemented objects and their members. For this reason it should be used for a data modelling language.

3.11.8 Inheritance

Inheritance is part of every modern class oriented language. Inheritance defines the relation between two different objects. A base class which provides a given set of functionality and a sub class. The sub class will inherit all the functionality of the base class and together with its own features provide a new object. Inheritance will work with an arbitrary number of layers. In different object oriented languages there are several variations of inheritance available.

- *Single inheritance* is the basic type of inheritance which is provided by every object oriented language. Single inheritance means that an object may only inherit functionality from one base class.

- With *multiple inheritance* a class may derive its functionality from several different base classes. Most object oriented languages will not support this feature.

- The concept of *virtual inheritance* only makes sense in connection with multiple inheritance. In case of several layers of inheriting objects it may occur that a given class derives its functionality from the same base class through more than one path of inheritance. When creating an instance of one of these classes, with virtual inheritance the ambiguous base class will only be instantiated once, otherwise several copies of the same class will be kept in memory.

Single inheritance is the smallest common denominator for all object oriented languages. As the structure of the event objects is not complicated, single inheritance should also be sufficient for its description and more complex concepts like multiple or virtual inheritance shall not be used.

3.11.9 Polymorphism

A class inheriting from another one has at least the same functionality as its base class. Polymorphism allows the inheriting class to overwrite parts of the functionality which is provided by the base class.

Polymorphism is provided by most of the object oriented languages and so should also be part of the description language for event objects.

3.12 Use of Coding Conventions and Rules

As there are many people contributing to the software in large experiments it is needed to establish some conventions how source code shall be written and organised. For the LHCb experiment this was done for example for C++ in several technical notes [5, 12]. The tools producing the source code automatically should obey the coding conventions as much as possible. There will be some places where automatic generation of code will not be able to follow the conventions (e.g. "use meaningful names for variables").

3.13 Uniform Layout and Readability

Readability is also an important feature of the output of the tools used for describing the event model. HEP experiments are supposed to run over several decades, in this time many different people will contribute to the code and also need to understand code that was written by other collaborators. In this sense the output of the code generation tools producing the implementation of an event object should all be produced with an easy to understand and coherent layout in the source files.

This will ease the task of reading the source files and understanding their content but also ease the task of debugging the sources if needed. Producing the same layout for all objects will give also confidence to the software developers when reading object source files that they can find the different parts of the implementation at well defined places and they will not need to scroll through the file or search for some keywords.

3.14 Distributed Development

Due to the distributed organisation of physics collaborations for large HEP experiments also the development of the software for these collaborations will be organised in a distributed way. Many developers around the globe will be involved into the generation and development also of the event model. In this respect it will be necessary to provide means to divide this development into different parts which may be put together in the end.

The granularity of the different pieces of the event model has to be of such a size that they may be developed by different people which are not working at the same place. On the other hand the granularity of the objects that will be defined should not be too fine grained, so the overhead for their development is not too big.

3.15 Documentation

From the higher level description of objects it should also be possible to extract some documentation information. There are two ways how this information can be extracted.

- The information may be extracted directly from the source of the description of the objects. This involves some tools which will process the descriptions of objects and generate some documentation. Useful output formats are for example HTML [29] or LaTeX 2_ε [25].

- The other possibility is to produce the documentation information together with the source code in the comments that will be attached to classes, functions or members in the concrete implementation files. In a second step this source code can be processed by tools which will understand the documentation style and then generate the desired documentation.

  A well known tool for producing reference documentation from source code for example is doxygen [18]. In many experiments tools like doxygen are used for hand written code, so if the tools for code generation will also obey these rules for documentation generation, the documentation of user written and automatically written code may be combined and browsed together.

3.16   Generated Output

It will be useful if the amount of data for describing the model will be less than the amount of automatically generated code. This is also important as the physicists describing the event model should not be bothered with tedious techniques or a large overhead in the syntax of the description language.

It can be seen in many concrete implementation languages that a file containing some implementation code contains many redundant bits of information which can be easily compacted. In C++ for example the information about a member variable also appears in the set- and get-function, as well as the constructors and destructors of the object and several other places like serialisation methods.

An example for this can be seen in Listing A.3 where for example the member variable "m_momentum" appears 11 times in the file. It would be useful if the variable would only appear once in the corresponding description file.

Chapter 4

Design

In this chapter the overall design of the model for describing event objects on a higher level and generating different implementations and other information from it will be described. In the first part a course description of the overall design will be presented, followed by the sections describing the different parts of the system in more detail. The chapter will be concluded by a section describing software which is closely related to the system.

4.1 Overall Design

The overall design of the tools for developing and processing the event model description (see Figure 4.1) was divided into three parts.

- A *front end*, which will parse the object model definitions. The front end will also check whether the descriptions of the objects are syntactically correct and issue warnings and errors if needed.

- An *internal representation* of the object model to the system will be filled by the front end parser which is the central source for all subsequent steps.

- *Back ends* will use this memory representation to produce the required output. These back ends will produce a representation of the data in a given implementation language (e.g. C++) but can also produce other information such as a meta representation of the data, used for reflection or documentation of the objects.

4.2 Front End

The front end will parse text files of object descriptions written by the users. These object definitions are the only input to the system that the users have to maintain. The goal was to define a language that describes objects on a higher level and does not need to be changed when new back ends are implemented. With this technique the long lifetime of the object description will be guaranteed.

27

Figure 4.1: Object Description Framework Overall Design
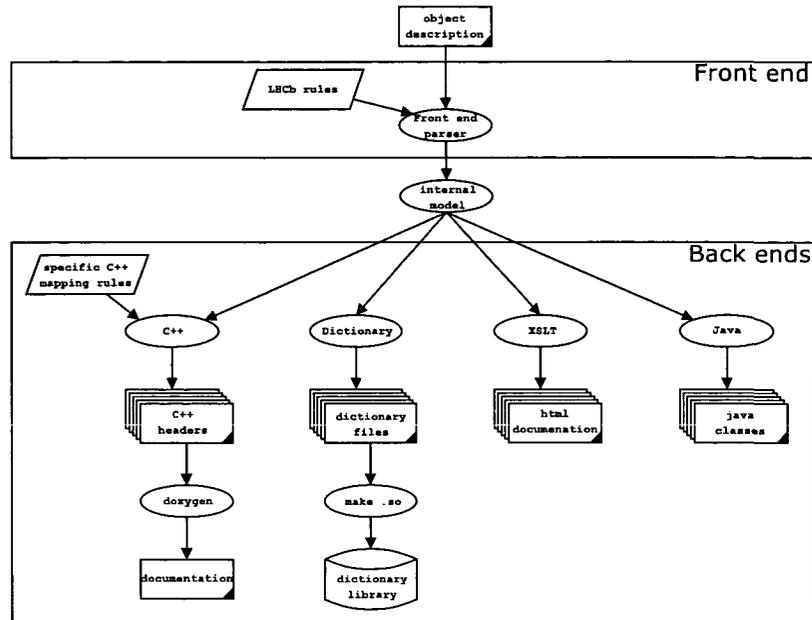
When filling the in memory model, the front end will do some lexical and syntactical parsing of the description language and report when the constraints are not met.

The front end will also be able to check additional rules which are not related to syntactical or lexical parsing of the description language (see "LHCb rules" in Figure 4.1). These rules are specific to the LHCb experiment and to the policy of how the event model should be generated in a general way. An example for these specific rules is that every class, attribute and relation should also contain some description.

4.2.1   Parser

For checking the definition language a parser will be needed. The choices are:

- Either to write a new parser which can do a processing of the language which is specific to the needs of the definition language. Defining a specific language and writing all the tools will give the developers the possibility to really tie down the language and tools to the specific needs for high level descriptions of event objects. On the other hand all these tools need to be designed, developed and maintained which will be an overhead which could be avoided.

- The other possibility is to use a language and parsers which are already defined and implemented. In this case the language needs to be flexible

enough to be extended and adapted, to the problem domain. A drawback in this case might be, that one depends on the development of the tools of third parties and once bugs or flaws in the design have been spotted it might take some time until those are resolved and new versions of the tool will be released.

4.3 Internal Model

After all general rules have been met and the language has been parsed successfully, a model which is internal to the tool will be filled. This model will contain all information which was provided by the users through their object definitions.

There are reasons for translating the information from the description files into another model:

- The front end and the different back ends, may be better decoupled.

- The access to the model will be faster if the information is provided with the means of the implementation language rather than parsing the object descriptions over and over again.

A class diagram of the internal model may be found in Figure 4.2.

4.4 Back Ends

After the object descriptions have passed the step of the front end and the in memory model of the objects descriptions has been filled, this model may serve different back ends which will produce the desired output.

There will not be any need for developers to alter files which were generated by the back ends. In fact automatically generated files will contain some information that suggests to change the source description file and regenerate the output instead of changing the automatically generated file itself. This will also be the only way how to keep the information coherent on different platforms and compilers.

Each of the following sections will contain a description of a possible back end to the system. Furthermore an implementation of a sample object will be shown to illustrate the capabilities of the language. This sample object has the following characteristics:

- Single inheritance from a class "KeyedObject"

- Two attributes ("momentum" and "helicity") with the appropriate set and get functions

- One relation "originVertex" with a set and a get function

- Serialising and deserialising functions

- A function which dumps the content of the object to a stream

Figure 4.2: The internal model
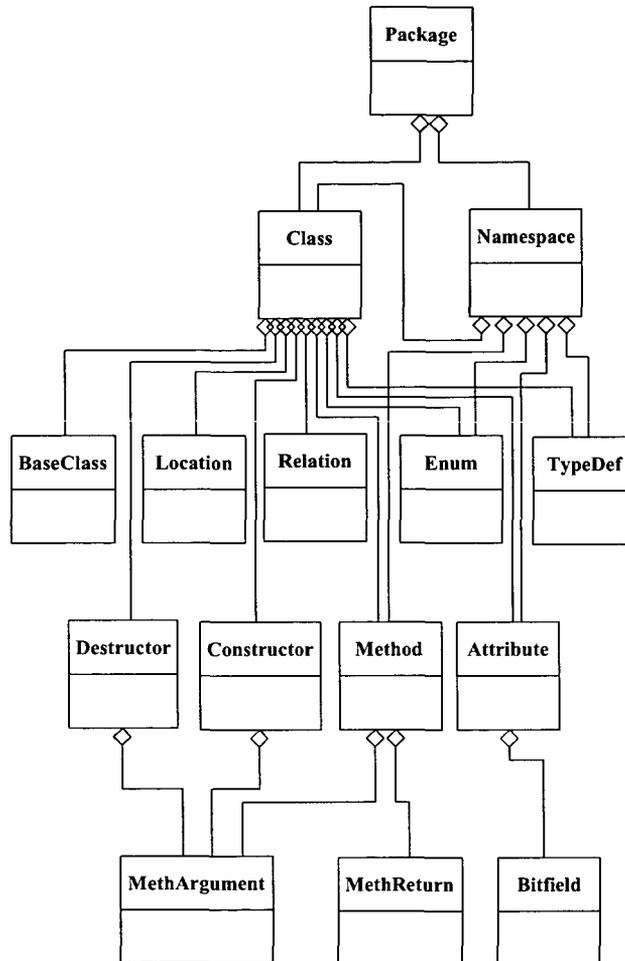
4.4.1   C++

As C++ is the most widely used implementation language in high energy physics for large software projects at the moment, it is also the most important back end to implement. This back end will produce a representation of the objects in C++. Implementations of objects in C++ are normally divided into two parts:

- Header files which contain the declaration of the classes that will be exported to the outside world. Those files contain the description of the objects but in most cases no implementations. (Only short functions, typically in-lined may be implemented in header files)

- Implementation files which contain the source code for member functions

which were not implemented in the header file.

In most cases no implementation files for event objects will be needed because all the functions needed are short and so may be generated inline in the header file itself. For this reason the aim of this back end is to only produce the object descriptions in header files but not in implementation files.

While producing the header-files this back end should be capable of performing the following tasks:

- Produce class declarations

- Generate accessor methods for attributes and relations

- Include C++ header files automatically when appropriate

- Generate documentation with the implementation code

- Obey the LHCb coding conventions for C++

The capabilities of the C++ back end are limited in the sense of C++, as the full functionality of C++ was not needed for representing the event model in this language. The main functionality needed was to represent attributes of objects and relations between these objects. The attributes are translated into class members of C++ classes. The relations are handled by an internal mechanism of the LHCb software which resembles C++ smart pointers [39].

The back end will also produce implementations of simple functions like accessors to members or serialising and streaming functions automatically. If there is a method missing in a class the users may either only put the declaration of the method into the class or also implement it with the means of the definition language if they want to. This can be useful in case of 'inline' functions. In the case of only declaring the function the actual implementation of the method will be left to the user in an extra implementation file.

A compilation unit in C++ is never self contained, it will always depend on other pieces of software. The event model will be split into different parts which are developed by different people. These parts will depend on each other and the information about other pieces of the event model will be included by the back end automatically whenever this is needed. For this reason the back end will contain a database of all types which should be known for the compilation of the event model. While parsing the definition language the tool will look for types of e.g. members of a class or argument and return types of functions and include the proper types into the header file when generating it.

If the automatic inclusion works this will facilitate the life of object designers because they do not have to bother where the right types are located and how to include them properly. If the automatic inclusion of types fails for some reason, the syntax of the description language provides means to force the inclusion of the proper type which is missing.

In LHCb there exist some coding guidelines for the implementation of C++ code. The goal of this back end is not only to reflect the structure of the objects but also meet these guidelines.

In addition to the source code also some documentation for the classes and objects will be generated. In a subsequent step it is though possible to extract this information and generate some general description about the event objects from it.

An example for the output of this back end may be found in Listing 4.1. This example only contains two members and one relation. The length of the listing should also document the redundancy of C++ code.

Listing 4.1: Example back end - C++ description

```
class MCParticle : public KeyedObject<int> {

public:

    MCParticle() : m_momentum(0.0, 0.0, 0.0, 0.0), m_helicity(0) {}
    virtual ~MCParticle() {}

    const HepLorentzVector& momentum() const;
    HepLorentzVector& momentum();
    void setMomentum(const HepLorentzVector& value);

    double helicity() const;
    void setHelicity(double value);

    const MCVertex* originVertex() const;
    MCVertex* originVertex();
    void setOriginVertex(const SmartRef<MCVertex>& value);

    virtual StreamBuffer& serialize(StreamBuffer& s) const;
    virtual StreamBuffer& serialize(StreamBuffer& s);
    virtual std::ostream& fillStream(std::ostream& s) const;

private:

    HepLorentzVector    m_momentum;
    double              m_helicity;
    SmartRef<MCVertex> m_originVertex;
};

inline const HepLorentzVector& MCParticle::momentum() const {
    return m_momentum;
}

inline HepLorentzVector& MCParticle::momentum() {
    return m_momentum;
}

inline void MCParticle::setMomentum(const HepLorentzVector& value) {
    m_momentum = value;
}

inline double MCParticle::helicity() const {
    return m_helicity;
}

inline void MCParticle::setHelicity(double value) {
    m_helicity = value;
}

inline const MCVertex* MCParticle::originVertex() const {
    return m_originVertex;
}

inline MCVertex* MCParticle::originVertex() {
    return m_originVertex;
}

inline void MCParticle::setOriginVertex(const SmartRef<MCVertex>& value) {
    m_originVertex = value;
}

inline StreamBuffer& MCParticle::serialize(StreamBuffer& s) const  {
    KeyedObject<int>::serialize(s);
    s << m_momentum
      << (float)m_helicity
      << m_originVertex(this);
    return s;
}

inline StreamBuffer& MCParticle::serialize(StreamBuffer& s) {
```

```
float l_helicity;
KeyedObject<int>::serialize(s);
s >> m_momentum
   >> l_helicity
   >> m_originVertex(this);
m_helicity = l_helicity;
return s;
}

inline std::ostream& MCParticle::fillStream(std::ostream& s) const {
s << "{ "
  << " momentum:\t" << m_momentum << std::endl
  << "  helicity:\t" << (float)m_helicity << " } ";
return s;
}
```

4.4.2 C++ Reflection

Today many modern compiled object oriented and scripting languages (e.g. Java, Python) provide reflection capabilities on their objects. Reflection is the ability to query the internal structure of objects at runtime and also interact with them (i.e. set or get values of members or call functions). Reflection is essential for generic tasks like persistence of objects or interactive usage, for example when working with objects from a terminal prompt through a scripting language. The reflective information capabilities of the C++ language (RTTI) are limited and not suitable for the tasks mentioned above. For this reason a software package was especially designed to support reflection in C++ (see Appendix B).

This back end will produce appropriate source files which contain all the information to fill the reflection structure. These descriptions are C++ classes which contain the needed meta information about the objects which will be compiled in a later step into libraries and then can be loaded by the reflection software to provide the meta information about these objects.

The goal of this back end is to provide reflection information on:

- All namespaces which are used in the model reflecting their names and short descriptions.

- All classes of the event model containing their names and some short documentation.

- The inheritance tree between classes. This also contains the relative memory offset of one class to another one.

- All fields of a class. This information contains the name, the type, a description, the memory offset of the field relative to the beginning of the class and the modifiers of the field.

- All methods containing information like return type, name of the method, argument types and a description.

With the information produced by this back end it will be possible to traverse the event model on a meta level, i.e. without knowing the concrete implementation it will be possible to query from an object instance what kind of class it is what are the members of this class and its methods etc. It will also be possible to get and set values of members and to invoke member functions and retrieve their return values.

Because the C++ header files and their reflection information will always be
produced from the same source file it will also be guaranteed that the informa-
tion is coherent and up to date. Also for this reason the developers of the event
model are asked to not alter either the header nor the reflection files which are
generated by the two back ends because otherwise incoherences between the
representation of an object and its meta representation could occur.

An output of this back end for the example object may be found in Listing
4.2.

Listing 4.2: Example back end - C++ reflection

```
static void* MCParticle_momentum_3 (void* v)
{
  const HepLorentzVector& ret = ((MCParticle*)v)->momentum();
  return (void*)&ret;
}

static void MCParticle_setMomentum_4 (void* v, std::vector<void*> argList)
{
  ((MCParticle*)v)->setMomentum(*(HepLorentzVector*)argList[0]);
}

static void* MCParticle_helicity_9 (void* v)
{
  static double ret;
  ret = ((MCParticle*)v)->helicity();
  return (void*)&ret;
}

static void MCParticle_setHelicity_10 (void* v, std::vector<void*> argList)
{
  ((MCParticle*)v)->setHelicity(*(double*)argList[0]);
}

static void* MCParticle_originVertex_11 (void* v)
{
  MCVertex* ret = ((MCParticle*)v)->originVertex();
  return (void*)ret;
}

static void MCParticle_setOriginVertex_12 (void* v, std::vector<void*> argList)
{
  ((MCParticle*)v)->setOriginVertex((MCVertex*)argList[0]);
}

static void* MCParticle_constructor_1 ()
{
  static MCParticle* ret = new MCParticle();
  return ret;
}

class MCParticle_dict
{
public:
  MCParticle_dict();
};

static MCParticle_dict instance;

MCParticle_dict::MCParticle_dict()
{
  std::vector<std::string> argTypes = std::vector<std::string>();
  MetaClass* metaC = new MetaClass("MCParticle",
                     "The Monte Carlo particle kinematics information",
                     0);

  MCParticle* cl = new MCParticle();
  metaC->addSuperClass("KeyedObject<int>",
                     (((int)cl)-((int)((KeyedObject<int>*)cl))));
  delete cl;

  metaC->addConstructor("default constructor",
                     MCParticle_constructor_1);

  metaC->addField("momentum",
                     "HepLorentzVector",
                     "4-momentum-vector",
                     OffsetOf(MCParticle, m_momentum),
                     MetaModifier::setPrivate());

  metaC->addField("helicity",
                     "double",
                     "Helicity",
                     OffsetOf(MCParticle, m_helicity),
                     MetaModifier::setPrivate());

  metaC->addField("originVertex",
```

```
                       "SmartRef<MCVertex>",
                       "Pointer to origin vertex",
                       OffsetOf(MCParticle, m_originVertex),
                       MetaModifier::setPrivate());

        metaC->addMethod("momentum",
                         "4-momentum-vector",
                         "HepLorentzVector",
                         MCParticle_momentum_3);

        argTypes.clear();
        argTypes.push_back("HepLorentzVector");
        metaC->addMethod("setMomentum",
                         "4-momentum-vector",
                         argTypes,
                         MCParticle_setMomentum_4);

        metaC->addMethod("helicity",
                         "Helicity",
                         "double",
                         MCParticle_helicity_9);

        argTypes.clear();
        argTypes.push_back("double");
        metaC->addMethod("setHelicity",
                         "Helicity",
                         argTypes,
                         MCParticle_setHelicity_10);

        metaC->addMethod("originVertex",
                         "Pointer to origin vertex",
                         "MCVertex",
                         MCParticle_originVertex_11);

        argTypes.clear();
        argTypes.push_back("MCVertex");
        metaC->addMethod("setOriginVertex",
                         "Pointer to origin vertex",
                         argTypes,
                         MCParticle_setOriginVertex_12);

        MetaPropertyList* pl = new MetaPropertyList();
        pl->setProperty("Author", "Gloria Corti");
        pl->setProperty("ClassID", "210");
        metaC->setPropertyList(pl);
}
```

4.4.3 Java

Java is another object oriented implementation language. At the moment it is not used in the LHCb framework software, so there is no urgent need for generating a back end for this language at the moment. This back end should generate the Java representations of the event objects.

A problem with the definitions in Java is, that there is only one file allowed for the definition and declaration of an object. This file will contain all the code, including the function bodies which belong to a given class. In most cases the back end will produce functions automatically (e.g. set- and get-methods), but the language also allows users to provide self defined functions. If they want to do this they have either the choice to only declare the function, which will only generate the signature of the function or to define it. When defining a function also the implementation code of the function has to be provided.

If the function is only declared but no function body is provided it may be difficult for the tool to generate the functions correctly. In case of Java where there is only one file allowed per object definition, several solutions to this problem are possible:

- It could be a requirement for the generation of the Java output that the self defined functions always have also to contain the function body, so the tool may produce the full output and no user changes are necessary.

- The tool could do some post processing, concatenating the user provided hand written functions with the ones which were created automatically.

- The problem could also be solved programatically when the user provides his handwritten functions in a class which inherits from the one which is generated by the tools and so extending its functionality.

As Java already inherently provides reflection information at compile time there is no need to generate this information explicitly.

An implementation of the example event object in Java may be found in Listing 4.3.

Listing 4.3: Example backed - Java description

```
public class MCParticle extends KeyedObject
                        implements Serializable {

    private HepLorentzVector    m_momentum;
    private double              m_helicity;
    private MCVertex            m_originVertex;

    public void init() {
        m_momentum = new HepLorentzVector(0.0,0.0,0.0,0.0);
        m_helicity = 0;
    }

    public final HepLorentzVector momentumF() {
        return m_momentum;
    }

    public HepLorentzVector momentum() {
        return m_momentum;
    }

    public void setMomentum(final HepLorentzVector value) {
        m_momentum = value;
    }

    public double helicity() {
        return m_helicity;
    }

    public void setHelicity(final double value) {
        m_helicity = value;
    }

    public final MCVertex originVertexF() {
        return m_originVertex;
    }

    public MCVertex originVertex() {
        return m_originVertex;
    }

    public void setOriginVertex(final MCVertex value) {
        m_originVertex = value;
    }

    private void writeObject(java.io.ObjectOutputStream out)
        throws IOException {
        out.writeObject(m_momentum);
        out.writeDouble(m_helicity);
        out.writeObject(m_originVertex);
    }

    private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException {
        m_momentum     = (HepLorentzVector) in.readObject();
        m_helicity     = (double)           in.readDouble();
        m_originVertex = (MCVertex)          in.readObject();
    }

    private void fillStream(java.io.OutputStreamWriter s)
        throws IOException {
        s.write("{ ");
        s.write(" momentum:");
        s.write(m_momentum.toString());
        s.write("   helicity:");
        s.write(java.lang.Double.toString(m_helicity));
        s.write(" }");
    }
}
```

4.4.4 C#

C# may be a successor of C++ in the future. C# is similar to C++ and also the generation of the event model in this language will be similar. An advantage of C# is, that it also already contains the means to generate the reflection information about its object when compiling the source code.

Although C# was originally written only for the windows platform, there are already compilers developed also for other platforms. (The sample code in this section was checked and compiled with *cscc* [22] the C# compiler for linux developed by the free software foundation)

Two important tasks when generating the source code for the event objects will be facilitated when using C#:

- The capability for serialising objects will be provided inherently by the programming language together with the reflection information about the object. A keyword [Serializable] will provide a hint to the compiler that the object can be made persistent. (The keyword [NonSerialized] may be used in conjunction with certain object members to indicate that they should not be written out)

- The explicit set- and get-functions for attributes and relations will be replaced by the concept of properties. Properties will encapsulate the object member inside the class and provide the capabilities of retrieving and/or setting its value.

An example for an implementation of the example event object in C# may be found in Listing 4.4.

Listing 4.4: Example back end - C# description

```
[Serializable]
public class MCParticle : KeyedObject
{
    private HepLorentzVector  m_momentum;
    private double            m_helicity;
    private MCVertex          m_originVertex;

    public MCParticle()
    {
        this.m_momentum = new HepLorentzVector(0.0, 0.0, 0.0, 0.0);
        this.m_helicity = 0;
    }

    ~MCParticle() {}

    public HepLorentzVector Momentum
    {
        get
        {
            return (m_momentum);
        }
        set
        {
            m_momentum = value;
        }
    }

    public double Helicity
    {
        get
        {
            return (m_helicity);
        }
        set
        {
            m_helicity = value;
        }
    }
```

```
public MCVertex OriginVertex
{
  get
  {
    return (m_originVertex);
  }
  set
  {
    m_originVertex = value;
  }
}

virtual public void fillStream(ref MemoryStream m)
{
  StreamWriter s = new StreamWriter(m);
  s.WriteLine("{");
  s.WriteLine("momentum: {0}", this.m_momentum);
  s.WriteLine("helicity: {0}", this.m_helicity);
  s.WriteLine("}");
  s.Close();
}
}
```

4.4.5 Python

One may also imagine to generate the information not only in object oriented compiled languages but also in scripting languages like Python [43] as it is a strong component in the Gaudi framework.

In the case of interactive scripting it could make sense to generate the information of the event model also in Python and then access the model directly instead of indirect access through a python binding to the main implementation language of the software framework. The implementation of the sample event object in Python may be found in Listing 4.5.

Listing 4.5: Example back end - Python description

```
class MCParticle(KeyedObject) :

  def __init__(self):
    m_momentum = (0.0,0.0,0.0,0.0)
    m_helicity = 0
    m_originVertex = 0


  def momentum(self):
    return self.m_momentum

  def setMomentum(self, value):
    self.m_momentum = value


  def helicity(self):
    return self.m_helicity

  def setHelicity(self, value):
    self.m_helicity = value


  def originVertex(self):
    return m_originVertex

  def setOriginVertex(self, value):
    self.m_originVertex = value


  def serializeOut(self, s):
    s += '|' + self.m_momentum
    + '|' + self.m_helicity
    + '|' + m_originVertex.persInfo

  def seralizeIn(self, s):
    slist = string.split(s,'|')
    self.m_momentum = slist[0]
    self.m_helicity = slist[1]
    self.m_originVertex.persInfo = slist[2]

  def fillStream(self, s):
    s += '{ ' + 'momentum:\t' + self.m_momentum + '\n'
    + ' helicity:\t' + self.m_helicity + ' }'
    return s
```

4.4.6   Other Back Ends

The data description language itself is probably not designed for browsing and displaying it in a human readable form. For this reason it will be necessary to also process this information so that it can be viewed with some standard tools like HTML browsers or text editors. This back end should perform the following tasks:

- Show the information in several "information levels". Depending on the level of information the users wants to see, the back end should be capable of displaying either the full information about an object or only parts of it. It could be useful e.g. to only display names and types of attributes when displaying the object for general use while for debugging it could be also useful to display informations like modifiers of the attribute or the initialisation value when the object gets instantiated.

  Another use case could be, that the information only for a specific back end will be displayed. This could be useful in case of self defined methods which can also contain some implementation code which will be long if the implementation for all different back end languages will be shown at once.

- This back end should also provide means to traverse the event model. As the different objects of the model are related to each other, an online version could be capable to follow links between the different pieces and so to walk through the model.

4.5   Related Parts

There are some parts which are not directly part of the system but which also have to be taken into account and are strongly related to the overall design. The most important of these tasks are persistence and interactivity.

4.5.1   Object Persistence

As already mentioned, the data volume that will be produced with the detector will be huge and a big amount of this data produced will be needed to store onto persistent media. The task of making the data, obtained from the detector and processed by different pieces of software, persistent is a non trivial one. Due to the high volume of data it has to be optimised both for speed and compression of data. The design of the event model may support these tasks.

For the first task of speed it will be necessary that the information contained in the model may be accessed quickly. For this reason the get and set functions have to work as quick as possible. For instance in C++ this may be achieved by inlining the function and saving a stack call when processing the data. In some cases special containers will be needed to store collections of small events. Although these containers also have to serve other tasks, the access time to these containers will also be crucial.

The compression of the data is also an important aspect. The tools for persistence of the data will try to compress the data as much as possible. If the data is already compressed when read by these tools it will be of course better also for the speed of the processing of the data. As already mentioned before there are many small objects in the model and collections of those. The compression of those objects will be either done inside special containers or if the information is only bitwise the compression will be done through bit fields.

4.5.2 Interactivity

For the task of interactivity speed and compression are not so important. It will be more important to access the data through some interfaces which provide translations to scripting languages. These converters also need to know the content of the event model. In order to do this, introspection information of the data contained is necessary. In the cases where reflection information is not provided intrinsically by the implementation language it has to be maintained externally. This means that the meta information about objects needs to be generated by some tools that process the event model.

Chapter 5

Technical Choices

After defining the logical structure of the model, the decision about the usage of the concrete languages have to be taken. The first part of this chapter will discuss the possible choices for the language which can be used for the description of the event objects. Each section containing a proposal for a language will start with a short introduction into the language, followed by a description of its advantages and shortcomings. Each section will be concluded by a table listing advantages and disadvantages.

In the second part the possible implementation languages for the tools and parsers for the language will be discussed in more depth. Each of the sections also contains a description of the language, a discussion and a table summarising its advantages and disadvantages.

Both parts will be concluded by a section which compares the different approaches and chooses one of them.

5.1 Data Description Language

The most important decision that had to be taken was the one about the data model description language itself which will be used for describing the event data. Several choices were possible.

5.1.1 Homegrown Language

When using a homegrown language, first the syntax for this language has to be created. It has to be guaranteed that the syntax of this language is complete and enough for the description of the event objects. The development of tools for parsing and handling the language will be completely left to the developers.

In previous experiments at CERN such as ALEPH this approach was used. A language, called DDL, and a set of tools for its handling (ADAMO) were implemented (see section 2.4.1).

The advantage of this approach would be the easiness of creation of such files, as no special environment, such as editors or compilers, would be needed. Every text editor will be enough to create such an object description.

41

A drawback to this freedom is, that creators of such files will not have immediate feedback whether they built syntactically correct files until the data is read with the tool.

Homegrown language also means freedom for the designers of the language because there are no constraints whatsoever. They may design a language syntax which is especially tied down to the specific needs of event models of HEP experiments or even to the constraints of one specific experiment. On the other hand no constraints also means that every tool for parsing and processing the event descriptions will have to be written from scratch.

Table 5.1: Advantages/Disadvantages Homegrown Languages

⊕	object descriptions are easy to produce	⊖	parsers have to be written explicitly
⊕	no extra tools required for production	⊖	maintenance of the tools completely in the hands of the developers
		⊖	display of the information is only possible with special tools or reading sources directly
		⊖	syntax of the language has to be defined

5.1.2   C++

C++ has been one of the most important languages in object oriented programming for several years. It is a powerful language with a complex syntax which is standardised since 1998 [52].

An advantage of C++ is its strong syntax which can be checked easily with many different compilers. Although the language has been standardised for some time, no compiler completely implements it, hence the checking of the object descriptions with different compilers could result in different responses.

With the C++ syntax all requirements for the description of event objects will be met. On the other hand the syntax of C++ is too powerful for the description of the event objects and there are no means to constrain the language to a certain subset. Designers of event objects may be tempted to use concepts of C++ which will bind the description too much on the special features of C++. If this happens, object descriptions might not be portable to other languages anymore.

C++ is also difficult to learn. At the moment it is popular as a programming language in the HEP community and people will not have problems to use it for object descriptions. Nevertheless it may phase out at some time and then designers will be forced to learn a difficult language for the only purpose of describing a simple event object model.

As C++ is a complex language and difficult to maintain it should be the goal of the tools processing the data to produce output in this language rather than urging the users to provide descriptions with it.

Table 5.2: Advantages/Disadvantages C++

⊕	strong syntax	⊖	difficult syntax
⊕	currently popular language	⊖	redundant syntax
		⊖	should be target not definition language
		⊖	no means for constraining

5.1.3 IDL

IDL [38] is the interface definition language of the Common Object Request Broker Architecture (CORBA) of the Object Management Group (OMG). CORBA aims to provide inter-operability between different programming languages and platforms. It tries to simplify heterogeneous distributed computing systems and to be independent of the language and platform where software is produced.

IDL is a language which has been developed for several years. The CORBA project provides already bindings to many different implementation languages which proves that the language per se is complete. IDL is exactly what is needed for the task of describing interfaces in an language independent way, but there are some drawbacks to this solution.

The major drawback of IDL is, that it is only suitable for defining interfaces. Interfaces by definition, do not contain data. It is also incompatible with the requirement that in some special cases the developers of the event object may also provide some implementation code for self-defined functions. Extending the language in such a way that it will also accept pieces of implementation code will be difficult.

As IDL is designed for the description of interfaces it will also be difficult to integrate concepts like attributes or relations to other objects. IDL also does not care how data is structured internally.

Other experiments at CERN have tried to produce language independent descriptions for event objects using IDL with some extensions [3, 4].

Table 5.3: Advantages/Disadvantages IDL

⊕	well known language	⊖	difficult to extend
⊕	independent of platform and language	⊖	restricted to interface definition

5.1.4   UML

UML [20, 49] is a widely used graphical language to describe object models. It has a strict syntax and is often used to describe large software projects. Out of the many possibilities to describe actions and situations in software using UML the class diagrams and package diagrams should be used. While the class diagrams describe objects, their ingredients and the relations between them, the package diagrams will describe relations between packages.

The way how UML is used, is to generate graphical representations of objects, their content and their relations with special editors. These editors also provide language bindings with which the implementation code for objects may be generated in an automatic way.

If a language binding is needed but not available by the chosen tool which in turn is needed as an implementation language in the LHCb collaboration one has to wait until the new binding will be developed.

UML is standardised only on its graphical representations, there is no standard for a written language which describes UML diagrams. Every tool may generate a written description of the diagrams but they will be different. This means that the development of the event model will be bound to one single tool which will allow the exchange of the object information.

The best tool for modelling with UML, Rational Rose [6, 45, 46], is not a freeware tool and buying licenses for it quite expensive, which may not be feasible for all institutes collaborating in LHCb.

Running the tools on different platforms is also sometimes difficult. For example the native platform for Rational Rose is windows. While it provides also the possibility to run on Linux the operability on this platform is restricted and execution time much slower. Freeware tools for handling UML are mostly implemented for one platform only.

Table 5.4: Advantages/Disadvantages UML

⊕	widely used language	⊖	difficult to extend
⊕	strict syntax	⊖	most modelling tools are not free-ware
		⊖	only limited choice of language bindings

5.1.5 XML

The Extensible Markup Language (XML) [54] is a simple and flexible text format, originally designed to meet the challenges of large scale electronic data publishing. Nowadays XML is also used in many other environments for data definition and exchange. XML is also a wide spread and well known language in the computing environment for which several tools such as browsers, editors, parsers and language bindings exist.

The syntax of an XML document can either be described in so called DTDs or XML Schemas. While DTD provides a limited functionality, XML Schema is a complex language which gives the developer of the syntax a lot of means to go to a detailed level of description.

XML consists of two main entities, namely elements and attributes. Elements define the objects of the language while attributes are always parts of elements and specify their behaviour. As the developer of the language is also the creator of its syntax, extending it with new features is simple and straightforward.

The advantage of this will be, that the definition of the language is completely free and in the hands of the developer. It will though be possible to start with a small set of language constraints and extend them whenever needed. Of course the surrounding environment has also to support such a flexible way of dealing with new features which have to be introduced into the language.

A drawback of XML is, that it is a verbose language and that for some special characters which are part of the syntax escape sequences have to be used if they are part of the data (e.g. "<" for "<"). To overcome the disadvantage of verbosity default values in the syntax can be used which will not have to be typed in most cases by the developers of the event objects.

5.1.6 Comparison

There are many choices of description languages. There are already some projects and frameworks which aim to provide object descriptions on a language and/or platform independent level (e.g. IDL, UML). Although these languages are adaptable they will not provide the flexibility that is needed by the language for the object descriptions. Although a syntax which is suitable for describing

Table 5.5: Advantages/Disadvantages XML

⊕	flexible language	⊖	verbose language
⊕	easy definition of syntax available (DTD)	⊖	escape sequences
⊕	sophisticated syntax definition possible (XML Schema)		
⊕	widely used		
⊕	use of default values possible		
⊕	many parsers available		

most of the cases may be developed in a short time and will be stable for some more time, the expectation was, that due to the long lifetime of the experiment, implementation languages and constraints will evolve which will change the requirements to the description language. In that respect a description language which may adapt to a changing environment will be the best choice.

C++ and UML are also very powerful languages which provide much more functionality than is actually needed for the task of describing event objects. There are no means in these languages to restrict the syntax and as it should be tried to keep the language for the object description as simple as possible the power of these two languages is considered a disadvantage in this case.

Also the fact that the development and evolution of the language and tools should be fully in the hands of the experiment developers was a point that was important. For example in case of UML this is not completely the case and with evolution of the languages or upcoming new languages one has to wait for the providers of the tools and languages to provide the proper bindings and facilities.

In case of XML the whole development will be in the hands of the developers at CERN which will be more work but also provides the flexibility to change the syntax on the short term. Also in case of text files the development would be in charge of the local developers, but in contrast to XML several general tools like parsers or editors would have to be written from scratch which are already available in a wide variety for XML for different platforms and implementation languages.

Because of its ability of easy extension and its strict syntax XML with a specific DTD was chosen as the language for the description of the objects.

XML was also chosen because it was already used in LHCb, e.g. for the detector description. So it was hoped that people are already used to working

with this language and it will not take a lot of time for them to get up to speed with it.

5.2 Implementation Language

It was also necessary to decide on the implementation language for the tools which would be used for the parsing of the description language and the different back ends of the system. In general any language that would be capable of parsing a given description language and producing some output would be sufficient for this choice. Possible choices in this case were compiled object oriented languages such as C++ or Java as well as scripting languages like Python or Perl.

Another constraint that was set up by the Gaudi environment was that the language chosen had to work on two different platforms, namely windows and linux, which is done to ensure the platform/compiler independence of the source code.

5.2.1 Scripting Languages

It would be possible to implement the tools needed with scripting languages like Perl [41] or Python [43]. Scripting languages in general have the advantage to allow faster prototyping of code. Python especially has also the advantage that it supports object oriented programming features, which will enhance the usability and readability of the code. For big software projects scripting languages will not be optimal because, as they are only interpreted, they will run slower than compiled languages which can be optimised to the machine hardware. Although there are some features in scripting languages to produce so-called byte-code which resembles compiled code closely.

Both Perl and Python are well known and well supported languages, for which there exist also a lot of external tools. For Python e.g. there are several XML parsers which understand both DTDs and XML Schemas. So an implementation with a scripting language should also be possible.

Table 5.6: Advantages/Disadvantages Scripting Languages

⊕	fast prototyping possible	⊖	slower execution of code
⊕	good documentation available		
⊕	many modules and extensions available		
⊕	Python and Perl widely used		

5.2.2   Compiled Languages

Compiled languages have the advantage to be faster but on the other hand the development phase will take more time because of recompilation of code. C++ [52] and Java [31] are two compiled languages which also support features of modern object oriented programming. Although Fortran was the dominant language in high energy physics until recently, it was not taken into account, because it does not support object oriented programming and is phasing out for software in high energy physics.

It has also to be chosen, with which tools one would like to parse the object description code (XML). For C++ and Java the dominant XML parser at the moment is called Xerces [11], which was developed as open source software in the apache project.

Table 5.7: Advantages/Disadvantages Compiled Languages

⊕	fast execution of code	⊖	Java proprietary language
⊕	many modules and extensions available	⊖	design and implementation will take longer
⊕	C++ and Java rather up to date languages	⊖	recompilation after every code change
⊕	C++ and Java support object oriented programming		
⊕	Good documentation available		
⊕	C++ standardised by ISO		

5.2.3   Comparison

Although the tool itself is completely independent of Gaudi, C++ was chosen for the implementation language. The reasons for choosing C++ were, that execution time of the tools will be faster which may play a role when parsing several hundreds of definition files. Another reason was, that C++ is already used in the framework as the main implementation language which will also speed up the maintenance and readability by other developers, who are already used to the language.

As a tool for parsing the description language, Xerces-C [11] was chosen as there existed a C++ implementation of this parser and it was also already used in Gaudi for the detector description part. Xerces has been developed for several years and is already in a stable phase. It is also able to verify XML documents either with DTD or XML Schema.

Chapter 6

Implementation

In this section the concrete implementation that was used in LHCb will be explained in more detail. The first section will give a short introduction into XML, followed by the description language which was developed. The next section will give an explanation of the rules which were applied when running the different tools. The chapter will be concluded by an explanation how the tools were integrated into the Gaudi software framework and used to automatically produce the source code.

6.1 Introduction into XML

XML [54] is a simple and flexible text format which was derived from the SGML standard [51]. XML is standardised by the world wide web consortium[1] (W3C). The current recommendation of the W3C for XML is available at [8].

The main parts of an XML document are its elements and attributes. Elements describe the objects of the document and may be nested into each other. Attributes are always attached to an element and describe its behaviour. In the example in Listing 6.1 the elements are <package>, <class> and <attribute>, the attributes are *name, author, id, serializers, type* and *init*.

Listing 6.1: Example XML document

```
<package name=''Event''>
  <class name=''MCParticle'' author=''Gloria Corti'' id=''210''
        serializers=''FALSE''>
    <attribute name=''particleID'' type=''ParticleID'' init=''0''/>
  </class>
</package>
```

XML documents may be either well-formed or valid. Well-formed is the weaker statement for checking the correctness of an XML document. A well-formed document follows the rules of the XML specification that for example the elements are correctly nested. Valid documents also contain a syntax description to which they have to adhere.

---

[1]http://www.w3c.org

The syntax of the LHCb object description language is currently described with a DTD. The two main elements for DTD descriptions are `<!ELEMENT>` and `<!ATTLIST>` which describe the elements and their attributes.

In a DTD the possible sub elements are given in the parenthesis behind the element name. The cardinality of each sub-element is specified by a character after the element name as follows:

- ?: 0 or 1 occurrence of the element

- *: 0 or more occurrences of the element

- +: 1 or more occurrences of the element

Each element has a list of attributes attached. Each line of the listing contains the attribute name, the possible values of the attribute and a possible keyword to specify whether the attribute is required or not. Possible values for attributes are:

- `CDATA` specifies that the attribute may contain any text.

- It is also possible to specify a pool of possible values inside parenthesis. If this is done, only one of the specified values may be used as a value for the attribute. After the parenthesis a default value inside quotes may be given.

Keywords to specify whether an attribute is required or not, are:

- `#REQUIRED` means that a value for this attribute has to be provided by the developer of the language. For some attributes it is obvious that they are required by the language, e.g. name of a class. Other attributes are required due to conventions in the LHCb collaboration (e.g. descriptions).

- `#IMPLIED` means that a value for the attribute may be provided. If no value will be provided the description of the object will be still correct.

- `#FIXED` was used to attach a given attribute to an element. Users of the language are not allowed to change the values for these attributes.

A part of the DTD describing the example XML document in Listing 6.1 can be found in Listing 6.2.

Listing 6.2: Example DTD

```
<!ELEMENT package ((import*,class*,namespace*)*)>
<!ATTLIST package
        name CDATA #REQUIRED
>

<!ELEMENT class (attribute*)>
<!ATTLIST class
        name CDATA #REQUIRED
        author CDATA #REQUIRED
        id CDATA #IMPLIED
        serializers (TRUE | FALSE) "TRUE"
>
```

6.2 Gaudi Object Description Language

As described in the previous chapter, XML was chosen as description language. The Gaudi Object Description language (GOD) consists of 19 elements which should allow developers to define their event objects with enough detail.

For the GOD language it was decided to start the description of the language syntax with a DTD and switch to XML Schema if the language reaches a level of complexity that DTD is not able to handle anymore. As the event object description only requires a simple syntax it is still possible to describe the whole language with a DTD (see Listing A.1).

6.2.1 Element Description

In this section the different elements of the GOD language (see Figure 6.1) will be explained in more detail. In mean each XML element of the GOD language has 4 attributes. As the description of all the attributes will be too much detail only the most important ones will be explained. The rest can be looked up in the DTD of the language in Listing A.1.

Figure 6.1: Elements in the Gaudi Object Description Language

- `<gdd>`
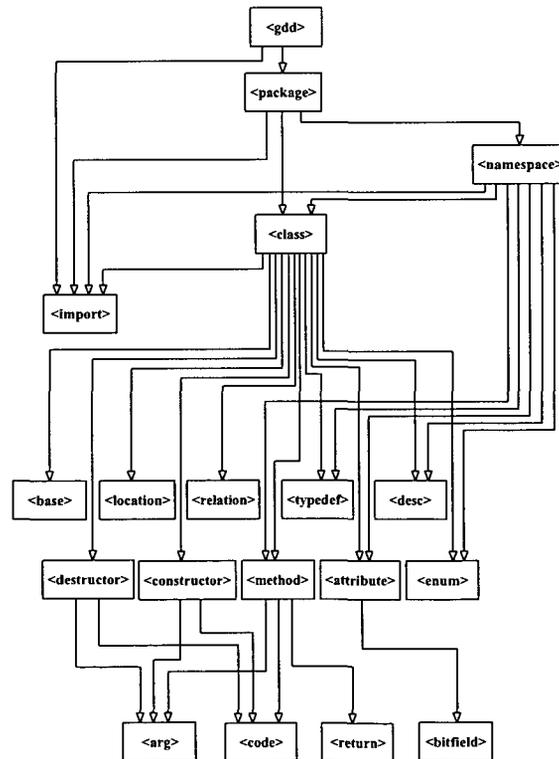
  - Sub elements: `<import>`, `<package>`
  - Attributes: *version*

  `<gdd>` is the top level element. Elements in XML are arranged in a tree and the top level element is the only root of the tree.

  The only attribute of this element contains the *version* number which could be used for versioning of the syntax of the XML document. So far there were no backward incompatible changes to the syntax definition of the event object language and so this version attribute was not used so far.

- `<import>`

  - Sub elements: none
  - Attributes: *name, std, soft, ignore*

  In general external units of source code which are required for the compilation will be included automatically whenever they are needed. If this fails for some reason, `<import>` will introduce the dependency to another source code unit of the software which this one depends on. The `<import>` element can be used in several levels of the syntax. It may be used as a sub-element of `<gdd>` which means that the import unit will appear in every package and subsequently every class. Or it may also only be sub-element of `<package>` or `<class>` and appear down the tree accordingly to these objects.

  Attributes of `<import>` define whether it should be treated as a unit that is provided by the language itself (*std*) or it should be treated as a declaration unit only which will declare an object but not import it, to avoid cyclic dependencies (*soft*).

- `<package>`

  - Sub elements: `<import>`, `<class>`, `<namespace>`
  - Attributes: *name*

  `<package>` may contain `<class>`, `<namespace>` and `<import>` elements. Together they build a set of objects and declarations which inherently belong together (e.g. for a given sub-detector).

- `<class>`

  - Sub elements: `<constructor>`, `<destructor>`, `<base>`, `<desc>`, `<location>`, `<attribute>`, `<relation>`, `<method>`, `<import>`, `<typedef>`, `<enum>`
  - Attributes: *name, author, desc, filename, id, location, stdVectorTypeDef, keyedContTypeDef, templateVector, templateList, serializers*

As `<class>` is the central object in the description framework. It resembles the objects that have to be described (e.g. a MCVertex or MCParticle). Sub elements of `<class>` contain all the information that a given object has, e.g. its members or its relations to other objects. As classes are the central piece of software in object oriented languages they also play a central role in the event description language. They support inheritance to other classes and will encapsulate their members in the different sections of the object. Most of the elements of the event description syntax are direct or indirect sub-elements of the `<class>` element.

`<class>` is an important element it also has a lot of attributes. A *name* and an *id* which will make it unique, an *author* and a *desc*ription used for documentation purposes, a *location* which will describe the location of the class in the transient event store.

- `<desc>`

  – Sub elements: `#PCDATA`

  – Attributes: *xml:space*

  `<desc>` contains a description of a class. Although `<class>` has an attribute for its description, `<desc>` may be used if a long and in depth description for a class is necessary.

- `<location>`

  – Sub elements: none

  – Attributes: *name, place, noQuote*

  Every event object needs to specify its location in the transient event store. The primary place of the object will be specified with the *location* attribute of `<class>`. If the object may appear in more than one place the `<location>` element may be used to define these places.

- `<namespace>`

  – Sub elements: `<desc>`, `<enum>`, `<class>`, `<method>`, `<import>`, `<attribute>`, `<typedef>`

  – Attributes: *name, author, desc*

  `<namespace>` is an element on the same level as `<class>`. Nevertheless it may contain `<class>` elements, furthermore other elements like `<typedef>` or `<enum>`. The difference between `<class>` and `<namespace>` is that the latter is only a container from which no instances can be derived. It only makes a given set of objects unique, so they will not clash with others.

  `<namespace>` also has several attributes like *author* and *description* for documentation purposes.

- `<base>`

  – Sub elements: none

– Attributes: *name, virtual, access*

<base> is a sub-element of <class>. <base> will be used to describe the inheritance between objects. In the Gaudi framework only single inheritance between real objects is allowed.

The attributes of <base> describe the *name* of the base object and the way the inheritance to this other object is done.

- <enum>

  – Sub elements: none

  – Attributes: *name, desc, value, access*

The <enum> element is a child element of both <class> and <namespace>. It consists of a set of variables which will be given some numbers. This setting of numbers can be either done automatically, starting at 0, or explicitly.

<enum> has a *value* attribute which will hold a comma separated list of variables which are the items of the enumeration. Each variable may be followed by an equal sign and a value giving a variable a value.

- <typedef>

  – Sub elements: none

  – Attributes: *desc, type, def, access*

The <typedef> element is a way of increasing readability in a program. Typedefs are aliases to other constructs in a program.

Besides the usual attribute for a *desc*ription the two main attributes of this element are *type* which holds the original type and *def* which stores the name with the alias for this type.

- <constructor>

  – Sub elements: <arg>, <code>

  – Attributes: *desc, argList, argInOut, initList*

In object oriented languages constructors are responsible for creating a new instance of an object. Constructors are able to assign values to the members of the new instance of the object and also may execute some further code which may be necessary at creation time of the instance. The allocation of memory will be done automatically when the constructor is called.

Besides the usual *desc*ription attribute, the constructor also accepts an *argList* attribute with which one may specify arguments in a simple way.

- <destructor>

  – Sub elements: <arg>, <code>

– Attributes: *desc, argList, argInOut*

The `<destructor>` element will destroy elements and free the memory that was allocated for them. Destructors are, as well as constructors, specific to object oriented programming. Although not all modern languages work with destructors (some also use garbage collection instead), it is a crucial part for languages which do use them.

`<destructor>` will accept a *desc* attribute for documentation and and *argList* attribute for simple description of arguments.

- `<method>`

  – Sub elements: `<arg>`, `<return>`, `<code>`

  – Attributes: *name, desc, template, access, const, virtual, static, inline, friend, type, argList, argInOut*

  Most of the methods that are needed for the event objects, e.g. set- and get-methods for attributes or relations will be generated automatically. Nevertheless it will also be necessary in some cases to define and declare methods which are not generated automatically. These methods may either be special set or get methods, e.g. treating the arguments in a special way, or do something special related to the object. Possible sub-elements of `<method>` are `<code>` which will take some language specific implementation code and `<arg>` which will describe an argument with more detail than the attribute *argList* could do. It is also possible to only declare a method and implement it in another place, in that case the `<code>` element will not be needed.

  The method element accepts several arguments. Besides the ones for documentation purposes, i.e. *name* and *desc*, it also accepts an *argList* which describes simple arguments of the method. A *return* attribute describes the return type of the method. The *access* attribute specifies in which section of the object the `<method>` resides (public, protected, private).

- `<arg>`

  – Sub elements: none

  – Attributes: *type, name, const, inout*

  The `<arg>` element is a sub-element of `<constructor>`, `<destructor>` and `<method>`. While all these three elements also have an attribute to declare simple arguments, sometimes it may be necessary to define arguments with more complex definitions. There are some special rules inside the system how arguments should be treated by default. The `<arg>` element in most cases has to be used when these rules have to be overwritten.

  Arguments of the `<arg>` element are *type* and *name* which denote the language type and the name of the argument.

- `<return>`

    - Sub elements: none

    - Attributes: *type, const*

The `<return>` element is a sub-element of `<method>`. It specifies the return value of the method. Its arguments specify the *type* and the *const*ness of the returned type.

- `<code>`

    - Sub elements: #PCDATA

    - Attributes: *lang, xml:space*

The `<code>` element is a sub-element of `<constructor>`, `<destructor>` and `<method>`. It will contain language specific implementation code of these functions if necessary. This is the only case were an introduction of language dependencies into the event model is needed. The only important attribute of the `<code>` element describes the *lang*uage that it uses.

- `<attribute>`

    - Sub elements: `<bitfield>`

    - Attributes: *type, name, desc, init, array, access, compression, serialize, setMeth, getMeth*

The `<attribute>` element is one important piece of content of an object. Attributes are the member variables of a class which will be stored within. Examples for attributes are the energy of a particle or its id. Types of attributes may be any simple but also complex type which is aggregated into the object.

The `<attribute>` element has several XML attributes. *type* and *name* for the basic description. An *init* attribute describes the initialisation value it should take when the object is constructed. The *setMeth* and *getMeth* denote whether the set- and get-method for this attribute should be constructed automatically or not. The default for these methods is, that they will be created by default, the users only have to set this attribute to false if they do not want these methods or want to overwrite them. The *access* attribute specifies whether the attribute should reside in the public, protected or private section of the object. The *serialize* attribute specifies whether the attribute should be written on a persistent medium when the object will be made persistent.

- `<relation>`

    - Sub elements: none

    - Attributes: *type, name, desc, access, multiplicity, serialize, setMeth, getMeth, addMeth, remMeth, clrMeth*

`<relation>` is another important part for describing an event object. Relations are connections to other objects of the event model. There will be the possibility of a 1 to 1 or a 1 to many relation. Both kinds of relations will be treated with special containers in LHCb.

The `<relation>` element has similar attributes like the `<attribute>`. Additionally there is an attribute *multiplicity*, which specifies 1 to 1 or 1 to many relations and some attributes which denote whether an add-, remove- or clear-method should be created in case of a 1 to many relation. The default for the accessor methods is true, which means that they will be created by default.

- `<bitfield>`

  - Sub elements: none
  - Attributes: *name, length, desc, startBit, setMeth, getMeth*

  `<bitfield>` is a special sub-element of `<attribute>`. A reason for using bit fields will be the compact storage of bitwise information. This set may be stored in such a bitfield. The length of the bitfield may also be specified explicitly for each implementation language.

  Attributes of `<bitfield>` are the usual *name* and *description*, a *length* attribute to describe how long the field is and a *setMeth* and *getMeth* attribute to specify whether this bitfield should have an explicit setter or getter method.

6.3 Rules

The rules that were implemented may be divided into two parts. First there are general rules which will apply to all back ends of the system. These rules include e.g. general coding conventions. Second there are specific rules which only apply to a given back end. In the LHCb software framework there are for example some specific rules which only apply to the C++ language. Specific rules that are implemented also imply the automatic behaviour of the tool for this back end, which is special dealing with elements and triggering the output on the existence of combinations of elements. The different levels of rules and their input can also be seen in Figure 4.1.

6.3.1 LHCb Rules

As these rules apply to every back end of the system they may be checked at an early stage of processing, e.g. while running the parser over the description language. The following list will give a short overview and description of some of these rules:

- Every class, method, attribute and relation has to have a description. For this reason the *desc* attribute is required for each of the elements in the description language.

- Classes have to be contained in a package.

- Each package should have a unique name and start with a capital letter

- Concatenated names should be built without underscores, each first letter being uppercase

- Member variables of classes should consist of "m_" and the variable name, starting with a lowercase letter.

- Static variable should start with a "s_" concatenated with the variable name.

- Names of get functions shall be the variable name starting with a lower letter, e.g.the get function for the attribute m_particleID will be

```
const ParticleID& particleID() const;
```

- Set functions shall be concatenated with "set" and the variable name starting with a uppercase letter, e.g.

```
void setParticleID(const ParticleID& value);
```

- The length of every line should not exceed 80 characters.

6.3.2   Specific Rules for C++

For the time being there exist only coding rules for C++ in the LHCb collaboration. There are some rules which were applied from the LHCb coding conventions for C++ [5, 12]:

- Every class shall have a function which returns a "ClassID". The ClassID is a unique number which is a shortcut to the current class. This will speed up the access to classes.

- Each class should have a header-file and an implementation-file. The default file name is the class name concatenated with '.h'

- Standard includes should look like

```
#include <vector>
```

  All other includes should look like

```
#include ''Pathto/Includefile.h''
```

- Each header file should contain one class. If a class has an inner class it may be part of the same header-file.

- Every header file has a "header guard" to avoid multiple inclusions, e.g.

```
#ifndef PACKAGE_NAME_H
#define PACKAGE_NAME_H 1

... class body ...
```

- Forward declarations should be used whenever possible to avoid cyclic dependencies

- A class should start with the public declarations, followed by the protected and private ones.

- For clarification typedefs are allowed. They should be placed at the beginning of the public section of the class.

- There shall be no implementations of functions inside the class body. Inline functions should be implemented after the class body in the header file, e.g.

```
class MCParticle
{
  double virtualMass() const;
};

inline double MCParticle::virtualMass() const
{
  return m_momentum.m();
}
```

- The constructor should initialise all variables and objects of a class at creation time.

- A virtual destructor is mandatory for every class.

- The const modifier should be used for functions which do not change the object (e.g. get-methods) or arguments and return values which will not change or are not allowed to be changed.

- In general objects should be passed by const reference. Small objects (e.g. basic types) can also be passed by value.

- When declaring functions, each argument should be put on a separate line.

- Comments should be compliant to the doxygen rules e.g.

```
/** @class MCParticle MCParticle.h Event/MCParticle.h
 *
 *  The Monte Carlo particle kinematics information
 *
 *  @author Gloria Corti
 *  @date    17/07/2003
 *
 */
```

- Every method should be prepended by a description in doxygen style e.g.

```
/// Pointer to parent particle
```

In addition to the official coding conventions some more rules were applied which were thought to be useful for the implementation of the event objects in C++:

- attributes, relations, enumerations and typedefs are put into the private section of the class by default.

- methods are created by default in the public section of the object.

- access methods (set, get, add, ...) for attributes, relations and bit fields will be created by default.

- If the <class> element has an attribute id, the class is treated as an event class for which the accessor methods for the class ID will be generated.

- For the <class> element either the attribute *desc* or the sub-element <desc> which contain the description of the class, have to exist. If both are provided they will be concatenated.

- If no <constructor> element with 0 arguments was provided, a standard constructor will be created.

- Initialisation of <attribute>s will only be done in the standard constructor by default.

- If no <destructor> with 0 arguments was provided, a default destructor will be created.

- If an attribute type is complex, the set- and get-methods for it will use const references when using it as an argument. Otherwise the argument will be passed by value.

- For the accessor methods for <relation> the type will always be embedded into a smart pointer type which was developed for this purpose.

- Types which are forward declared before the class will be imported afterwards.

- Relation types are forward declared by default. No include is necessary for the class declaration.

- If no sub element <code> was provided for <constructor>, <destructor> or <method>, only the declarations of these functions will be generated. Implementations are supposed to be provided externally.

- The default for the return type of self defined methods is void.

These rules will only be met for the code that will be generated by the back ends for the C++ header files and their corresponding reflection information.

6.3.3 Not Verifiable Rules

Unfortunately there are some rules in the LHCb coding conventions which are difficult to check or cannot be met easily.

- Every variable or class should have a meaningful name.

- Class names shall be nouns or noun phrases.

- Function names shall be verbs or verb phrases

- Functions that create a new object should start with "make" or "create"

The first three items heavily rely on the use of a dictionary system to check the created names and their sense, but even with the use of a dictionary it will not be clear whether this could be successful, because sometimes developers will create words which are well introduced in the community and understood but not part of a dictionary (e.g. "perstistency") .

It will though be easier to check these rules after the files have been created with some specific tools which are specialised on the checking of coding rules. In the LHCb software framework such tools exists which, by default, will check the implementation code and complain if the coding rules are not met.

6.4 C++ Tools

The minimal requirements for LHCb using C++ as implementation language were two tools:

- GODWriteCppHeader: This tool will create a representation of the event objects in C++ header files. The tool has several options which can be passed to it (see Listing 6.3). Most of the options were introduced to ease the integration of the tool into the build procedure (see Section 6.4.2).

- GODWriteCppDict: This tool will create the corresponding reflection information source code for the event objects for the C++ language. The tool will accept the same options as GODWriteCppHeader (except the environment variable 'GODDOTHOUT' will be called 'GODDICTOUT').

Listing 6.3: Usage statement of GODWriteCppHeader.exe

```
Usage: GODWriteCppHeader.exe [-h] [-v] [-i] [-o [path]] [-x [path]] xml-file(s)
Produce .h-files out of xml-files

    -h              display this help and exit
    -v              display version information and exit
    -i              add additional file-package-information from './AddImports.txt'
    -o [path]       define possible outputdestination with following precedence
                        -o path     use 'path'
                        -o          use environment-variable 'GODDOTHOUT'
                        default     use local directory
    -x [path]       define location of 'GaudiCppExport.xml' which holds information
                    about include-file<->package dependencies, with this precedence
                        -x path     use 'path'
                        -x          use environment-variable 'GODXMLDB'
                        default     use '$(GAUDIOBJDESCROOT)/xml_files'
    xml-file(s)     xml-file(s) to be parsed (must have extension '.xml')
```

6.4.1   Build Procedure

The whole procedure how event packages in C++ are built in LHCb is shown in Figure 6.2.

- The developer of the event package will provide at most two types of files:
    - The XML description of the event package
    - If necessary the implementations of member methods which were only declared in the description files.

- GOD will generate both the C++ header files for the event objects and their dictionary information.

- The header files and the handwritten implementation files will be compiled into the event library.

- The dictionary information will be compiled into a separate library and linked against the event library

Figure 6.2: Build Procedure for the C++ Implementation
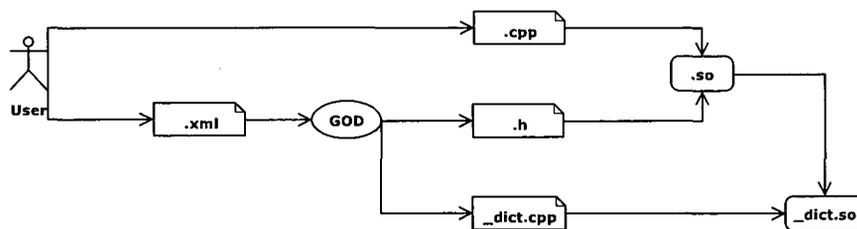
6.4.2   Integration with the Build System

The source code of the packages of the LHCb software is stored with the concurrent versions system (cvs) [17]. If developers want to modify a package, they will check it out from the cvs server and use a tool for the configuration and building of the package, which is called cmt [2, 16]. Cmt generates automatically the configuration, makefiles and environment settings for a given package on a given platform.

In case of the packages which contain the sources for the event objects, only the XML descriptions of the objects will be stored on cvs. When checking out an event package, cmt will be used to automatically call GODWriteCppHeader and GODWriteCppDict to generate the source code and the reflection information for the event objects. After this step the sources will be compiled into libraries.

Integrating the generation of the source code into an automatic procedure when building packages facilitates both the life of developers when using a package and of release managers when a new version of the LHCb software will be produced.

Chapter 7

Evaluation and Outlook

This chapter contains the evaluation of the code generation tools and a short outlook to future developments and possible improvements of the tools.

7.1 Evaluation of Gaudi Object Description

7.1.1 Description Language

Although having the drawback to be a verbose language it turned out that XML was a good choice for the description language for several reasons.

Language Enhancements

Describing the syntax of the description language with a DTD allowed to start with a minimal subset of elements and enhance the language when new functionalities were requested by the user community. Examples for enhancements to the language were bit fields or the location element which were added to the GOD language later.

To enhance the tools with some new functionality three steps need to be carried out:

- The syntax of the description language has to be changed

- The front end has to be made aware of the new concept

- The back ends need to retrieve the new information and produce the corresponding output

This three steps allowed short development cycles and produce quickly new versions of the GOD tools with new functionalities.

Language Independence

The Gaudi framework and the LHCb software on top of it are currently implemented in C++ using object oriented features. The design of the GOD tools was done, with the probability in mind, that implementation techniques or even languages may change to which the tools need to be adapted.

In case of new implementation techniques the different back ends will be changed accordingly. An example for a new implementation technique was the introduction of a new container class for event objects to which the GOD tools were adapted.

In case of new implementation languages they will also have to adhere to object oriented programming concepts, because Gaudi is based on them and also the LHCb software. As the common denominator of the currently available object oriented languages is reflected in the object description language it should be possible to describe the LHCb event model with any language which also complies to concepts of object oriented programming. In chapter 4 it can be seen that the implementation of one object in different object oriented languages is very similar and that these implementations can be fulfilled with the language at hand.

7.1.2 Tools

Execution Time

On an average machine (800 Mhz) running a GOD tool to produce C++ header files or reflection information for an event package takes less than 1 second. As compilation of an event package takes much more time, the introduction of GOD into LHCb was not slowing down the build procedure of software.

Produced Output

One important requirement when designing the tools, was that the users will have to write a factor less code to describe the event objects, than what will be automatically produced by the generation tools.

The ratio between input and output code is calculated on the basis of lines of XML code and its generated C++ code. For the current implementation of the LHCb event model (version 13.0) the input-output ratio of XML code to generated C++ header files is 1:4. The overall ratio from XML code to all generated C++ code is 1:12.

Apart from the direct ratio of a description file to one implementation the amount of all code that was produced by the code generation tools in respect to the rest of the software written in LHCb can be taken into account.

The different pieces of software in LHCb can be seen in Figure 2.1. The lines of code (in thousands) of each of the applications and libraries can be found in Table 7.1.

The event model is part of the general LHCb software and contains around 75.000 lines of code which is around 40 % of the experiment specific code (LHCb). In total LHCb software consists of 854.000 lines of code, out of which the 75.000 lines from the event model are around 9%.

Table 7.1: Libraries and applications in LHCb

Project	kloc
Gaudi	152
LHCb	178
Gauss	110
Boole	71
Brunel	122
DaVinci	199
Panoramix	22

Produced Versions of LHCb Software

The usage of the object description tools by the users in LHCb started in December 2001. Since that time 24 iterations of the LHCb event model were produced. This seems to be a high number, but has to be seen in connection to the fact that the start of the usage of the tools was also the start of the redesign of the LHCb event model which was an urgent task at that time.

7.2 Future Improvements and Outlook

The software for object description was developed with the long lifetime of the experiment in mind. From this point of view the flexibility and extensibility of the software was a major concern. Extensions in the following fields can be carried out.

- *Extensions to the Language:* If needed new concepts for the object description language itself will be introduced. During the development phase of the package it was already proven that extending the language and the depending software is feasible in short development cycles which allow flexible adaptation to upcoming needs of the user community.

- *New Back ends:* Not only changes to the language itself but also new back ends for new languages could be needed in the future that may become important. In that case a new tool will be created. It will make use of the already existing front end and the model that is filled with it. Walking through this model it will output the descriptions of the event model in the syntax of the new language.

- *Integration with LCG software:* The LHC Computing Grid (LCG) is a new project at CERN which aims to provide hard- and software computing facilities for the 4 upcoming experiments. Concerning the LCG software there are already some projects [19, 21, 37]. In the future LHCb will adopt these software packages and integrate them into the Gaudi framework. This will also require changes to the C++ and reflection back end of the GOD tools.

Chapter 8

Conclusion

In this thesis a new technique for the description of event data objects for the LHCb experiment has been presented.

A high level description language for event objects using XML and a DTD was introduced. With a description of the complexity of the underlying event data model, it was shown that a subset of object oriented modelling techniques will be sufficient to implement an event model with enough detail. It was also shown that the requirements of the LHCb experiment for the implementation of an event model can be sufficiently met with the description language proposed.

A system using this high level description language producing concrete implementations of objects was described. It was shown that the system supports the production of different implementations of event objects. Due to the design of the system for handling the event object descriptions, enhancements to the description language can be achieved and implemented in short development cycles. Examples for enhancements are introduction of new functionalities or adaptations of the implementations of the event objects to new programming techniques.

Together with the implementation of the system producing C++ objects, a package which provides reflection information for C++ objects was introduced. It was shown that reflection is important for tasks like persistence of objects or interactive programming and that the reflection information about the C++ event objects in LHCb is already used for these tasks.

It was also shown that it is possible to describe the event model of LHCb with a concise high level description language which is easy to use and produce. As a result of the compressed definition of objects the factor of description code to automatically generated source code for the C++ implementation of the LHCb event model is 1:12. Furthermore the automatic generation of the LHCb event model in C++ together with its reflection information is 40 % of the total amount of source code of the common LHCb software. The object description language and description tools have been used for the generation of the LHCb event model since 2001.

The overall goal of describing the event model for the LHCb experiment in an implementation language independent way throughout the lifetime of the experiment should be achievable with these tools and techniques.

67

Appendix A

LHCb Event Model

This appendix contains the DTD of the description language explaining the syntax that was used for describing the event objects and an example XML description of a class with its corresponding output of the C++ and reflection back end of the system.

A.1 DTD

The DTD (see Listing A.1) contains the syntax description of the Gaudi object description language. The DTD describes in total 19 elements and 93 attributes which is enough for the description of the event model.

Listing A.1: Event Model DTD

```
<?xml encoding="UTF-8"?>
<!ELEMENT gdd ((import*,package+)+)>
<!ATTLIST gdd
        version CDATA "1.0"
>
<!ELEMENT import EMPTY>
<!ATTLIST import
        name CDATA #REQUIRED
        std (TRUE | FALSE) "FALSE"
        soft (TRUE | FALSE) "FALSE"
        ignore (TRUE | FALSE) "FALSE"
>
<!ELEMENT package ((import*,class*,namespace*)*)>
<!ATTLIST package
        name CDATA #REQUIRED
>
<!ELEMENT class ((desc?, base?, import?, location?, enum?,
                  typedef?, constructor?, destructor?,
                  method?, attribute?, relation?)*)>
<!ATTLIST class
        name CDATA #REQUIRED
        author CDATA #REQUIRED
        desc CDATA #REQUIRED
        filename CDATA #IMPLIED
        id CDATA #IMPLIED
        location CDATA #IMPLIED
        stdVectorTypeDef (TRUE | FALSE) "FALSE"
        keyedContTypeDef (TRUE | FALSE) "FALSE"
        templateVector (TRUE | FALSE) "TRUE"
        templateList (TRUE | FALSE) "TRUE"
        serializers (TRUE | FALSE) "TRUE"
>
<!ELEMENT desc (#PCDATA)>
<!ATTLIST desc
        xml:space (default | preserve) #FIXED "preserve"
>
<!ELEMENT location EMPTY>
<!ATTLIST location
        name CDATA #REQUIRED
        place CDATA #REQUIRED
        noQuote (TRUE | FALSE) "FALSE"
>
<!ELEMENT namespace ((desc?, typedef?, enum?, class?, import?,
                  attribute?, method?)*)>
```

```
<!ATTLIST namespace
        name CDATA #REQUIRED
        author CDATA #IMPLIED
        desc CDATA #REQUIRED
>
<!ELEMENT base EMPTY>
<!ATTLIST base
        name CDATA #REQUIRED
        virtual (TRUE | FALSE) "FALSE"
        access (PUBLIC | PROTECTED | PRIVATE) "PUBLIC"
>
<!ELEMENT enum EMPTY>
<!ATTLIST enum
        name CDATA #REQUIRED
        desc CDATA #REQUIRED
        value CDATA #REQUIRED
        access (PUBLIC | PROTECTED | PRIVATE) "PRIVATE"
>
<!ELEMENT typedef EMPTY>
<!ATTLIST typedef
        desc CDATA #REQUIRED
        type CDATA #REQUIRED
        def CDATA #REQUIRED
        access (PUBLIC | PROTECTED | PRIVATE) "PRIVATE"
>
<!ELEMENT constructor (arg*, code?, arg*)>
<!ATTLIST constructor
        desc CDATA #REQUIRED
        argList CDATA #IMPLIED
        argInOut CDATA #IMPLIED
        initList CDATA #IMPLIED
>
<!ELEMENT destructor (arg*, code?, arg*)>
<!ATTLIST destructor
        desc CDATA #REQUIRED
        argList CDATA #IMPLIED
        argInOut CDATA #IMPLIED
>
<!ELEMENT method ((arg*, return?, arg*, code?, arg*) |
                  (arg*, code?, arg*, return?, arg*))>
<!ATTLIST method
        name CDATA #REQUIRED
        desc CDATA #REQUIRED
        template CDATA #IMPLIED
        access (PUBLIC | PROTECTED | PRIVATE) "PUBLIC"
        const (TRUE | FALSE) "FALSE"
        virtual (TRUE | FALSE | PURE) "FALSE"
        static (TRUE | FALSE) "FALSE"
        inline (TRUE | FALSE) "FALSE"
        friend (TRUE | FALSE) "FALSE"
        type CDATA "void"
        argList CDATA #IMPLIED
        argInOut CDATA #IMPLIED
>
<!ELEMENT arg EMPTY>
<!ATTLIST arg
        type CDATA #REQUIRED
        name CDATA #IMPLIED
        const (TRUE | FALSE) "FALSE"
        inout (BYVALUE | INPUT | INOUT | BOTH) "INPUT"
>
<!ELEMENT return EMPTY>
<!ATTLIST return
        type CDATA #REQUIRED
        const (TRUE | FALSE) "FALSE"
>
<!ELEMENT code (#PCDATA)>
<!ATTLIST code
        lang (CPP | JAVA) "CPP"
        xml:space (default | preserve) #FIXED "preserve"
>
<!ELEMENT attribute (bitfield*)>
<!ATTLIST attribute
        type CDATA #REQUIRED
        name CDATA #REQUIRED
        desc CDATA #REQUIRED
        init CDATA #IMPLIED
        array CDATA "FALSE"
        access (PUBLIC | PROTECTED | PRIVATE) "PRIVATE"
        compression (TRUE | FALSE) "TRUE"
        serialize (TRUE | FALSE) "TRUE"
        setMeth (TRUE | FALSE) "TRUE"
        getMeth (TRUE | FALSE) "TRUE"
>
<!ELEMENT relation EMPTY>
<!ATTLIST relation
        type CDATA #REQUIRED
        name CDATA #REQUIRED
        desc CDATA #REQUIRED
        access (PUBLIC | PROTECTED | PRIVATE) "PRIVATE"
        multiplicity (1 | N | n | M | m) "1"
        serialize (TRUE | FALSE) "TRUE"
        setMeth (TRUE | FALSE) "TRUE"
        getMeth (TRUE | FALSE) "TRUE"
        addMeth (TRUE | FALSE) "TRUE"
        remMeth (TRUE | FALSE) "TRUE"
        clrMeth (TRUE | FALSE) "TRUE"
>
<!ELEMENT bitfield EMPTY>
```

```
<!ATTLIST bitfield
        name CDATA #REQUIRED
        length CDATA #REQUIRED
        desc CDATA #REQUIRED
        startBit CDATA #IMPLIED
        setMeth (TRUE | FALSE) "TRUE"
        getMeth (TRUE | FALSE) "TRUE"
>
```

A.2 Example Class

To demonstrate parts of the capabilities of the system the current implementation of the MCParticle class was chosen. In Listing A.2 the XML description of the MCParticle class is shown. The MCParticle class is a typical class of the event model. It describes 1 base class, 3 attributes, 3 relations and 3 self-defined methods, one containing the implementation code while the other two are implemented in an external file.

Listing A.2: MCParticle.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gdd SYSTEM "gdd.dtd">
<gdd>
  <package name="Event">
    <class
      author="Gloria Corti"
      desc="The Monte Carlo particle kinematics information"
      id="210"
      name="MCParticle"
      location="MC/Particles"
    >
        <base name="KeyedObject&lt;int&gt;"/>
        <attribute
          desc="4-momentum-vector"
          name="momentum"
          type="HepLorentzVector"
          init="0.0, 0.0, 0.0, 0.0"
        />
        <attribute
          desc="Particle ID"
          name="particleID"
          type="ParticleID"
          init="0"
        />
        <attribute
          desc="Describe if a particle has oscillated"
          init="false"
          name="hasOscillated"
          type="bool"
        />
        <attribute
          desc="Helicity"
          name="helicity"
          type="double"
        />
        <method
          const="TRUE"
          desc="Retrieve virtual mass"
          name="virtualMass"
          type="double"
        >
            <code>  return m_momentum.m(); </code>
        </method>
        <method
          const="TRUE"
          desc="Short-cut to pt value"
          name="pt"
          type="double"
        >
            <code> return m_momentum.perp(); </code>
        </method>
        <method
          const="TRUE"
          desc="Pointer to parent particle"
          name="mother"
          type="const MCParticle*"
        >
            <code>
if( originVertex() ) {
  return originVertex()->mother();
}
else {
  return 0;
}
            </code>
```

```
    </method>
    <relation
     desc="Pointer to origin vertex"
     name="originVertex"
     type="MCVertex"
    />
    <relation
     desc="Vector of pointers to decay vertices"
     multiplicity="M"
     name="endVertices"
     type="MCVertex"
    />
    <relation
     desc="Pointer to Collision to which the vertex belongs to"
     name="collision"
     type="Collision"
    />
   </class>
  </package>
</gdd>
```

Running the C++ back end on the description file above will produce the
C++ header file in Listing A.3. This file contains 374 lines of automatically
generated C++ code which reflects the description in Listing A.2.

The automatically generated C++ header file contains the following items:

- Some warning for the developers to not alter this file

- The header guards which prevent multiple inclusion of the header file

- The include files needed for compilation of this file

- The forward declarations for types which don't need to be included immediately

- The class id of the object

- A declaration where to find the object in the transient store

- A description of the class

- The class with:

  - The public part which contains declarations of:

    * The constructor with all initialisation values for attributes
    * The default destructor
    * Some methods to retrieve the class ID
    * The get- and set-methods for attributes
    * The get-, set-, add-, remove- and clear-methods for relations
    * The self defined methods
    * Methods for serialisation of the object
    * A method to dump the information of the object to a stream

  - An empty protected part

  - A private part which contains

    * The attributes of the object
    * The relations of the object

- Some more include files which are put after the class

- The implementations of the functions declared in the public part of the class

- A typedef which is used for access of the object in the transient store

- The second part of the header guard

Listing A.3: MCParticle.h

```
//  *************************************************************************
//  *                                                                       *
//  *                    ! ! ! A T T E N T I O N ! ! !                       *
//  *                                                                       *
//  *  This file was created automatically by GaudiObjDesc, please do not    *
//  *  delete it or edit it by hand.                                         *
//  *                                                                       *
//  *  If you want to change this file, first change the corresponding       *
//  *  xml-file and rerun the tools from GaudiObjDesc (or run make if you     *
//  *  are using it from inside a Gaudi-package).                            *
//  *                                                                       *
//  *************************************************************************

#ifndef Event_MCParticle_H
#define Event_MCParticle_H 1

// Include files
#include <algorithm>
#include "Kernel/CLHEPStreams.h"
#include "CLHEP/Vector/LorentzVector.h"
#include "Event/KeyedObject.h"
#include "GaudiKernel/ObjectList.h"
#include "GaudiKernel/ObjectVector.h"
#include "Event/ParticleID.h"
#include "GaudiKernel/SmartRef.h"
#include "GaudiKernel/SmartRefVector.h"
#include "GaudiKernel/StreamBuffer.h"


// Forward declarations
class Collision;
class MCVertex;


// Class ID definition
static const CLID& CLID_MCParticle = 210;


// Namespace for locations in TDS
namespace MCParticleLocation {
  static const std::string& Default = "MC/Particles";
}


/** @class MCParticle MCParticle.h
 *
 *  The Monte Carlo particle kinematics information
 *
 *  @author Gloria Corti
 *  created Thu Jul 17 09:15:30 2003
 *
 */

class MCParticle: public KeyedObject<int>
{

public:

  /// Default Constructor
  MCParticle()
      : m_momentum(0.0, 0.0, 0.0, 0.0),
      m_particleID(0),
      m_hasOscillated(false),
      m_helicity(0.0) {}

  /// Destructor
  virtual ~MCParticle() {}

  /// Retrieve pointer to class definition structure
  virtual const CLID& clID() const;
  static const CLID& classID();

  /// Retrieve virtual mass
  double virtualMass() const;

  /// Short-cut to pt value
  double pt() const;
```

```
/// Pointer to parent particle
const MCParticle* mother() const;

/// Retrieve 4-momentum-vector
const HepLorentzVector& momentum() const;

/// Retrieve 4-momentum-vector (non-const)
HepLorentzVector& momentum();

/// Update 4-momentum-vector
void setMomentum(const HepLorentzVector& value);

/// Retrieve Particle ID
const ParticleID& particleID() const;

/// Retrieve Particle ID (non-const)
ParticleID& particleID();

/// Update Particle ID
void setParticleID(const ParticleID& value);

/// Retrieve Describe if a particle has oscillated
bool hasOscillated() const;

/// Update Describe if a particle has oscillated
void setHasOscillated(bool value);

/// Retrieve Helicity
double helicity() const;

/// Update Helicity
void setHelicity(double value);

/// Retrieve Pointer to origin vertex (const)
const MCVertex* originVertex() const;

/// Retrieve Pointer to origin vertex (non-const)
MCVertex* originVertex();

/// Update Pointer to origin vertex
void setOriginVertex(const SmartRef<MCVertex>& value);

/// Retrieve Vector of pointers to decay vertices (const)
const SmartRefVector<MCVertex>& endVertices() const;

/// Retrieve Vector of pointers to decay vertices (non-const)
SmartRefVector<MCVertex>& endVertices();

/// Update Vector of pointers to decay vertices
void setEndVertices(const SmartRefVector<MCVertex>& value);

/// Add Vector of pointers to decay vertices
void addToEndVertices(const SmartRef<MCVertex>& value);

/// Remove Vector of pointers to decay vertices
void removeFromEndVertices(const SmartRef<MCVertex>& value);

/// Clear Vector of pointers to decay vertices
void clearEndVertices();

/// Retrieve Pointer to Collision to which the vertex belongs to (const)
const Collision* collision() const;

/// Retrieve Pointer to Collision to which the vertex belongs to (non-const)
Collision* collision();

/// Update Pointer to Collision to which the vertex belongs to
void setCollision(const SmartRef<Collision> value);

/// Serialize the object for writing
virtual StreamBuffer& serialize(StreamBuffer& s) const;

/// Serialize the object for reading
virtual StreamBuffer& serialize(StreamBuffer& s);

/// Fill the ASCII output stream
virtual std::ostream& fillStream(std::ostream& s) const;

protected:

private:

HepLorentzVector        m_momentum;      ///< 4-momentum-vector
ParticleID              m_particleID;    ///< Particle ID
bool                    m_hasOscillated; ///< Describe if a particle has oscill
double                  m_helicity;      ///< Helicity
SmartRef<MCVertex>      m_originVertex;  ///< Pointer to origin vertex
SmartRefVector<MCVertex> m_endVertices;  ///< Vector of pointers to decay vrtcs
SmartRef<Collision>     m_collision;     ///< Pointer to Collision to which ...

};


// ------------------------------------------------------------------------
//    end of class
// ------------------------------------------------------------------------


// Including forward declarations
```

```
#include "Event/MCVertex.h"
#include "Event/Collision.h"


inline const CLID& MCParticle::clID() const
{
  return MCParticle::classID();
}

inline const CLID& MCParticle::classID()
{
  return CLID_MCParticle;
}

inline double MCParticle::virtualMass() const
{
    return m_momentum.m();
}

inline double MCParticle::pt() const
{
    return m_momentum.perp();
}

inline const MCParticle* MCParticle::mother() const
{

  if( originVertex() ) {
    return originVertex()->mother();
  }
  else {
    return 0;
  }

}

inline const HepLorentzVector& MCParticle::momentum() const
{
  return m_momentum;
}

inline HepLorentzVector& MCParticle::momentum()
{
  return m_momentum;
}

inline void MCParticle::setMomentum(const HepLorentzVector& value)
{
  m_momentum = value;
}

inline const ParticleID& MCParticle::particleID() const
{
  return m_particleID;
}

inline ParticleID& MCParticle::particleID()
{
  return m_particleID;
}

inline void MCParticle::setParticleID(const ParticleID& value)
{
  m_particleID = value;
}

inline bool MCParticle::hasOscillated() const
{
  return m_hasOscillated;
}

inline void MCParticle::setHasOscillated(bool value)
{
  m_hasOscillated = value;
}

inline double MCParticle::helicity() const
{
  return m_helicity;
}

inline void MCParticle::setHelicity(double value)
{
  m_helicity = value;
}

inline const MCVertex* MCParticle::originVertex() const
{
    return m_originVertex;
}

inline MCVertex* MCParticle::originVertex()
{
    return m_originVertex;
}

inline void MCParticle::setOriginVertex(const SmartRef<MCVertex>& value)
{
    m_originVertex = value;
```

```
}

inline const SmartRefVector<MCVertex>& MCParticle::endVertices() const
{
    return m_endVertices;
}

inline SmartRefVector<MCVertex>& MCParticle::endVertices()
{
    return m_endVertices;
}

inline void MCParticle::setEndVertices(const SmartRefVector<MCVertex>& value)
{
    m_endVertices = value;
}

inline void MCParticle::addToEndVertices(const SmartRef<MCVertex>& value)
{
    m_endVertices.push_back(value);
}

inline void MCParticle::removeFromEndVertices(const SmartRef<MCVertex>& value)
{
    SmartRefVector<MCVertex>::iterator iter =
        std::remove(m_endVertices.begin(), m_endVertices.end(), value);
    m_endVertices.erase(iter, m_endVertices.end());
}

inline void MCParticle::clearEndVertices()
{
    m_endVertices.clear();
}

inline const Collision* MCParticle::collision() const
{
    return m_collision;
}

inline Collision* MCParticle::collision()
{
    return m_collision;
}

inline void MCParticle::setCollision(const SmartRef<Collision>& value)
{
    m_collision = value;
}

inline StreamBuffer& MCParticle::serialize(StreamBuffer& s) const
{
    unsigned char l_hasOscillated = (m_hasOscillated) ? 1 : 0;
    KeyedObject<int>::serialize(s);
    s << m_momentum
      << m_particleID
      << l_hasOscillated
      << (float)m_helicity
      << m_originVertex(this)
      << m_endVertices(this)
      << m_collision(this);
    return s;
}

inline StreamBuffer& MCParticle::serialize(StreamBuffer& s)
{
    unsigned char l_hasOscillated;
    float l_helicity;
    KeyedObject<int>::serialize(s);
    s >> m_momentum
      >> m_particleID
      >> l_hasOscillated
      >> l_helicity
      >> m_originVertex(this)
      >> m_endVertices(this)
      >> m_collision(this);
    m_hasOscillated = (l_hasOscillated) ? true : false;
    m_helicity = l_helicity;
    return s;
}

inline std::ostream& MCParticle::fillStream(std::ostream& s) const
{
    char l_hasOscillated = (m_hasOscillated) ? 'T' : 'F';
    s << "{ "
      << " momentum:\t" << m_momentum << std::endl
      << "   particleID:\t" << m_particleID << std::endl
      << "   hasOscillated:\t" << l_hasOscillated << std::endl
      << "   helicity:\t" << (float)m_helicity << " } ";
    return s;
}


//Defintion of keyed container for MCParticle
typedef KeyedContainer<MCParticle, Containers::HashMap> MCParticles;


#endif   ///Event_MCParticle_H
```

The dictionary information of the MCParticle class described in Listing A.2, can be found in the subsequent Listing A.4.

Listing A.4: MCParticle_dict.cpp

```
//   ************************************************************************
//   *                                                                      *
//   *                        ! ! ! A T T E N T I O N ! ! !                  *
//   *                                                                      *
//   *   This file was created automatically by GaudiObjDesc, please do not  *
//   *   delete it or edit it by hand.                                       *
//   *                                                                      *
//   *   If you want to change this file, first change the corresponding     *
//   *   xml-file and rerun the tools from GaudiObjDesc (or run make if you  *
//   *   are using it from inside a Gaudi-package).                          *
//   *                                                                      *
//   ************************************************************************


//Include files
#include "GaudiKernel/Kernel.h"
#include "GaudiKernel/SmartRef.h"
#include "GaudiKernel/SmartRefVector.h"

#include <string>

#include "MCParticle_dict.h"

#include "GaudiIntrospection/Introspection.h"


//----------------------------------------------------------------------------
class MCParticle_dict
//----------------------------------------------------------------------------
{
public:
  MCParticle_dict();
};

//----------------------------------------------------------------------------
static void* MCParticle_virtualMass_0const (void* v)
//----------------------------------------------------------------------------
{
  static double ret;
  ret = ((MCParticle*)v)->virtualMass();
  return &ret;
}

//----------------------------------------------------------------------------
static void* MCParticle_pt_1const (void* v)
//----------------------------------------------------------------------------
{
  static double ret;
  ret = ((MCParticle*)v)->pt();
  return &ret;
}

//----------------------------------------------------------------------------
static void* MCParticle_mother_2const (void* v)
//----------------------------------------------------------------------------
{
  return (MCParticle*) ((MCParticle*)v)->mother();
}

//----------------------------------------------------------------------------
static void* MCParticle_momentum_3 (void* v)
//----------------------------------------------------------------------------
{
  const HepLorentzVector& ret = ((MCParticle*)v)->momentum();
  return (void*)&ret;
}

//----------------------------------------------------------------------------
static void MCParticle_setMomentum_4 (void* v, std::vector<void*> argList)
//----------------------------------------------------------------------------
{
  ((MCParticle*)v)->setMomentum(*(HepLorentzVector*)argList[0]);
}

//----------------------------------------------------------------------------
static void* MCParticle_particleID_5 (void* v)
//----------------------------------------------------------------------------
{
  const ParticleID& ret = ((MCParticle*)v)->particleID();
  return (void*)&ret;
}

//----------------------------------------------------------------------------
static void MCParticle_setParticleID_6 (void* v, std::vector<void*> argList)
//----------------------------------------------------------------------------
{
  ((MCParticle*)v)->setParticleID(*(ParticleID*)argList[0]);
}
```

```
//--------------------------------------------------------------------------
static void* MCParticle_hasOscillated_7(void* v)
//--------------------------------------------------------------------------
{
  static bool ret;
  ret = ((MCParticle*)v)->hasOscillated();
  return (void*)&ret;
}

//--------------------------------------------------------------------------
static void MCParticle_setHasOscillated_8(void* v, std::vector<void*> argList)
//--------------------------------------------------------------------------
{
  ((MCParticle*)v)->setHasOscillated(*(bool*)argList[0]);
}

//--------------------------------------------------------------------------
static void* MCParticle_helicity_9(void* v)
//--------------------------------------------------------------------------
{
  static double ret;
  ret = ((MCParticle*)v)->helicity();
  return (void*)&ret;
}

//--------------------------------------------------------------------------
static void MCParticle_setHelicity_10(void* v, std::vector<void*> argList)
//--------------------------------------------------------------------------
{
  ((MCParticle*)v)->setHelicity(*(double*)argList[0]);
}

//--------------------------------------------------------------------------
static void* MCParticle_originVertex_11(void* v)
//--------------------------------------------------------------------------
{
  MCVertex* ret = ((MCParticle*)v)->originVertex();
  return (void*)ret;
}

//--------------------------------------------------------------------------
static void MCParticle_setOriginVertex_12(void* v, std::vector<void*> argList)
//--------------------------------------------------------------------------
{
  ((MCParticle*)v)->setOriginVertex((MCVertex*)argList[0]);
}

//--------------------------------------------------------------------------
static void* MCParticle_endVertices_13(void* v)
//--------------------------------------------------------------------------
{
  const SmartRefVector<MCVertex>& ret = ((MCParticle*)v)->endVertices();
  return (void*)&ret;
}

//--------------------------------------------------------------------------
static void MCParticle_setEndVertices_14(void* v, std::vector<void*> argList)
//--------------------------------------------------------------------------
{
  ((MCParticle*)v)->setEndVertices(*(SmartRefVector<MCVertex>*)argList[0]);
}

//--------------------------------------------------------------------------
static void MCParticle_clearEndVertices_15(void* v)
//--------------------------------------------------------------------------
{
  ((MCParticle*)v)->clearEndVertices();
}

//--------------------------------------------------------------------------
static void MCParticle_addToEndVertices_16(void* v, std::vector<void*> argList)
//--------------------------------------------------------------------------
{
  ((MCParticle*)v)->addToEndVertices((MCVertex*)argList[0]);
}

//--------------------------------------------------------------------------
static void MCParticle_removeFromEndVertices_17(void* v, std::vector<void*> argList)
//--------------------------------------------------------------------------
{
  ((MCParticle*)v)->removeFromEndVertices((MCVertex*)argList[0]);
}

//--------------------------------------------------------------------------
static void* MCParticle_collision_18(void* v)
//--------------------------------------------------------------------------
{
  Collision* ret = ((MCParticle*)v)->collision();
  return (void*)ret;
}

//--------------------------------------------------------------------------
static void MCParticle_setCollision_19(void* v, std::vector<void*> argList)
//--------------------------------------------------------------------------
{
  ((MCParticle*)v)->setCollision((Collision*)argList[0]);
}
```

```cpp
//---------------------------------------------------------------------------
static void* MCParticle_constructor_1()
{
    static MCParticle* ret = new MCParticle();
    return ret;
}
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
static MCParticle_dict instance;
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
MCParticle_dict::MCParticle_dict()
{
    std::vector<std::string> argTypes = std::vector<std::string>();

    MetaClass* metac = new MetaClass("MCParticle",
        "The Monte Carlo particle kinematics information",
        0);
    MCParticle* c1 = new MCParticle();
    metac->addSuperClass("KeyedObject<int>",
        (((int)(c1))-((int)((KeyedObject<int>*)(c1)))));
    delete c1;

    metac->addConstructor("default constructor",
        MCParticle_constructor_1);

    metac->addField("momentum",
        "HepLorentzVector",
        "4-momentum-vector",
        OffsetOf(MCParticle, m_momentum),
        MetaModifier::setPrivate());

    metac->addField("particleID",
        "ParticleID",
        "Particle ID",
        OffsetOf(MCParticle, m_particleID),
        MetaModifier::setPrivate());

    metac->addField("hasOscillated",
        "bool",
        "Describe if a particle has oscillated",
        OffsetOf(MCParticle, m_hasOscillated),
        MetaModifier::setPrivate());

    metac->addField("helicity",
        "double",
        "Helicity",
        OffsetOf(MCParticle, m_helicity),
        MetaModifier::setPrivate());

    metac->addField("originVertex",
        "SmartRef<MCVertex>",
        "Pointer to origin vertex",
        OffsetOf(MCParticle, m_originVertex),
        MetaModifier::setPrivate());

    metac->addField("endVertices",
        "SmartRefVector<MCVertex>",
        "Vector of pointers to decay vertices",
        OffsetOf(MCParticle, m_endVertices),
        MetaModifier::setPrivate());

    metac->addField("collision",
        "SmartRef<Collision>",
        "Pointer to collision to which the vertex belongs to",
        OffsetOf(MCParticle, m_collision),
        MetaModifier::setPrivate());

    metac->addMethod("virtualMass",
        "Retrieve virtual mass",
        "double",
        MCParticle_virtualMass_0const);
    argTypes.clear();

    metac->addMethod("pt",
        "Short-cut to pt value",
        "double",
        MCParticle_pt_1const);
    argTypes.clear();

    metac->addMethod("mother",
        "Pointer to parent particle",
        "MCParticle*",
        MCParticle_mother_2const);
    argTypes.clear();

    metac->addMethod("momentum",
        "4-momentum-vector",
        "HepLorentzVector",
        MCParticle_momentum_3);
    argTypes.clear();
    argTypes.push_back("HepLorentzVector");
    metac->addMethod("setMomentum",
        "4-momentum-vector",
        argTypes,
```

```
                      MCParticle_setMomentum_4 );

    metaC->addMethod("particleID",
                     "Particle ID",
                     "ParticleID",
                     MCParticle_particleID_5 );

    argTypes.clear();
    argTypes.push_back("ParticleID");
    metaC->addMethod("setParticleID",
                     "Particle ID",
                     argTypes,
                     MCParticle_setParticleID_6 );

    metaC->addMethod("hasOscillated",
                     "Describe if a particle has oscillated",
                     "bool",
                     MCParticle_hasOscillated_7 );

    argTypes.clear();
    argTypes.push_back("bool");
    metaC->addMethod("setHasOscillated",
                     "Describe if a particle has oscillated",
                     argTypes,
                     MCParticle_setHasOscillated_8 );

    metaC->addMethod("helicity",
                     "Helicity",
                     "double",
                     MCParticle_helicity_9 );

    argTypes.clear();
    argTypes.push_back("double");
    metaC->addMethod("setHelicity",
                     "Helicity",
                     argTypes,
                     MCParticle_setHelicity_10 );

    metaC->addMethod("originVertex",
                     "Pointer to origin vertex",
                     "MCVertex",
                     MCParticle_originVertex_11 );

    argTypes.clear();
    argTypes.push_back("MCVertex");
    metaC->addMethod("setOriginVertex",
                     "Pointer to origin vertex",
                     argTypes,
                     MCParticle_setOriginVertex_12 );

    metaC->addMethod("endVertices",
                     "Vector of pointers to decay vertices",
                     "SmartRefVector<MCVertex>",
                     MCParticle_endVertices_13 );

    argTypes.clear();
    argTypes.push_back("MCVertex");
    metaC->addMethod("setEndVertices",
                     "Vector of pointers to decay vertices",
                     argTypes,
                     MCParticle_setEndVertices_14 );

    metaC->addMethod("clearEndVertices",
                     "Vector of pointers to decay vertices",
                     MCParticle_clearEndVertices_15 );

    argTypes.clear();
    argTypes.push_back("MCVertex");
    metaC->addMethod("addToEndVertices",
                     "Vector of pointers to decay vertices",
                     argTypes,
                     MCParticle_addToEndVertices_16 );

    argTypes.clear();
    argTypes.push_back("MCVertex");
    metaC->addMethod("removeFromEndVertices",
                     "Vector of pointers to decay vertices",
                     argTypes,
                     MCParticle_removeFromEndVertices_17 );

    metaC->addMethod("collision",
                     "Pointer to Collision to which the vertex belongs to",
                     "Collision",
                     MCParticle_collision_18 );

    argTypes.clear();
    argTypes.push_back("Collision");
    metaC->addMethod("setCollision",
                     "Pointer to Collision to which the vertex belongs to",
                     argTypes,
                     MCParticle_setCollision_19 );

    MetaPropertyList* pl = new MetaPropertyList();
    pl->setProperty("Author", "Gloria Corti");
    pl->setProperty("ClassID", "210");
    metaC->setPropertyList(pl);
}
```

Appendix B

Reflection in C++

In this chapter an approach for reflection in C++ will be presented. After an introduction into the problem of reflection in C++, some of the technical details of the model in conjunction with examples how to build and use the reflection system will be discussed. This will be followed by a list of current and envisaged implementations which give an insight about the usability of the model. The chapter will be concluded by an outlook to a new approach on how to implement C++ reflection.

B.1 Introduction

Computational Reflections is the ability of a system to maintain information about its internal state and the possibility to change it. Reflection in a programming-language enables it to:

- Retrieve information about classes that were loaded before

- Retrieve information about fields or methods of a given object, like their type or name

- Interact with the fields of these objects e.g. getting or setting its values

- Interact with methods of a given object like calling methods and retrieving their return-values

- Construct new instances of classes

There are already several languages like Java available which provide reflection information. E.g. with the java.lang.reflect [32, 33] package it is possible to accomplish all the tasks listed above.

Other examples of languages using reflection information are Smalltalk, Lisp, Scheme, Prolog, Python or Tcl/Tk.

The C++ language itself has only limited means of reflection about its objects which is called the runtime type information (RTTI) [53]. RTTI is used for example for tasks like dynamic casting of objects to check whether objects are compatible or not. The typeid() function will return the internal type_name object which represents an object type in C++. The possibilities of querying this type_name object are limited to getting its internal name as a string. It is further only possible to retrieve RTTI information about an object if the class describing the object contains virtual functions.

In the last years there were several approaches to introduce reflection also to C++ [14, 15, 26, 35]. These approaches were either only implemented as a proof of concept and so not complete enough to be used in a concrete application or are part of a system which would needed to be integrated into the application software. In the next sections a general stand alone way of generating meta information about objects will be introduced together with an explanation on how to load and use it with a proposed API.

B.2 The Model

B.2.1 Current Implementation

The first version of the model was implemented with the java.lang.reflect model in mind. This was done for two reasons, mainly because of the easiness of the Java approach and also because Java is similar to C++ and so no major adaptations were expected in subsequent iterations. In Figure B.1 an object-model of this first implementation can be seen. The model is split into two packages called 'Reflection' and 'ReflectionBuilder'. While 'Reflection' is the API to provide the information to the users, the 'ReflectionBuilder' is capable of filling the model with the information needed. This split was done because of security-reasons and also because of the possibility to develop the two packages in a more independent way.

The basic type of this model is the item class. All C++ concepts of this class are derived from it. The introduction of the PropertyList class allows to store all information that is not part of the C++ ISO standard [52]. This PropertyList can be attached to every meta item during the building process of the model (see section B.2.2). Examples for properties are the author name or class ID.

B.2.2 Generation and Use of the Reflection

After explanation of the model that was designed and implemented so far the chain for obtaining reflection information is the following:

- First the meta information about the objects needs to be generated.

- This information will be compiled into a library and loaded into the system.

- After the loading was completed the meta-information about the objects is available and users may access them through the reflection API.

Figure B.1: First Implementation of the Reflection Model
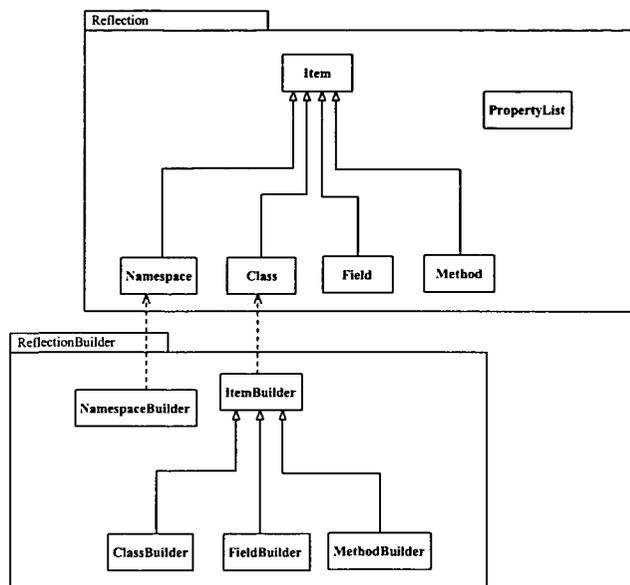
Dictionary Generation

For a given example class (see Listing B.1) the corresponding meta information (see Listing B.2) may be currently generated using the ReflectionBuilder interface in three different ways.

Listing B.1: Example Class

```
1    class MyClass : public MyBaseClass {
2    public:
3      int myMethod() const;
4    private:
5      int myVar;
6    };
```

- The dictionary information may be deduced from some high level description of objects. This approach was chosen for the event model in LHCb [47]. From an XML description of event objects the C++ header files and their corresponding reflection descriptions are produced.

- If no high level description of objects is available it is possible to parse the C++ header files [37]. The command internally uses a front end of the gcc compiler called gcc_xml [23]. This front end extracts information about the classes in XML format. A python script then interprets the XML and generates the source code of the reflection information.

- The last possibility is to generate the dictionary information by hand. The syntax for dictionaries was designed with the possibility in mind that they

will also be writable and readable by humans and so the writing of such files should not be difficult.

Listing B.2: Meta Information for the Example Class

```
1    class MyClass_dict {
2    public:
3      MyClass_dict();
4
5      static void* MyClass_myMethod(void* v) {
6        static int ret = ((MyClass*)v)->myMethod();
7        return &ret;
8      }
9    };
10
11   MyClass_dict::MyClass_dict() {
12     ClassBuilder C(''MyClass'', ''description of MyClass'',
13                    typeid(MyClass), sizeof(MyClass));
14     C.addField(''myVar'', ''int'', ''description of myVar'',
15                OffsetOf(MyClass,m_myVar));
16     C.addMethod(''myMethod'', ''desc of myMethod'', ''int'',
17                 MyClass_myMethod);
18     C.build();
19   };
```

The meta information about classes which is generated contains also private members and functions. The reflection model was designed in a way that also this private information about classes will be accessible from the user side. If this feature is not desired there are two ways to disable it. The simplest way is just not to generate the information about the private part of the class so it will also not be accessible from the user side. If one wants to disable this private access only in a few cases it will also be able to distinguish on the user API whether a member or function of a class is private or not. A thin layer above the reflection can then trigger the access to this private information.

Another choice that has to be made is whether the generation of dictionary information is intrusive or not, i.e. whether the original C++ code describing the objects needs to be changed or not. The whole problem boils down to the fact that at load time of the dictionary the software needs to have access to the private members of the classes to calculate their offsets.

- Intrusive approaches will modify the original classes in a way that the reflection software gains access to the private areas of the objects. This can be done with friendship between classes.

- The implemented approach to this problem is the more user friendly way to generate dictionary information in a non intrusive way. This can be achieved with some modifications to C++ outside the class which contains the original object. They are mainly based on redefinition of modifiers (e.g. #define private public) before including the header file containing the class in order to gain access to its private (and protected) members. After the class was included the modifiers will be undefined. The prerequisite that is needed for this approach is that every class has the modifiers set

(this is absolutely necessary, as members of a class are private by default if no modifiers are present).

Loading and Filling the Dictionary

After the dictionary-information was generated it will be compiled into a dynamic library. This library when loaded will fill the meta model using the ReflectionBuilder component. While loading, the reflection will keep track of unresolved types. This enables users to check whether the model is already complete or some other libraries with meta information about other types need to be loaded.

Using the Dictionary

After the dictionary information has been loaded successfully users may start interrogating its objects. A simple program illustrating what can be done may be found in Listing B.3. The first thing that needs to be done is entering the model through retrieving an instance of a meta class (Listing B.3, line 1). This instance of the meta class can be either retrieved through the string representation of the class or its type_info. After this instance has been obtained it may be e.g. queried:

- whether it is a basic type (Listing B.3, line 3)

- whether it is a container-like type (Listing B.3, line 3)

- to return a list of fields of this class (Listing B.3, line 6-7)

- to return a list of methods of this class (Listing B.3, line 8-9)

- to return a list of base-classes of this class (Listing B.3, line 10-11)

Listing B.3: Using the Dictionary - Retrieving Basic Information

```
1    const Class* mc = Class::forName(''MyClass'');
2
3    bool isBasic = mc->isPrimitive();
4    bool isContainer = mc->isContainer();
5
6    const std::vector<const Field*> fields =
7         mc->fields(NOMODIFIER);
8    const std::vector<const Method*> methods =
9         mc->methods(NOMODIFIER);
10   const std::map<const Class*,std::pair<int,int(*)(void*)>>
11        baseClasses = mc->superClasses();
```

One may now interrogate fields of a class further. First one has to retrieve its meta representation (see Listing B.4, line 1), then it is possible e.g. to print the name and the name of the type of the field (see Listing B.4, lines 2-3). Given that there is an instance of the class, it is also possible to get or set values of fields (see Listing B.4, lines 5-7). The type of a field itself is again a class which may be queried further in the way described above.

Listing B.4: Using the Dictionary - Querying and Modifying Fields

```
1    const Field* mf = mc->field(''myVar'',NOMODIFIER);
2    std::cout << mf->name() << '' ''
3      << mf->type->name() << std::endl;
4
5    void* baseOfClass = new MyClass();
6    int val = mf->get(baseOfClass, int());
7    mf->set(baseOfClass, 4711);
```

Also the methods of a class can be queried further, e.g. the list of arguments or the return value which are again meta classes (see Listing B.5, lines 1-12). It is also possible to invoke a method and retrieve its return value (see Listing B.5, lines 14-15).

Listing B.5: Using the Dictionary - Querying and Calling Methods

```
1    const std::vector<const Method*>::const_iterator mIt;
2    const std::vector<const Class*>::const_iterator cIt;
3    for (mIt = methods.begin(); mIt != methods.end(); ++mIt) {
4      const std::vector<const Class*> args =
5          (*mIt)->argumentTypes();
6      std::cout << (*mIt)->returnType()->name() << '' ''
7        << (*mIt)->name() << ''('';
8      for (cIt = args.begin(); cIt != args.end(); ++cIt) {
9        std::cout << (*cIt)->name() << '' '';
10     }
11     std::cout << std::endl;
12   }
13
14   const Method* mm = mc->method(''myMethod'');
15   int ret = mm->invoke(baseOfClass, int());
```

B.3   Applications Using the Reflection

In the context of LHC several applications make already use of the dictionary system.

B.3.1   LCG Persistence Framework

The persistence project (POOL) [19] of the LHC Computing Grid (LCG) project uses the dictionary information to store and retrieve objects from persistent media. When writing, the Storage Service [21] of POOL will access the data members of the objects directly and write the values to the persistent medium. When reading back the dictionary information about the objects is used to create an instance of a transient class and then fill the object appropriately. POOL will also be used in Gaudi for persistence of objects.

B.3.2   Interactive Scripting Environment

Another possibility for using the reflection information is in interactive scripting environments. A python binding to the reflection packages was already developed which can be used to interactively work with objects from the python

interpreter. An example for usage of this package are the experiment software frameworks, where it may be used to work e.g. with events from the python prompt.

B.3.3 Event Data Browser

A prototype of an event data browser was also developed. This browser will display the transient event objects currently loaded into memory. In order to do that the browser needs to know the layout of objects in memory which it should display using the reflection.

B.3.4 Other Possible Applications

One may think of many other possible applications and language bindings to the reflection packages. A language binding that is currently thought of is the binding to Java which will then allow Java applications to use the reflection information loaded.

B.4 Outlook and Summary

After the implementation of the first approach which showed the technical feasibility a second approach was designed which resembles more the characteristics of C++. For this approach specific C++ concepts like pointers or references were taken into account. Figure B.2 shows the user-side API of the new Reflection model. What was the class type in the previous model is now split into more different types. Some of these types also have an explicit scope (e.g. Class or Enum). The PropertyList for storing information outside the C++ standard was also retained in this model.

Figure B.2: Second Implementation of the Reflection Model
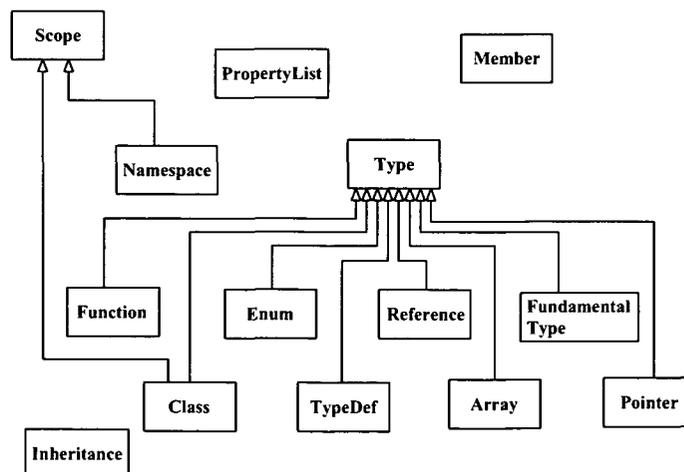
Currently there are attempts to standardise reflection in C++. This attempt is the extended type information (XTI). Unfortunately it seems to be too early to discuss the outcomes of this project as only some design proposals were available so far. XTI also tries to resemble the C++ ISO standard [52]. It is expected that it will take some more time until the XTI model is implemented, standardised and adopted by different compilers so it can be used in a broad range.

When XTI is standardised and adopted it is the plan to also use the XTI model from within LCG software. As the model also resembles the C++ standard closely the changeover to the new model is not expected to be difficult.

Listings

Tables

Figures

Glossary

ADAMO	A software framework for event data handling in HEP experiments.[1]
ALEPH	One of four LEP experiments[2]
ALICE	One of four LHC experiments[3]
ATLAS	One of four LHC experiments[4]
CERN	European Organisation for Nuclear Research located in Geneva, Switzerland and France.[5]
CMS	One of four LHC experiments[6]
CORBA	The *C*ommon *O*bject *R*equest *B*roker *A*rchitecture of the OMG group
DTD	*D*ocument *T*ype *D*efinition, a syntax definition language for XML documents
GEANT4	Software for simulation of the passage of particles through matter[7]
HEP	*H*igh *E*nergy *P*hysics
IDL	The *I*nterface *D*efinition *L*anguage, used in CORBA
OMG	The *O*bject *M*anagement *G*roup[8]
Perl	*P*ractical *E*xtraction and *R*eport *L*anguage, a scripting language[9]
Python	A scripting language[10]

---

[1] http://adamo.web.cern.ch/Adamo
[2] http://www.cern.ch/aleph
[3] http://alice.web.cern.ch/Alice/
[4] http://atlas.web.cern.ch/Atlas/Welcome.html
[5] http://www.cern.ch
[6] http://cmsinfo.cern.ch/Welcome.html
[7] http://geant4.web.cern.ch/geant4
[8] http://www.omg.org
[9] http://www.cpan.org
[10] http://www.python.org

Gaudi	The software framework for the LHCb and ATLAS experiments.[11]
LEP	The *L*arge *E*lectron *P*ositron collider. A particle accelerator in operation at CERN from 1989 to 2000.
LHC	The *L*arge *H*adron *C*ollider. A new particle accelerator at CERN, due to completion in 2007.
LHCb	One of four LHC experiments[12]
ROOT	An object oriented data analysis framework developed at CERN[13]
TES	*T*ransient *E*vent *S*tore
UML	The *U*nified *M*odelling *L*anguage[14]
XML	The *E*xtensible *M*arkup *L*anguage[15]
XML Schema	A syntax description language for XML[16]

---

[11] http://proj-gaudi.web.cern.ch/proj-gaudi
[12] http://lhcb.web.cern.ch/lhcb
[13] http://root.cern.ch/
[14] http://www.omg.org/uml
[15] http://www.w3.org/XML
[16] http://www.w3c.org/XML/Schema

Bibliography

[1] Adamo. http://adamo.web.cern.ch/Adamo.

[2] Christian Arnault. CMT : a software configuration management tool. In . International Conference on Computing in High Energy and Nuclear Physics, feb 2000.

[3] Alain Bazan et al. The athena data dictionary and description language. In H.S. Chen, editor, *Proceedings of CHEP 2001*, pages 494–497, Beijing, P.R. China, sep 2001. 2001 Conference for Computing in High Energy and Nuclear Physics (CHEP), Science Press New York Ltd. ISBN 1-880132-77-X.

[4] Alain Bazan et al. The athena data dictionary and description language. In *proceedings to be published in fall 2003*. 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP), mar 2003.

[5] Pavel Binko. C-plus-plus coding conventions. Technical Report LHCb-98-049, CERN, 1211 Geneva 23, Switzerland, may 1998.

[6] Wendy Boggs and Michael Boggs. *Mastering UML with Rational Rose*. Sybex, San Francisco, CA, 1999. ISBN 0-7821-2453-4.

[7] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1998. ISBN 0-201-57168-4.

[8] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Maler Eve. Extensible Markup Language (XML) 1.0 (second edition), W3C recommandation. http://www.w3.org/TR/2000/REC-xml-20001006, oct 2000.

[9] Rene Brun and Fons Rademakers. Root - an object oriented data analysis framework. In *Proceedings AIHENP'96 Workshop*, pages 81–86, Lausanne, CH, sep 1996. Nucl. Inst. & Meth. in Phys. Res.

[10] C++ Boost. http://www.boost.org.

[11] C++ Xerces. http://xml.apache.org/xerces-c/index.html.

[12] Olivier Callot. Revised C++ coding conventions. Technical Report LHCb-2001-054, CERN, 1211 Geneva 23, Switzerland, apr 2001.

[13] Marco Cattaneo et al. GAUDI - the software architecture and framework for building LHCb data processing applications. In . International Conference on Computing in High Energy and Nuclear Physics, feb 2000.

[14] S. Chiba. A metaobject protocol for C++. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, okt 1995.

[15] Tyng-Ruey Chuang, Y. S. Kuo, and Chien-Min Wang. Non-intrusive object introspection in C++: Architecture and application. In *Proceedings of the 20th International Conference on Software Engineering*, pages 312–321. IEEE Computer Society Press, 1998.

[16] CMT. http://www.cmtsite.org.

[17] Concurrent Versions System (cvs). http://www.cvshome.org.

[18] Doxygen. http://www.doxygen.org.

[19] Dirk Düllmann et al. POOL project overview. In *proceedings to be published in fall 2003*. 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP), mar 2003.

[20] Martin Fowler and Kendall Scott. *UML Distilled, Applying the Standard Object Modeling Language*, volume 2nd ed. Addison-Wesley, Reading, Ma., aug. 1997.

[21] Markus Frank et al. POOL storage, cache and conversion services. In *proceedings to be published in fall 2003*. 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP), mar 2003.

[22] DotGNU Project. http://www.dotgnu.org.

[23] Gccxml. http://www.gccxml.org.

[24] Geant4. http://geant4.web.cern.ch/geant4.

[25] Michel Goosens, Frank Mittelbach, and Alexander Samarin. *The Latex Companion*. Addison-Wesley, Boston, 15th edition, nov 1993. ISBN 0201541998.

[26] B. Gowing and V. Cahill. Meta-object protocols for C++: The iguana approach. In *Proceedings of Reflection '96*, pages 137–152, San Francisco, USA, apr 1996.

[27] The GNU scientific library (GSL). http://www.gnu.org/software/gsl.

[28] Herwig. http://hepwww.rl.ac.uk/theory/seymour/herwig.

[29] HyperText Markup Language (HTML). http://www.w3c.org/MarkUp.

[30] HyperText Transfer Protocol (HTTP). http://www.w3.org/Protocols.

[31] Java. http://java.sun.com.

[32] Java 1.4.1 java.lang.reflect package. http://java.sun.com/j2se/1.4.1/docs /api/java/lang/reflect/package-summary.html.

[33] The Java Tutorial; java.lang.reflect. http://java.sun.com/docs/books/tutor ial/reflect/index.html.

[34] The LHCb Collaboration. LHCB letter of intent, a dedicated LHC collider beauty experiment for precision measurements of CP-violation. Technical Report CERN/LHCC 95-5, LHCC/I 8, CERN, CH-1211 Geneva 23, aug 1995. http://cdsweb.cern.ch/search.py?recid=290868.

[35] Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell. Reification and reflection in C++: an operating systems perspective. Technical Report UIUCDCS-R-92-1736, ., Department of Computer Science, Urbana-Champaign, 1992.

[36] Pere Mato et al. Status of the GAUDI event-processing framework. In H.S. Chen, editor, *Proceedings of CHEP 2001*, pages 209–212, Beijing, P.R. China, sep 2001. 2001 Conference for Computing in High Energy and Nuclear Physics (CHEP), Science Press New York Ltd. ISBN 1-880132-77-X.

[37] Pere Mato et al. SEAL: Common core libraries and services for LHC applications. In *proceedings to be published in fall 2003*. 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP), mar 2003.

[38] J. Thomas Mowbray and C. Raphael Malveau. *CORBA Design Patterns*. Wiley Computer Publishing, New York, 1997. ISBN 0-471-15882-8.

[39] Markus Frank on behalf of the LHCb Gaudi team. Data persistency solution for LHCb. In . International Conference on Computing in High Energy and Nuclear Physics, feb 2000.

[40] Oracle. http://www.oracle.com.

[41] Perl. http://www.cpan.org.

[42] Pythia. http://www.thep.lu.se/ torbjorn/Pythia.html.

[43] Python. http://www.python.org.

[44] Z. Qian et al. Use of the ADAMO data management system within ALEPH. In *International Conference on Computing in High-energy Physics*, pages 283–298, Asilomar, CA, USA, feb 1987.

[45] Terry Quatrani. *Visual Modeling with Rational Rose and UML*. Addison-Wesley, Reading, MA, 1998. ISBN 0-201-31016-3.

[46] Rational Rose. http://www.rational.com.

[47] Stefan Roiser et al. Event data definition in LHCb. In *proceedings to be published in fall 2003*. 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP), mar 2003.

[48] Root. http://root.cern.ch/root.

[49] Doug Rosenberg and Kendall Scott. *Use Case Driven Object Modeling with UML*. 3rd. ser. Addison-Wesley, Reading, Ma., mar 1999.

[50] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991. ISBN 0-13-629841-9.

[51] International Standardization Organization (ISO), editor. *Information processing - Text and office systems - Standard Generalized Markup Language (SGML)*. International Organization for Standarization, Geneva, 1st ed. edition, oct. 1986. Ref.No. ISO 8879-1986(E), ISO 8879:1986/A1:1988(E), ISO 8879:1986/Cor.1:1996(E).

[52] International Standardization Organization (ISO), editor. *Programming languages – C++*. Americal National Standards Institute, New York, 1st ed. edition, sep 1998. Ref.No. ISO/IEC 14882:1998(E).

[53] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Massachusetts, 3rd. ed. edition, 1997. ISBN 0-201-88954-4.

[54] Extensible Markup Language (XML). http://www.w3c.org/XML.

Curriculum Vitae

Stefan Roiser

born Nov. 27, 1971 in Vienna
Austrian citizen, married

Education

2001-2003	Technical University of Vienna; Graduate Studies
1993-2001	Technical University of Vienna; Computer Science Programme; Graduation as Diplom-Ingenieur
1991-2000	Study programme at the Technical University of Vienna; Degree: 'Academically Certified Data Engineer'
1986-1991	Commercial College; Tulln; Lower Austria
1982-1986	High School; Vienna
1978-1982	Primary School; Vienna

Professional Experience

from 2003	Fellowship at CERN; Geneva
2001-2003	PhD-Studentship at CERN; Geneva
2000-2001	Research assistant with the Austrian Research Institute for Artificial Intelligence; Vienna
1999-2000	Civil service with the Red Cross; Tulln; Lower Austria
1997-2000	System administrator at the electronic data processing department of the Federation of Austrian Industry; Vienna
1991-2000	Administration of family owned business

Skills

Languages	German (mother tongue) English (fluent reading, speaking and writing skills) French (good reading and speaking, average writing skills)
Computing	Object Oriented Programming and Design, Data Modelling, Artificial Intelligence, Web Programming

List of Publications

- S. Roiser on behalf of the LCG SEAL Reflection Team. Reflection in C++. Technical Report LHCB-2003-116, CERN, CH-1211 Geneva 23, sep 2003.

- S.Roiser, M. Cattaneo, G. Corti, M. Frank, P. Mato Vila and S. Miksch. Event data definition in LHCb. In *proceedings to be published in fall 2003.* 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP), mar 2003.

- D. Düllmann on behalf of the POOL project. POOL project overview. In *proceedings to be published in fall 2003.* 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP), mar 2003.

- M. Frank, D. Düllmann, G. Govi, I. Papadopoulos and S. Roiser. POOL storage, cache and conversion services. In *proceedings to be published in fall 2003.* 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP), mar 2003.

- P. Mato, J. Generowicz, M. Marino, L. Moneta, S. Roiser and L. Tuura. SEAL: Common core libraries and services for LHC applications. In *proceedings to be published in fall 2003.* 2003 Conference for Computing in High Energy and Nuclear Physics (CHEP), mar 2003.

- P. Mato, M. Cattaneo, M. Frank, S. Ponce, F. Ranjard, S. Roiser, I. Belyaev, C. Arnault, P. Calafiura, C. Day, C. Legget, M. Marino, D. Quarrie and C. Tull. Status of the GAUDI event-processing framework. In H.S. Chen, editor, *Proceedings of CHEP 2001,* pages 209-212, Beijing, P.R. China, sep 2001. 2001 Conference for Computing in High Energy and Nuclear Physics (CHEP), Science Press New York Ltd. ISBN 1-880132-77-X.

- S. Roiser and G. Dorffner. An intelligent web-based tool for the virtual research enterprise. Technical Report TR-2001-07. Austrian Research Institute for Artificial Intelligence (ÖFAI), Freyung 6, A-1010 Vienna, Austria, 2001.
http://www.ai.univie.ac.at/cgi-bin/tr-online?number+2001-07

- S. Roiser. Scientific, technical and user centered aspects of a webbased intelligent search engine for a database of neural computation applications. Masters thesis, Technical University Vienna, Vienna, 2001. Title ID AC03177452.

Lebenslauf

Stefan Roiser

geb. 27. Nov. 1971 in Wien
Österreichischer Staatsbürger, verheiratet

Ausbildung

2001-2003	Technische Universität Wien; Doktoratsstudium der Technischen Wissenschaften
1993-2001	Technische Universität Wien; Diplomstudium Informatik; Graduiert als Diplom-Ingenieur
1991-2000	Technische Universität Wien; Kurzstudium Datentechnik; Titel "Akademisch geprüfter Datentechniker"
1986-1991	Handelsakademie des Fonds der Wiener Kaufmannschaft; Tulln; Niederösterreich
1982-1986	Unterstufe; Albertus Magnus Schule; Wien
1978-1982	Volksschule; Marianum; Wien

Berufserfahrung

ab 2003	Fellowship CERN; Genf
2001-2003	Doktoratsstudent CERN; Genf
2000-2001	Mitarbeiter am Österreichischen Forschungsinstitut für Artificial Intelligence; Wien
1999-2000	Zivildienst Rotes Kreuz; Tulln; Niederösterreich
1997-2000	System Administrator in der EDV Abteilung der Österreichischen Industriellenvereinigung; Wien
1991-2000	Administration in familieneigenem Unternehmen

Kenntnisse

Sprachen	Deutsch (Muttersprache) Englisch (sehr gute Lese-, Sprech- und Schreibkenntnisse) Französisch (gute Lese-, Sprech-, mittl. Schreibkenntnisse)
Computing	Objektorientierte Programmierung und Design, Datenmodellierung, Künstl. Intelligenz, Web Programmierung