Simulation and Reconstruction of Muons at the Munich ATLAS Cosmic Test Facility with Athena

Diplomarbeit an der
Fakultät für Physik der
Ludwig-Maximilians-Universität
München

Vorgelegt von Alexander Brandt

München, 4. November 2003

Erstgutachter: Frau Professor Dr. Dorothee Schaile
Zweitgutachter: Herr Professor Dr. Martin Fässler

Preface

The MuonCosmicTeststand (MCT) software project was launched in August 2002 by Philipp Schieferdecker. It is an approach for implementing a software package for a rather small experiment (the Munich ATLAS Cosmic Test Facility, section 1.2) using the instruments of a contemporary high-end experiment such as ATLAS (section 1.1). At present, the ATLAS collaboration is developing its software framework Athena, which is a state-of-the-art framework that benefits from many years of software experience in high energy physics and software development (chapter 2).

MCT does not aim to be the single authorized software solution for the Munich ATLAS Cosmic Test Facility; Oliver Kortner and Felix Rauscher already developed a stand-alone, framework independent software solution that performs all tasks necessary for the fulfillment of the design goals of the test stand. Motivations for MCT are

- learning about Athena considering a small experiment as an example and thus being confronted with a great variety of data processing tasks such as simulation, calibration and reconstruction,
- benefiting from useful Athena features: several tasks in high energy physics computing are already accomplished in Athena in a very general way and can easily be adopted (e. g. geometry- and event displays, figures 1.3 and 3.3),
- collaborating with other groups that operate muon test stands and easily exchanging code,
- verifying the results produced by the existing stand-alone software solution for the test stand and
- supporting the Athena developers by permanently giving feedback.

Especially the last point should not be underestimated, since only a very few data-taking experiments using Athena exist that rely on a broad variety of Athena components.

At the time this thesis started (November 2002), MCT was aleady capable of performing parts of the simulation using Geant4 (section 3.1). During its development, MCT profited from the experiences made with Oliver Kortner's and Felix Rauscher's program.

I would like to thank my Professor Dorothee Schaile and my supervisor Günter Duckeck for their steady support on my diploma thesis.

I owe thanks (and beer) to Philipp Schieferdeckers and Felix Rauschers endurance and expertise in answering my questions on both software development and test stand matters. Philipp gave me a perfect start-up in the beginning of my thesis work and Felix helped me with essential last-minute contributions.

Special thanks go to my parents, to whom this work is dedicated.

Insbesondere möchte ich meinen Eltern danken, denen ich diese Arbeit widme.

Contents

Chapter 1

Introduction

Investigating small structures requires big machines. The attribute 'big' refers to every aspect of a modern experiment in high energy physics. Not only are the operating machines big and of course expensive. Integrating the number of people involved over time yields a number that exceeds earlier experiments by far.[1]

LHC, the Large Hadron Collider, is currently being built into the ring tunnel facilities of its predecessor LEP (Large Electron-Positron Collider) and is supposed to be commissioned in 2007. LHC is facing great scientific achievements of LEP, some of which are measurements of W- and Z-Bosons, the exchange bosons of weak interaction and general precise Standard Model tests. Although the LEP concept was challenging for its developers from the beginning, the LHC storage ring is even more challenging for present scientists. The two concepts differ not only in kinetic energy transferred to the particles. LEP used to accelerate electrons or positrons up to about 100 GeV while Protons at LHC are supposed to reach 7 TeV. Interactions between two protons at LHC will differ significantly from those at LEP. To our present understanding, leptons, such as electrons and positrons, are point-like particles. Interactions, i. e. head on collisions between two particles are therefore much simpler than in the case of LHC, where the involved particles have a complex structure. The center of mass energy of a collision between constituents of protons at LHC is expected to be significantly lower than two times the energy of an accelerated proton - even worse, the energy is not exactly known.

LHC, as LEPs predecessor, is exposed to high expectations, and the physics it might allow us to study is marvelous from the current point of view. It will recreate conditions which prevailed in the universe 10^{-12} seconds after the Big Bang. With its four experiments, the LHC will become a touchstone for the standard model (e. g. Higgs mechanism) and theories beyond, such as Supersymmetry.

1.1 ATLAS Detector

ATLAS (A Toroidal LHC ApparatuS) is, besides CMS (Compact Muon Solenoid), one of the two general purpose experiments operating at the LHC. Its geometry is the ordinary and well approved onion skin configuration, consisting of (from inside to outside) tracker, calorimeter and muon system (1.1).

It is designed to exploit a wide range of the physics opportunities offered by LHC, among

[1] Due to the large number of participating scientists, the per capita cost of an experiment is comparable to or even lower than that of other disciplines in modern physics.

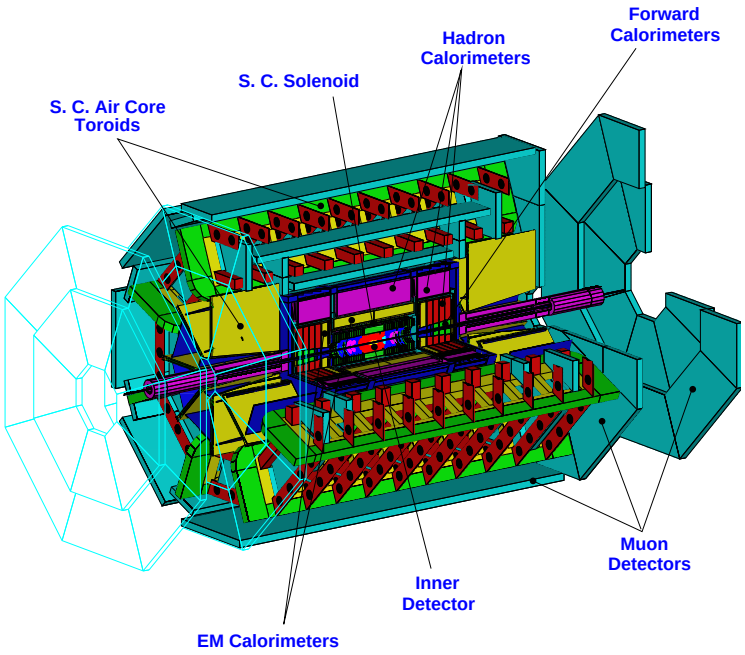

Figure 1.1: Outline of the ATLAS detector (from inside to outside): tracker, calorimeter and muon system. The beam pipe is located along the symmetry axis, with the interaction point in the center of the setup.

which the reason for spontaneous symmetry breaking in electroweak theory appears as well as the search for supersymmetry, detailed studies of heavy fermions and fermion compositeness. The search for the Higgs Boson is one primary goal of the ATLAS detector and widely dictates its design.

Discovery of new physics usually needs a high statistic, whereas ATLAS is designed to operate at high luminosity ($10^{34}\,\mathrm{cm}^{-1}\mathrm{s}^{-1}$), picking up as much information as possible.

Figure 1.1 shows an outline of the detector. The **inner detector** consists of three sub detectors covering the range $|\eta| \leq 2.5$[2]. A central solenoid magnet provides a magnetic field of 2 T and thus allows identification of charged particles and momentum determination.

The pixel detector consists of layers of semiconductor sensors with a very high granularity ("pixels" of size $50\,\mu\mathrm{m} \times 300\,\mu\mathrm{m}$). It is designed to deliver precise measurements very close to the interaction point of the beams and to determine the exact interaction point as well as displaced secondary vertexes of the produced particles.

Double layers of silicon strips are the sensitive material of the semiconductor trackers (SCT), aligned vertically to the beam direction. The two layers in one double layer are slightly tilt with respect to each other (40 mrad). Pixel and SCT sub detectors together build the precision tracker.

The third inner detector subsystem is the transition radiation tracker (TRT), consisting of ~ 36 layers of 4 mm diameter straw tubes. Its main purpose is to identify and measure charged

[2]The pseudo rapidity η is a function of the energy and the transverse momentum of a particle. In the relativistic limes, it changes additive under Lorentz transformations. Therefore, its distribution is independent of the reference framework.

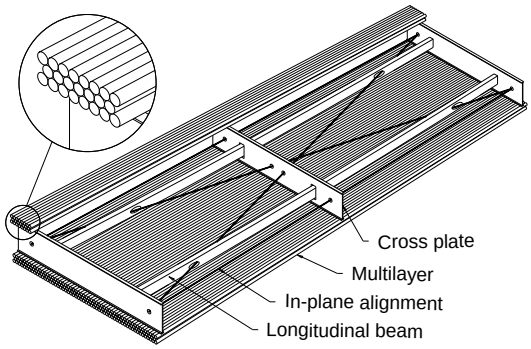

Figure 1.2: Outline of an ATLAS MDT chamber.

particles.

ATLAS **calorimeters** are separated into an electromagnetic (EM) and a hadronic part. The EM part is built by an accordion-shaped liquid argon (LAr) calorimeter, with lead absorber plates alternating with copper electrodes on Kapton carriers for the barrel and the end cap. The hadronic part is built of scintillating tiles in the barrel region and an LAr calorimeter with copper or tungsten absorbers in the end caps.

Overall dimensions of the detector are defined by the **muon spectrometer**. Outer chambers of the barrel are at a radius of about 11m. The muon system consists of superconducting barrel and end cap toroids which establish a magnetic field of 0.8 T to 1 T. An air core system reduces dead material in the detector and therefore the effect of multiple scattering.

The intended purpose of the muon system is twofold. One aim is to provide a fast trigger system. This is done by resistive plate chambers (RPC) in the barrel of the detector and thin gap chambers (TGC) at the end-caps. These chambers are very fast ($t < 25\,\mathrm{ns}$) but have a coarse spacial resolution.

Monitored drift tubes (MDT) of 3 cm diameter are grouped together in multilayers, two of them build up one chamber. They measure the coordinates of the muons in the bending plane and serve precise momentum information. MDTs form the major part of the spectrometer. They have a rather long drift time ($t \sim 500\,\mathrm{ns}$) but give a good spacial resolution. Figure 1.2 shows a schematic view of an ATLAS MDT chamber. The drift tubes are made of aluminum and are operated at pressure of 3 bar and voltage between its W-Re anode wire and the aluminum wall of 3080 V. The drift gas is a $93:7$ Ar : CO_2 gas mixture.

With its 1194 chambers, the whole muon system covers an area of more than $5300\,\mathrm{m}^2$.

1.2 Munich Cosmic Ray Test Facility

The University of Munich (LMU) in collaboration with the Max Planck Institute for Physics (MPI) is responsible for the construction of MDT chambers of type BOS (barrel outer small). These chambers consist of two multi layers of drift tubes, each layer built of three single layers. This chamber type is the second largest used at the ATLAS spectrometer and has a size of $3.9\,\mathrm{m} \times 2.2\,\mathrm{m} \times 0.5\,\mathrm{m}$.

It is the main purpose of the Munich Cosmic Ray Test Facility to test the chambers (including on-board electronics) and to produce maps of the chamber wire positions with an accuracy of $20\,\mu\mathrm{m}$. Furthermore, the test stand allows to gain experience with the operation and calibration

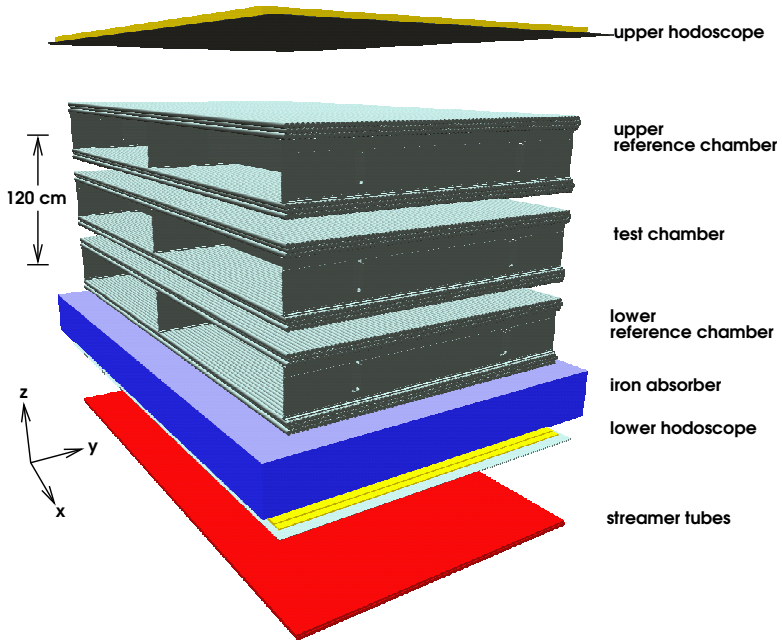

Figure 1.3: Image of the test stand generated with Athena tools including a size comparison and a coordinate system definition. The scintillator bars of the hodoscope are situated perpendicular to the drift tubes while the streamer tubes are parallel.

of MDT chambers, both on the hardware as on the software side.

Figure 1.3 shows a schematic view of the test stand setup. This image was generated from the geometry description inside the Athena framework. The setup consists of three MDT chambers, an iron absorber (underneath the MDT chambers), one scintillator layer above the chambers and a double layer below the iron absorber. At the very bottom is a layer of streamer tubes. Small deviations of the nominal chamber positions as well as small transformations of the chamber itself are detected with optical and capacitive alignment systems (RasNiK, [1]).

A coinciding signal of all the three scintillator layers causes the setup to record data, the scintillator layers act as a trigger. Segmentation of the scintillator layers limits the inclination of the muons in a plane perpendicular to the single scintillator bars.

A 34 cm iron absorber located between the scintillator layers ensures that only muons with a momentum greater than 600 MeV are triggered. In addition to that, a layer of streamer tubes situated below the lower hodoscope provides information about the multiple scattering angle inside the iron absorber. It is used as an estimate for the muon energy

Two of the MDT chambers act as reference chambers. Their wire position has been measured with an accuracy of 2 μm in an X-ray tomograph situated at CERN.

All chambers assembled by MPI and LMU will undergo a series test, presumably until the end of 2004.

Chapter 2

Athena Framework

Athena is the fully object-oriented framework for the integration of all data processing steps concerning the ATLAS experiment. It is currently being developed by the ATLAS collaboration and is designed to cover simulation, calibration, reconstruction and physics analysis of a high energy physics experiment. It is written entirely in C++.

Athena itself is based on the experiment-independent Gaudi framework [2], initially developed for the LHCb experiment. The development of Gaudi is much inspired by the experience that high energy physics experiments are expected to run for many years and it is very likely that underlying technologies, such as storage technologies, may change. Therefore Gaudi avoids any direct dependency on external technologies and introduces a layer of abstraction between them and Gaudi applications. Concrete external software is then integrated into a Gaudi application by the joboptions mechanism, described later in this chapter.

In the following sections, I will not explicitly distinguish between Gaudi and Athena contribution to the ATLAS software framework and generally write Athena, even if 'Gaudi/Athena' or 'Gaudi' would be more appropriate.

The Athena framework offers concepts, which allow a huge number of developers to concertedly work on a large and complex software project.[1] Intuitive communication mechanisms lead to humanly readable code and allow a consistent treatment of data from different stages of data processing or even different subsystems of the detector. Athena makes heavy use of the object-oriented features of C++ and follows modern guides for object oriented-software design (as in [3]). With its blackboard-like communication mechanism and its concept of separated, sequentially executed algorithms (see below), a maximum decoupling of data and algorithms is achieved. This makes Athena code flexible and easily to reuse or to extend.

Athena's most important concepts will be explained in this chapter. Some certain Athena packages become of special interest later for the MCT project and shall be introduced here, too. A more detailed and technical description of the Gaudi framework is found in the *Gaudi Developers Guide*, available from [2].

Conventions

The following, programming-related speech conventions will be used throughout this chapter and the rest of this document. A little experience in object-oriented programming on part of

[1] To give 'huge' and 'large' a number: the release 6.3.0 of the ATLAS software package occupied 4.8 gigabyte of disc space, excluding the underlying Gaudi framework. The number of subscribers of the most general ATLAS software mailing list ('atlas-sw') was 899 in October 2003.

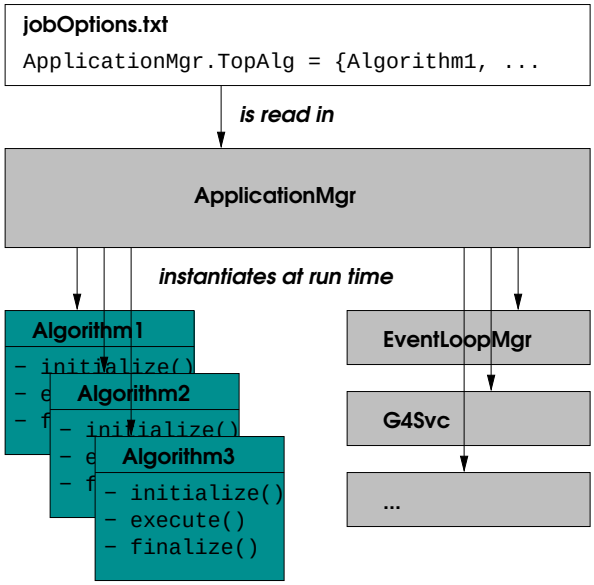

Figure 2.1: Central role of the application manager: user code is requested in joboptions files. The application manager instantiates the objects *at run time*.

the reader is not mandatory but facilitates reading.

In some places, the name of a class is used synonymously for an instance of this class, especially if only one instance of a class exists within a certain Athena job (e. g. **StoreGate** as described below).

Furthermore, the difference between an **Algorithm** and an algorithm is that the former is a class derived from the common Athena base class **Algorithm** (or an instance of a class derived from this base class), whereas the latter corresponds to the commonly accepted definition of an algorithm. Generally speaking, **bold-faced** words indicate class names.

Citations from the source code are set in the `typescript` font family. A *(...)* inside a source code citation indicates that small parts of the code have been omitted in order to increase readability.

I will adopt the commonly accepted "is-a" speech when referring to classes and their objects. Saying "ObjectB" is-a **ClassA** means: "ObjectB" is an instance of a class, let us say **ClassB**, that is derived from **ClassA**. In other words: every method defined in **ClassA** is served from "ObjectB". Or: every piece of code that can handle objects instantiated from **ClassA** will also be able to handle objects instantiated from **ClassB**, since **ClassB** inherits all properties from **ClassA**.

2.1 Joboptions Mechanism

An Athena application is concentrated around the 'athena.exe' executable. This executable is framework specific and does not contain any user code.

Figure 2.1 demonstrates the central role of the application manager, which is the only object contained and instantiated in the 'athena.exe' executable. Once initialized, it reads a joboptions

file[2], which is a simple, static text file. The joboptions define which libraries to load and which objects to instantiate for the application manager.

As figure 2.1 suggests, an Athena application is far from being a static application. An Athena job with a minimal joboptions text file is unable to produce anything reasonable.

The ability to dynamically load libraries is remarkable and differs from usual Linux/Unix library sharing: the libraries are loaded *at run time*. The libraries contain class implementations which represent the functional code of an Athena application. These classes have to be derived from common Athena base classes (the most important are **Algorithm**s, **Service**s and **Tool**s) some of which will be introduced later. The objects, instantiated by the application manager following directives in the joboptions file, can be configured via the same joboptions file. Properties of an object are usually assigned in an object oriented style, such as `ObjectName.ObjectProperty = ''value''`. The configuration of the application manager object does not differ from the configuration of an arbitrary object instantiated in an Athena application: `ApplicationMgr.TopAlg=''Algorithm1/MyAlg''` tells the application manager object to instantiate an object of class **Algorithm1** and assign it the name *'MyAlg'*. This object can later be configured via its name *'MyAlg'*.

As mentioned above, Gaudi aims to be able to handle changes in underlying technologies, such as storage technologies. In fact, when developing **Algorithm**s (or other components), the user should not care about the concrete underlying technology. He interacts with an abstract Gaudi interface which hides the concrete technology. The choice of technology is finally done in the joboptions file and can be easily changed.

A more advanced feature of the Athena framework is the possibility to instantiate multiple instances of one class, giving it different names. This way, every instance of this class can be configured individually.

2.2 Athena Components

In order to be able to instantiate objects at run time, the classes need to inherit from certain Athena base classes. They differ in their purpose and are described in the following subsections.

2.2.1 Athena Algorithms

Due to the nature of high energy physics experiments, data processing is subordinated to an event-by-event cycle.[3] Every step to be done in an Athena application (such as calculating muon tracks out of a collection of drift circles), will be the same for every event. Objects performing concrete calculations therefore have to be instances of classes that derive from a common base class (**Algorithm**) and overload three methods:

- `initialize()` To be done at the beginning of a job, such as configuration following the definitions in a joboptions file.
- `execute()` Perform reasonable calculations for each event.
- `finalize()` To be done at the end of a job, such as writing out job statistics.

[2]Instead of static text files, Athena has the ability to interact with the user via an interactive Python interface. This feature is supposed to become common in the future for Athena applications.

[3]This is the reason why high energy physics applications are usually relatively easy to parallelize.

Figure 2.1 gives an idea of how **Algorithm**s are instantiated. The joboptions define which **Algorithms** shall be instantiated and in which order their `execute()`-method shall be invoked. The presence of **Algorithm**s in Athena emphasizes its properties of being a *framework*: The user overloads classes, whose methods are invoked (only) by the framework itself.

2.2.2 Athena Tools

Instances of classes that inherit from the Athena **Tool** base class are helper objects, that can be requested by every other Athena component, e. g. Athena **Algorithm**s. Their purpose is to implement small algorithms that can be used at several places within an Athena job. This concept has two major advantages: first, frequently used algorithms need to be implemented only once and can be used from everywhere inside the Athena framework by simply requesting a certain **Tool** from the **ToolSvc** (tool service, see below). Second, since the **Tool** is requested by submitting a simple string to the **ToolSvc**, the concrete **Tool** requested can be steered via joboptions (this, of course, implies a common interface for **Tool**s that serve identical exercises).

The flexibility of choosing concrete **Tool**s via joboptions is exploited at different places in the MuonCT package. The MuonCT reconstruction **Algorithm** calls the concrete track fit algorithm via an abstract **Tool** interface (more precisely: an **ITool** interface). The concrete reconstruction **Tool** (and therefore the algorithm used) is defined via joboptions!

As **Algorithm**s, **Tool**s can implement `initialize()`- and `finalize()`-methods.

It is worth mentioning that components requesting **Tool**s have the choice of requesting one common instance of a **Tool** or an individual instance, the latter can be configured individually by the requesting component.

2.2.3 Athena Services

Services are objects that usually exist in one single instance and can be used from everywhere inside the Athena framework. The tool service **ToolSvc** has been mentioned above as an example for a service: every Athena component requesting a **Tool** needs to interact with the single **ToolSvc** instance.

2.2.4 Athena Converters

The concept of **Converter**s is central to a technology independent architecture. As mentioned above, one main goal of the Athena framework is to be independent of future technology changes. This is realized by adding additional layers of abstraction. In case of storage technology, this abstraction layer is provided by **Converter**s (and corresponding **ConversionSvce**s, conversion services). This leads to a strict distinction between the *transient representation* (i. e. in memory) of a data object and its corresponding *persistent representation* (e. g. on hard disk or tape). If the underlying technology changes, only converters have to be updated, not the operational code.

ConversionSvcs (one for each storage technology) and **Converter**s (one for each storable class *and* storage technology) work together hand in hand. The **Converters** are triggered via their `createObj()`- or `createRep()`-methods. The former creates a transient representation by interpreting external data while the latter persistifies an object. The argument list of these methods contains all information necessary. Communication between converters and their conversion services can be manifold and is left to the code designer. In one case described below (conditions

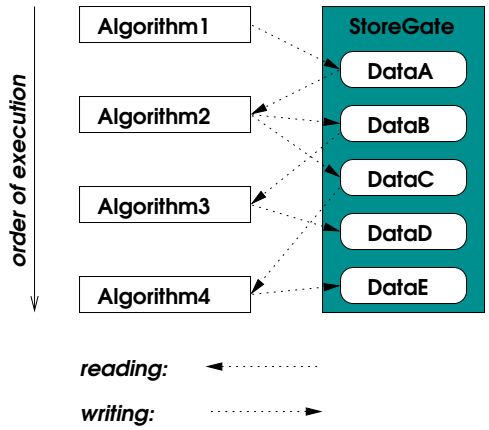

Figure 2.2: Illustration of a possible data flow within a chain of Athena **Algorithm**s.

database), the **ConversionSvc** provides filestream objects for either reading or writing to its converters.

2.3 StoreGate: Communication between Components

Athena jobs can be composed from many dozens of **Algorithm**s or even more. This gives rise to the question of communication between different **Algorithm**s. Athena developers have chosen a blackboard technology, named **StoreGate**. The idea is as follows: components (such as **Algorithm**s) of an Athena job have access to a store and are able to *write* and *read* to and from that store, but are unable to *modify* objects once they are inserted in that store. The objects stored in **StoreGate** are only identified by their type[4] and (optionally) a `std::string` key.

Figure 2.2 shows a possible data flow for a chain of **Algorithm**s. The `execute()`-method of each **Algorithm** is called in a defined order. In that method, data is read and/or produced.

With the introduction of **StoreGate**, a crucial aspect of C++ is addressed: the problem of ownership. In C++, unlike languages as e. g. Java, objects survive (and consume memory) until they are deleted explicitly. **StoreGate** takes over ownership of an object once it has been placed in **StoreGate**. At the end of processing one event, **StoreGate** (also referred to as event store) deletes all its contents.

Of course, there are classes of data which shall not be deleted after an event has been processed. For these types, e. g. detector description data, Athena offers further stores which differ from **StoreGate** only in their lifetime policy. The **DetectorStore** for example contains the geometry description of an experiment and survives throughout a complete Athena job.

2.4 Special Athena Packages

This section describes Athena packages of certain interest to the MCT project. They are not framework specific and could (theoretically) be easily exchanged, even though they make heavy use of the facilities described above.

[4]Identification by type is implemented via a unique number identifying each storable class.

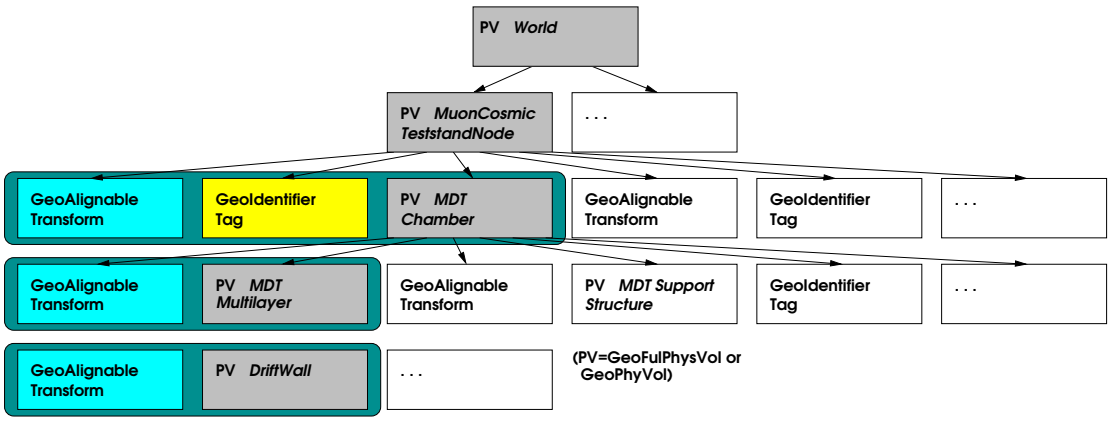

Figure 2.3: Cutout of the MCT GeoModel description. The grouping (green box) is realized *only* by the fact that the tree is interpreted sequence-sensitively. A detailed explanation is given in the text.

2.4.1 Geometry Model

Most of the jobs that can be processed with Athena need information about the detector geometry including information about used materials and where they are placed. This includes 'sensitive materials' like detectors as well as 'dead materials' such as absorbers.

GeoModel is the name of a concept currently used in the Athena framework for transient storage of geometry information. GeoModel neither defines how this information is collected nor how it is stored persistently. It is just an in-memory representation of the full detector setup. [4] gives a more detailed description on how the GeoModel is to be used.

The main design goal of GeoModel was to provide a slim representation of the detector setup. It basically consists of an *ordered tree* with **GeoGraphNode**s at each node. In class **GeoGraphNode**, only the basic properties are defined that are necessary to integrate an object into the GeoModel tree.

Figure 2.3 gives a sketch view of how a GeoModel tree is designed considering the MCT GeoModel tree as an example. Each box seen represents an object derived from **GeoGraphNode**. Concrete implementations of this class are in this example: **GeoPhysVol** or **GeoFulPhysVol** (abbreviated with PV in the figure), **GeoAlignableTransform** and **GeoIdentifierTag**.

GeoPhysVols or **GeoFulPhysVol**s represent the shape and the material of a given object, such as a MDT Multilayer. (The shape- and material-representation is actually realized by a pointer to an object of class **GeoLogVol**, which contains this information. This will become a crucial point when it comes to translation between different types of geometry descriptions as seen later.) The two classes differ in their ability to cache position information: the **GeoFulPhysVol** holds a copy of its global position while **GeoPhysVol** does not. The position information is likely to change in an alignable geometry and has to be updated continuously by certain mechanisms.

GeoAlignableTransforms[5] or **GeoTransforms** are objects which represent the local position of a physical volume (i. e. a **GeoPhysVol** or **GeoFullPhysVol**) inside another, given the concrete translation and rotation. It is important to note that only the local position is

[5]Later, **IOVGeoAlignableTransform** will be introduced as a transformation that is automatically aligned by the **IOVSvc**.

given. In order to determine the absolute position of an object (e. g. a DriftWall in figure 2.3), it is necessary to traverse the whole tree from the regarding node up to the root node and to concatenate all transformations seen on that path.

Since some volumes need a unique identification, GeoModel provides a **GeoIdentifierTag**. It serves information about an **Identifier** associated to a certain geometry object. **Identifier**s are tuples of numbers following a certain convention that contain identification of subsystem type and readout channel. In the figure, only a **GeoFulPhysVol** representing a MDT chamber has such a tag.

In figure 2.3, one sees that certain objects are grouped, such as the **GeoFullPhysVol** MDT chamber together with its transformation and an identifier tag. This is done only by the order in which these objects are bound to their parent node, which in this case is a **GeoFullPhysVol** named 'MuonCosmicTeststandNode'. When building a GeoModel tree, every **GeoGraphNode** inserted into a tree has to be interpreted as being associated with the next **GeoPhysVol** or **GeoFullPhysVol** being inserted. This makes the GeoModel tree an *ordered tree*.

GeoModel trees are not intended for fast readout. Due to its structure, it is time expensive to access certain parts of the detector. The recommended way to perform several actions that need to traverse through the geometry is via so-called **GeoNodeAction**s. An instance of **GeoNodeAction** can be regarded as a small object, that is handed from node to node automatically (either upwards or downwards the tree "from where it is inserted") and travels through the tree. In doing so, it can perform several tasks, such as collecting information about the geometry or invalidating cached position information of **GeoFulPhysVol**s.

Therefore, the MCT project has implemented a **MuonCTDetDescrMgr**, a detector description manager. Its duty is to traverse through the whole GeoModel tree and collect all information needed and offer it to the user (i. e. **Algorithm**s) for easy access.[6]

The transient representation of the GeoModel is stored in the detector store.

2.4.2 Interval of Validity Service

Since the measurements performed by the ATLAS subdetectors are extremely precise, they are sensitive to minimal fluctuations to outer conditions such as temperature. These can, for example, lead to small changes in the detector geometry on the micrometer scale or variations of the spectra of drift tubes.

In the case of the Munich Cosmic Ray Measurement Facility, realignment becomes necessary due to an additional reason: in the series test, the muon chambers are exchanged weekly and need to be realigned individually. Since events from different test chambers carry different run numbers, this can be fully handled by the **IOVSvc** (interval of validity service).

The mechanism described below allows users to access time varying data from the transient store without being aware of the data being updated automatically. The process is completely transparent to the user except for the fact that the user has to register data with the **IOVSvc** explicitly.[7]

Conditions data can be assumed to be constant during the time an event takes. The **IOVSvc** defines *intervals of validity* for user-defined types of conditions data in terms of run and event numbers (represented by two pairs of run and event number, a *from*- and a *till*-pair). The

[6] This will become a point of interest later, when the GeoModel is aligned automatically by the IOVSvc: the **MuonCTDetDescrMgr** need to be informed about all changes in geometry.

[7] It is currently being discussed to extend the IOV mechanism such that registering is no longer necessary.

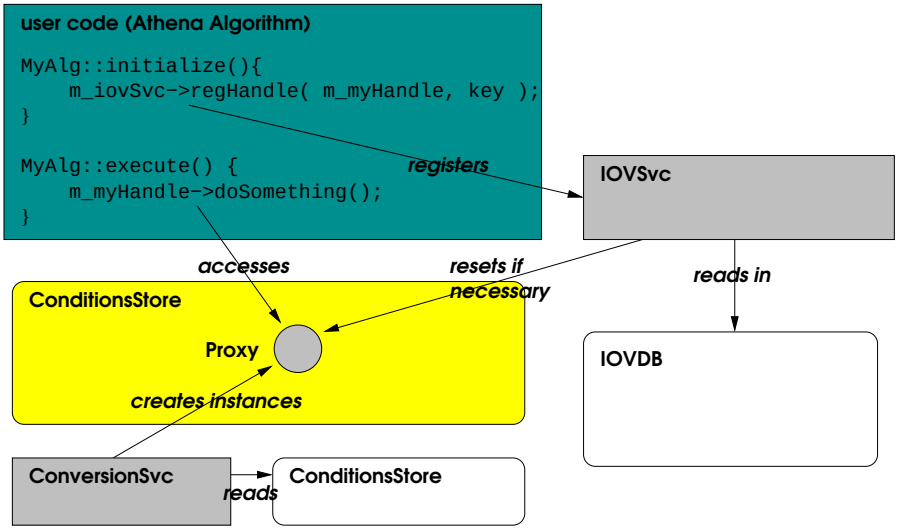

Figure 2.4: How **IOVSvc**-based access works: slanted words indicate verbs. The procedure seen in this figure is described in the text. Important: the two white stores (IOV-database and conditions store) are external stores and therefore easily exchangeable in the Athena sense.

IOVSvc allows the user to register **DataHandles**[8] for being updated when an event has run out of an interval of validity.

When a **DataHandle** named `myDataHandle` is registered with the **IOVSvc** via a

```
m_iovSvc->regHandle( myDataHandle, key );
```

call, the **IOVSvc** stores a so-called proxy in **StoreGate**. Let's assume, `myDataHandle` acts as a pointer to objects of type **myData**. The proxy contains no data but a *reference* to the real data. This reference is interpreted by converters. When the user dereferences `myDataHandle`, **StoreGate** is involved and notices that the proxy is empty and hands over all the information that is stored in the proxy to the converter, which itself creates an instance of the desired object by evaluating external data stores. From now, this instance remains in memory and is dereferenced directly via `myDataHandle`.

Since there can be more than one object of type **myData**, these objects are uniquely identified by a `std::string` named `key` in this example. (E. g. multiple instances of chamber positioning information, one for each chamber, have a key that uniquely identifies their chamber.)

Once an object that is accessed via a registered **DataHandle**, becomes invalid (e. g. the position of a drift chamber because the chamber has moved), the **IOVSvc** resets the **DataHandle** and fills the proxy with a new reference to the valid data object (valid means appropriate considering the current (run, event)-pair). As a result, the next dereferencing of the **DataHandle** again lets the converter read the conditions database and create an instance of the corresponding object. Figure 2.4 gives an impression of how this procedure works.

The **IOVSvc** retrieves information about references to valid conditions database entries by interpreting the IOV database. It was already mentioned that Athena is designed to be

[8] **DataHandle**s are dereferenced like pointers and the difference between C++-pointers and **DataHandle**s is negligible in this context.

independent of future technology changes. Therefore, the **IOVSvc** does not access the IOVDB (IOV database) directly. Instead, a layer of abstraction, namely the **IIOVDBSvc**, is inserted in between.

In principle, it is possible to avoid the conditions database completely. Instead of interpreting the tag provided by the IOVDB as a reference to an entry in the conditions database, the converter could interpret the tag (a `std::string`) as the data itself. This is done by streaming arbitrary complex data in a sequential "blob".

The two-database-design explained above has several advantages. It allows normal users to create several, even overlapping sets of calibration data and to store them in the conditions database. Later, a "group of experts" is able to inspect this data and store references to approved data in the IOV database. The latter might even be possible through an intuitive web interface.

In addition to that, since **IIOVDBSvc**s and converters are easily exchangeable via joboptions, the end user might override these authoritative decisions in his private development area and play with alternative calibrations.

Updating data objects does not exhaust the power of the **IOVSvc**. Later, in chapter 4.4, we will see that it becomes necessary to take certain actions once an object changes in memory. For this reason, the **IOVSvc** allows to register callback functions with data objects. They are called at the beginning of an event when an object is updated because an interval of validity becomes invalid. Even more, it is possible to register callback functions with other, already registered, callback functions. Whenever the first function is invoked by the **IOVSvc**, the other one is triggered, too.

A more detailed description of the IOV service can be found on the author's web page [5].

Chapter 3

Simulation and Reconstruction of Muons with the MCT Software Package

The MCT or MuonCosmicTeststand project consists of several logical units (such as Simulation, Reconstruction, Calibration) which themselves consist of several implementations of Athena components (**Algorithms**, **Service**s, etc.). For example, MuonCTGeoModel contains the complete geometry description of the Munich Cosmic Ray Measurement Facility. MuonCTG4Sim implements the simulation of the detector based on the Geant4 simulation toolkit, which is used inside Athena. MuonCTReco and MuonCTRecoUtils contain algorithms which evaluate test stand data and reconstruct muon trajectories. Other packages implement the Athena converters necessary to store or read test stand specific data. MuonCTCalibAlgs holds **Algorithm**s that determine calibration parameters of the test stand in an Athena job.

Instead of explaining each of these sub packages step by step, I will give an example of a simulation and a reconstruction job that can be processed by composing parts of them via a joboptions file. A short discussion of all Athena components and data classes in MCT can be found in appendix B.

3.1 Detailed Description of a Simulation Job

Simulation is an important part of physical unterstanding. In a high energy experiment, the measurement is influenced by many parameters, such as the amount of dead material, its geometry as well as the interaction of physical processes such as avalanche formation in gaseous detectors. In many cases, simulation, rather then analytic calculation, is the only way to gain predictions for an experiment.

A typical simulation job ends up with a data file containing digits, representing signals as they could have come from the experiment itself. Each of the following subsections represent **Algorithm**s coming from certain sub packages of the MCT project. They are executed in the order that these subsections appear. When all `execute()` methods are processed as described below, the event store contains several collections of digits for all different types of detectors in the cosmic test stand. From that point, it is possible to have a number of subsequent algorithms that further process the data, such as a reconstruction or calibration job. Saving of digits is realized by means of converters as discussed above. At the end of an event, the content of the

event store is purged, but before this is done, converters may create persistent representations of the data.

3.1.1 SingleParticleGun: Generation of Muons

Generation of particles for the Munich Cosmic Ray Measurement Facility is a rather simple task compared to complex experiments such as ATLAS. The trigger system of the real test stand provides that only one cosmic event at the same time is recorded. Therefore, in its `execute()`-method the particle generator simply dices energy and momentum within given bounds (defined in joboptions) and produces an object representing a muon with these properties and stores it in the event store (**StoreGate**) in the common HepMC-format [6]. **SingleParticleGun** is not part of the MCT package, it resides in an Athena package, GeneratorModules, together with other, more complex, particle generators.[1]

3.1.2 MuonCTG4Sim: Tracking of the Particle

The next **Algorithm** in the chain differs from other **Algorithm**s since it does *nothing* in its `execute()` method.

Athena uses Geant4 (also referred to as G4) as a simulation engine. Geant4 provides its own framework for stand-alone applications. Using one framework inside another leads to a number of unwanted effects and has to be avoided somehow. In Athena, the only access to Geant4 from within the Athena framework is by means of the **G4Svc** (Geant4 service), which is a service in the Athena sense as discussed above. The only task for **MuonCTG4Sim** is to define the simulation boundary conditions (such as geometry and type of physics desired within the simulation) in its `initialize()` method by interacting with the **G4Svc**. Everything else (namely the tracking of particles through the detector) is left to **G4Svc** and the Geant4 framework. The starting point for tracking is the HepMC object that is found in the event store.

The MuonCTG4Sim sub package itself contains more than just one **Algorithm**. It also holds algorithms that translate the GeoModel geometry description into a Geant4 framework specific geometry description. G4 sensitive detectors (**G4VSensitiveDetector**s), also implemented in this sub package, are algorithms attached to certain volumes of the detector. Whenever the tracking engine tracks a particle into a volume that has a sensitive detector attached, this algorithm is able to calculate hits and store them in the event store. In case of the sensitive detectors assigned to drift tubes, the hit objects stored mainly consist of the drift radii seen by single tubes.

Geometry conversion and the matter of sensitive detectors in the Athena framework are addressed in subsequent chapters.

3.1.3 ScintiPreDigitizer, TriggerSim: Trigger Simulation

In the real test stand, the scintillator hodoscope acts as a trigger for the whole setup. Events are only stored if they conform a certain signature, one of which is the demand of a restricted range of the angle of incidence.

ScintiPreDigitizer evaluates a small fraction of the hits stored in the event store and calculates corresponding digits as seen by the scintillator system of the test stand. The next algorithm in the chain, **TriggerSim**, again picks up the information from **ScintiPreDigitizer**

[1] The **SingleParticleGun** is obsolete now and has been replaced by a more flexible generator.

and examines if it fulfills the conditions. If not, a flag is set and the sequence of algorithms is stopped.[2]

3.1.4 ScintiDigitizer, MDTDigitizer, StreamerDigitizer: Digit Production

Now that the we assume that the simulated event would have been accepted by the trigger mechanisms, hits can be fetched from **StoreGate** and used to create digits.

The fundamental difference between hits and digits in a real experiment is, that hits have no representation in the data processing chain. In general, hits contain more information than digits. In the case of MDT chambers (**MDTDigitizer**), hits contain the information of where a particle passes a drift tube, including its radius as well as its distance to the tubes readout. Using this information, a time delay is calculated which leads to timing information at which the channel is read out. Mapping between radii at which charged particle pass by the anode wire and the time it takes until the signal reaches the wire, as well as an error estimation involves the so-called drift time relation. In addition to that, the **MDTDigitizer** takes into account the distance of the hit to the readout of its tube, for this leads to a signal run time delay that has to be considered. This relative time information is given in units of TDC counts, which for the current setup is $\frac{32}{25}\,\frac{\text{ns}}{\text{count}}$. The digit itself is smeared by a Gaussian distribution.

The digitization of streamer- and scintillator hits is more simple, since the only information kept is which channel saw a signal and which did not.

One speciality of the digitization of streamer digits is connected to the fact that a hit in one streamer volume is much likely to affect neighbor volumes. To accomodate this in MCT, the channel identification scheme allows easy acces to the identification of neighbor volumes. The pulse height seen by the neighbor volumes is calculated according to a simple electrostatic model and is introduced in appendix A together with a description of the physical processes leading to streamer hits.

At the end of the `execute()`-methods, all digits are stored in the event store. From here, as described above, they can be made persistent and/or be further processed.

3.2 Detailed Description of a Reconstruction Job

To a certain extend, the work done by the reconstruction job described below reverses the work of the simulation. It ties seamlessly to the simulation job, i. e. its algorithms could follow imediately. The goal of this reconstruction job is to create track parameters from mdt chamber, scintillator and streamer digits for each event in terms of three[3] local tracks, one for each chamber, and a global track, ideally through all three chambers. The track parameters with its fit information as well as digit information is stored in an n-tuple.

Of course, once again, having subsequent **Algorithm**s that pick up the track information from **StoreGate** and perform further calculations is conceivable. As an example, the **Algorithm ChamberPosition** uses track information to calculate the positions of the chambers.[4]

[2] This requires the **Algorithm**s to be organized in so-called sequences.

[3] In certain cases only two, as we will see later.

[4] The precision at which the chambers are aligned when they are installed is limited. The procedure that determines the exact positions is described later.

3.2.1 TDCDelayAdjust: Calculation of Run Time Corrections

A MDT chamber digit is a time equivalent given in TDC counts ($\frac{32}{25}\ \frac{\text{ns}}{\text{count}}$). It holds information about the difference between the global time a channel (tube) has been read out and a reference time given by the lower scintillator system. It can be shown that the reference time given by the lower hodoscope equals the global time at which the cosmic particle hits the scintillator system plus an unknown, but fixed constant.

In order to gain information about the "real" drift time, i. e. the difference between the time, the particle enters a gas volume and the time at which the ionized gas cloud reaches the wire, several corrections have to be done.

TDCDelayAdjust picks up MDTDigits from **StoreGate** and performs corrections due to two effects in the setup: the first and overbearing effect is the signal run time along the wire. Depending on the x-position (refer to figure 1.3 on page 8 for the coordinate system definition) of the hit, the read out electronic sees a signal that is delayed by the time it takes to travel along the wire. **TDCDelayAdjust** determines the x-coordinate of the impact and estimates the time delay. Second, the fact is accommodated that, even though the Muons travel nearly at the speed of light, the time at which single drift tubes are hit depends on their z-position. Furthermore, a very rough track fit is performed to estimate the inclination of the Muon trajectory, which affects the time of flight of the muon.

The time-of-flight correction is surmounted by the signal-run-time correction. Nevertheless, it is nearly about the order of magnitude of the design precision of the setup.

The result of this calculation is put into the event store in terms of **DriftTime**s, which simply hold a time for each channel. (This 'time' still is not the assumed drift time of the channel.)

As said above, **TDCDelayAdjust** simply picks up digits from **StoreGate**. This is even possible in the case that no preceding **Algorithm** stored them, provided that converters are involved. Depending on the converters defined in joboptions, the calculation can be done with real test stand data or with Monte Carlo data.

3.2.2 MDTTimeToRadTransform: Translation between Drift Times and Drift Radii

In addition to the run time effects addressed by **TDCDelayAdjust**, **MDTTimeToRadTransform** performs corrections that are necessary due to different cable length and electronic read out effects. In contrary to the corrections calculated by **TDCDelayAdjust**, these corrections do not vary with time or event (at least within one run). The drift time spectra for single tubes look alike, except for a additive constant given by cable lengths and electronics internals. **MDTTimeToRadTransform** takes into account this constant to calculate a "real drift time".

Figure 3.1 shows a drift time spectrum for a single tube. The offset is already corrected to be zero (the MCT **Algorithm**s that determines the edges of drift time spectra is introduced later).

The offset for each single tube is determined by an **Algorithm** in the MCT sub package MCTCalibAlgs, the **T0Fitter**. The results are organized and made available via the interval of validity-mechanism described later.

With the "real drift time", **MDTTimeToRadTransform** calculates an estimate for the drift radius and its error. The result, called a **DriftCircle** also contains information of x-, y- and z-positions and is stored in the event store.

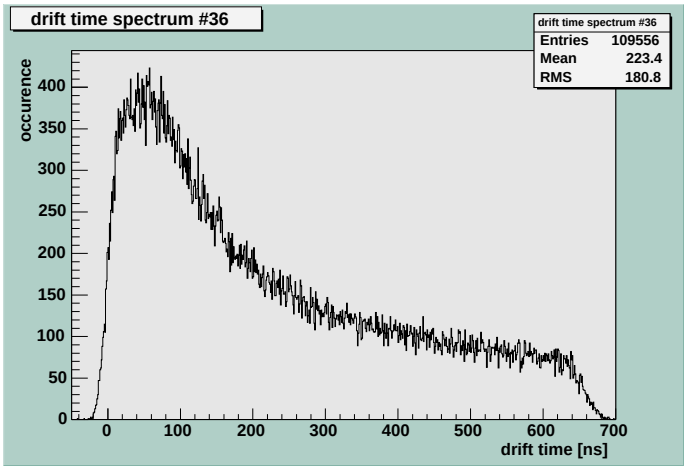

Figure 3.1: Drift time spectrum: the length of this spectrum corresponds to the maximum radius seen by a drift tube. The starting point (here corrected to be zero) depends on cable lengths and electronics internals.

3.2.3 MDTPatternFinder: Definition of Subsets of Drift Circles that might form a Muon Trajectory

A pattern is a subset of all drift-circles whose centers have a distance from a given straight line less than a certain road width. **MDTPatternFinder** catches collections of **DriftCircles** and examines them to establish if they might belong to a pattern.

As result, **MDTPatternFinder** stores several local patterns (subsets of drift-circles of one chamber) as well as global patterns (overall subsets of all three chambers) in the event store. Global patterns are always built of local patterns in each chamber.

3.2.4 MDTTrackFitter: Fitting of Tracks

MDTTrackFitter's purpose is to evaluate patterns by performing fits.

First, all global patterns are evaluated. By means of certain criteria, the "best" global pattern is determined. Accordingly, the (usually three) local patterns that build this global pattern are taken for the local fits. In the end, four tracks (one global and three local) in terms of slope and intercept are written to **StoreGate**, including fit information.

The selected parametrization of muon tracks reflects the fact that MDT chamber measurements are insensitive against the x coordinate (having in mind the coordinate system definition given in figure 1.3): four parameters, m_x, b_x, m_y and b_y describe two projections of a track,

$$\begin{aligned} x(z) &= m_x \cdot z + b_x \text{ and} \\ y(z) &= m_y \cdot z + b_y. \end{aligned}$$

(m_x, b_x) is determined by a rather simple, not further discussed fit that evaluates scintillator information while (m_y, b_y) are determined by the method just described.

MDTTrackFitter is very flexible when it comes to choosing strategies for the fit by exploiting the Athena tool mechanism (see above). Figure (3.2) gives an idea of which decisions have to be made when fitting tracks.

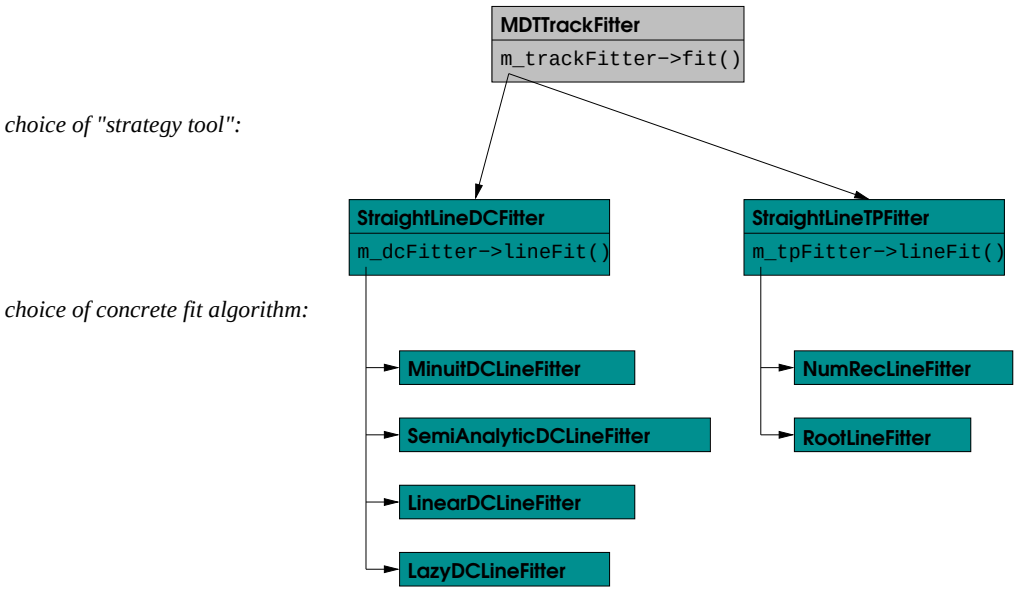

Figure 3.2: **MDTTrackFitter** strategy design: two decisions need to be defined. Via the joboptions mechanism, the end-user defines which strategy **MDTTrackFitter** uses. The next decision concerns the concrete algorithm used for fitting. The list of fit algorithm alternatives can easily be expanded.

In the first stage, the joboptions define which Strategy-**Tool** to use. At the moment, two strategies are implemented: One minimizes a straight line by minimizing residua in terms of distances of a straight line to drift circles. The other determines a set of points from a set of drift circles (one for each) and performs a linear regression with these points.

In the next stage, the concrete fitting algorithm for the selected strategy is defined. The list of concrete fit algorithms can be easily expanded. Examples of the already implemented fitters are **MinuitDCLineFitter** (uses MINUIT as minimization engine) and **NumRecLineFitter** (uses a linear regression algorithm found at [7]).

These definitions made in joboptions files affect the data member `m_trackFitter` of **MuonCTTrackFitter** or `m_dcFitter` (`m_tpFitter`) of **StraightLineDCFitter** (**StraightLineTPFitter**)[5].

3.2.5 MuonCTNTuple: N-Tuple Production

When the fitting is done, **MuonCTNTuple** catches a broad variety of the event information from **StoreGate** and interacts with the **NTupleSvc** to produce n-tuples. **MuonCTNTuple** is not aware of the concrete technology used by the **NTupleSvc** to actually write n-tuples on disk. This is defined via joboptions. An explanation of the concrete n-tuple produced is given in appendix D.

[5] 'tp' means track point, 'dc' means drift circle.

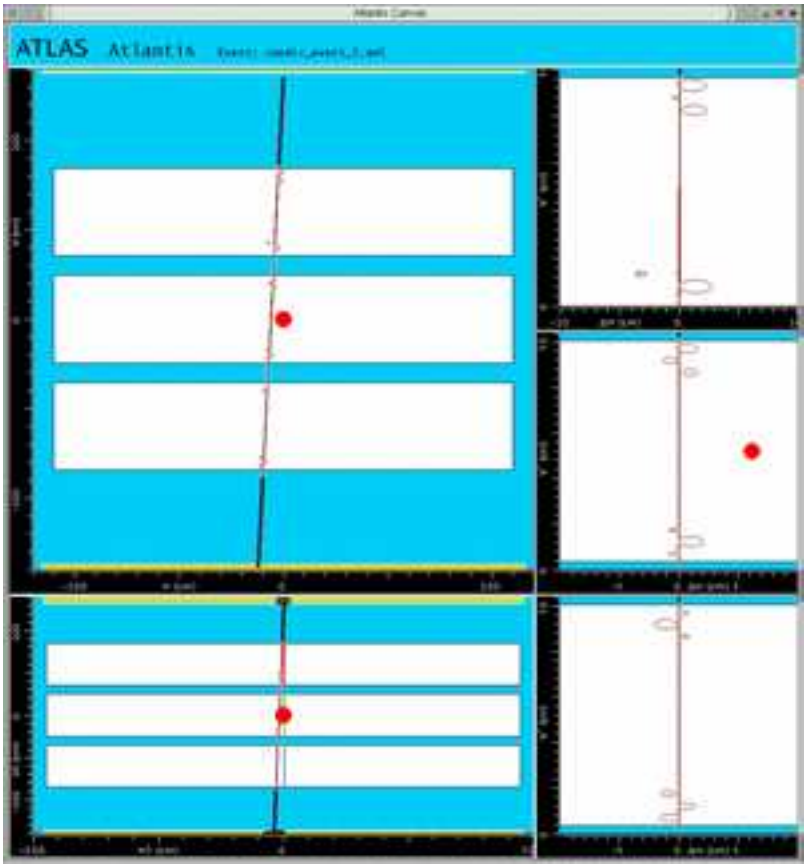

Figure 3.3: Atlantis event display: on the left hand side one sees the two vertical cuts through the detector (top: front view along x, bottom: side view along y). The right hand side shows three zoomed views (one for each chamber) of the track along x. Drift circles are shown as well as the reconstructed track.

3.2.6 AtlantisXMLTrackConverter: Visualization of Tracks

Atlantis (see [8]) is a java based event display with a graphical user interface. It reads in XML-files from disk. The purpose of **AtlantisXMLTrackConverter** is to collect fit information from **StoreGate** and produce XML output, that is later interpreted by a separate program, the Atlantis Event Display. Figure 3.3 shows the output of Atlantis for an example event.

Chapter 4

Geometry and Calibration Aspects in MCT

The previous chapter illustrated two of the main tasks that are fulfilled by the MCT package. Additional tasks, such as the determination of calibration parameters, can be realized by composing joboptions of the components residing in MCT sub packages.

The following sections elaborate single aspects in more detail. For every section, a problem is deduced followed by a solution given in a design- and implementation part. The former describes the solution in an abstract way, while the latter tries to give explanations on the concrete implementation. Readers who intend to work with the MCT package should have a look into the source code, available from [9], too, when reading the implementation part.

4.1 Geometry Model Conversion

As already mentioned in 3.1.2, Geant4 (see [10]) is currently the simulation engine of choice utilized inside Athena. Geant4 is designed to be a toolkit from which stand-alone simulation applications can be developed. Therefore, it provides its own geometry description classes, which are mandatory for a simulation run. Hit production inside Geant4 is implemented by means of so-called sensitive detector objects which are attached to certain volumes inside the Geant4 geometry tree. Sensitive detector objects contain algorithms capable of evaluating tracking information (such as energy and momentum of a particle, momentum direction and energy deposition in material) that is used for hit production. Whenever the tracking engine tracks a particle inside the boundaries of such a volume, the `ProcessHits()` method of the attached sensitive detector is invoked. Athena on the other hand currently uses GeoModel as the geometry description model, which is designed to fulfill requirements which are different from the Geant4 geometry model, such as minimized memory consumption. At the moment, in a simulation job both geometry descriptions reside in the transient store. For reasons of consistency it is inevitable to have one description being derived from the other. Since Geant4 can be regarded as a utilized helper toolkit, the Geant4 geometry needs to be derived from the GeoModel description. At the time this thesis was developed, no Athena-wide solution existed for this problem.[1] Therefore, the MCT software package implements its own class for geometry conversion. It is held very general so it can be used not only by the Munich test stand.

[1] If the Athena developers community decides to adhere to the two-framework-solution, an Athena-wide conversion mechanism is necessary and might be influenced by the already existing MCT conversion implementation.

4.1.1 Problem Definition: Derivation of Geant4 Boundary Conditions

Starting from an already in memory existing GeoModel geometry description, the conversion mechanism has to accommodate two tasks:

- One-to-one translation of GeoModel geometry description into a corresponding Geant4 description and
- incorporation of Geant4 sensitive detector objects.

Following 3.1.2, the Geant4 geometry representation needs to be registered with the **G4Svc** in the `initialize()` method of **MuonCTG4Sim**. At this stage of job execution, the geometry conversion has to be finished. Geometry conversion is subjected to to three principles, which turn out to be non trivial (the following three points are referred to as the *design principles* throughout this chapter).

1. When designing the GeoModel representation of a geometry setup, one should not have to care about later conversion processes.

The GeoModel tree for the ATLAS detector is very complex and is designed by many people. If GeoModel tree designers have to follow certain rules that are dictated by a later conversion process, violations of these rules are hard to detect. If a violation of a rule prevents parts of the geometry in Geant4 from being alignable, the effect is rather small, since e. g. chamber alignment in MCT has consequences on the micrometer scale.

2. Every volume that is alignable in GeoModel (by means of so-called **GeoAlignableTransforms**, see 2.4.1) needs to be alignable in Geant4, too.

In ATLAS, one aims to be able to simulate an ideal geometry as well as a misaligned geometry. The geometry of the chambers in MCT is misaligned due to limitations of the insertion of chambers into the test stand. Alignability of volumes in the GeoModel view of the setup is realized by **GeoAlignableTransforms**, which hold pointers to two transformations, one representing the nominal position of the volume, the other representing a small deviation. When constructing the Geant4 geometry tree, one has to accomodate the fact that Geant4 volumes only hold one transformation, which consequently needs to correspond to the nominal position concateneted with its small deviation.

3. The Geant4 representation needs to be lightweight and fast.

This is not a problem for the MCT project, since the geometry is rather simple. This is different for ATLAS, where the complexity is enormous. This makes it impossible to rebuild every real volume by an instance of a GeoModel- or Geant4-object. Identical volumes therefore need to be *shared*, i. e., multiple occurrences of identical volumes need to be realized via pointers and not via additional objects. Sharing of volumes is realized in GeoModel and ideally needs to be translated into its Geant4 correspondence.

Alignability in Geant4 is important for simulation studies of effects of misaligned geometries. Considering incompatibilities between GeoModel and Geant4 geometry description, the 2nd design principle turns out to be in opposition to a lightweight and fast Geant4 description. Furthermore, freeing the GeoModel tree designer from all rules might collide with alignability. This is explained later in the implementation section.

Integration of Geant4 sensitive detectors in the Athena concept is non-trivial, too. While the pure Geant4 geometry description has an analogy on the Athena side (which is GeoModel itself), Geant4 sensitive detectors have not.

4.1.2 Concept and Design

Geometry Conversion

A naive approach would be to catch the GeoModel tree from the detector store in the `initialize()`-method of **MuonCTG4Sim**, perform the conversion explicitly and register the result with the **G4Svc**. Instead, an Athena service is established that provides a more general solution. The **MuonCTGeoToG4Svc** is a service in the Athena sense and performs the calculation through an intuitive interface:

```
G4VUserDetectorConstruction* build(GeoModelExperiment* world)
```

(This interface definition reads as follows: the `build()`-method of the **MuonCTGeoToG4Svc** service takes as argument a pointer to a GeoModel root node (`GeoModelExperiment*`) and returns a pointer to the corresponding Geant4 setup (the geometry tree with all Geant4 sensitive detectors attached, `G4VUserDetectorConstruction*`)) This service is available from everywhere inside the framework and can be used to convert arbitrary GeoModel geometries.

Even though **MuonCTGeoToG4Svc** is a very general approach, the name[2] of the service has been chosen MCT-specific. In a larger Athena context, **MuonCTGeoToG4Svc** does not aim to be the general future solution for geometry conversion itself, but a source of inspiration for an Athena-wide solution. The name should state this clearly and furthermore avoid conflicts with possible future, general implementations.

Only a few lines are left for the `initialize()`-method of **MuonCTG4Sim** to establish a Geant4 simulation environment: Catch the GeoModel setup from the detector store,

```
status = m_dtStr->retrieve( world );
```

(`m_dtStr` is a pointer to the detector store) perform the conversion using the **MuonCTGeoToG4Svc**

```
G4VUserDetectorConstruction* teststand = m_GeoToG4Svc->build( world );
```

and finally register the result with the **G4Svc**

```
m_G4Svc->SetUserInitialization(teststand);.
```

This is the way the geometry conversion is performed in **MuonCTG4Sim**.

Sensitive Detector Integration

As stated above, there is no equivalent inside Athena for Geant4 sensitive detectors (**G4VSensitiveDetector**s). They are *optional* for Geant4 in the following sense: a Geant4 geometry description without a sensitive detector makes sense and can be used for tracking etc., while a sensitive detector that is not attached to a certain volume of a Geant4 geometry description is completely useless. Geant4 sensitive detectors itself are pure *algorithms* without any prior knowledge of the underlying geometry.

For **MuonCTGeoToG4Svc** an approach has been made that exploits an Athena feature that deals with *optional algorithms*: Athena tools. Tools (described in section 2.2.2) are objects, that are instantiated and made available via the **ToolSvc** *on demand*. The marriage between Athena **Tool**s and Geant4 **G4VSensitiveDetector**s results in **IG4VSensitiveDetectorTool**,

[2] Within Athena, a service is made available via its name.

using C++ multiple inheritance feature and is described below. **IG4VSensitiveDetectorTool**s can serve both interfaces, either of Geant4 sensitive detectors or Athena tools.

Since **Tool**s are made available via their name, **MuonCTGeoToG4Svc** receives a list of `std::string` in the joboptions file, e. g.

```
MuonCTGeoToG4Svc.SensitiveDetectors =
{ "MDTSD", "ScintillatorSD", "StreamerSD" };
```

(**MDTSD** is the sensitive detector producing drift tube hits, **ScintillatorSD** for scintillator hits and **StreamerSD** for the streamer tube hits.[3]) These are the names of the sensitive detector tools that shall be used in the simulation job described by that joboptions file. The **IG4VSensitiveDetectorTool**s themselves receive a `std::string` in the joboptions, too, which is the name of the volume they are attached to, e. g.:

```
ToolSvc.MDTSD.ActiveLogVolName = "DriftGasLog";
```

This means that **MDTSD** should be attached to a volume with name DriftGasLog, which, as one might guess, is the name of the gas volumes inside drift tubes. **MuonCTGeoToG4Svc** accounts for these names when performing the conversion. The design chosen allows the user to define different sensitive detector configurations (i. e. which ones are instantiated and which volumes they are bound to) in the joboptions without any need to recompile the code.

4.1.3 Implementation

This section will give implementation details. This information is not necessary for somebody only using the **MuonCTGeoToG4Svc**. A rather deep level of detail in this illustration of the underlying algorithms has been chosen since it reveals deeper lying incompatibilities between the two geometry descriptions. All code described here can be found in the MuonCTG4Sim sub package of MCT.[4]

Geometry Conversion

For performing the conversion of the geometry, a helper class is utilized: **GeoToG4Conversion**. The following methods are provided by this class:

- `G4VPhysicalVolume* Construct();`
- `G4VSolid* GeoShapeToG4 (const GeoShape*);`
- `G4Material* GeoMatToG4 (const GeoMaterial* );`
- `G4Element* GeoEleToG4 (const GeoElement*);`
- `G4LogicalVolume* GeoPhysToG4Log (PVConstLink);`
- `void setVisOpts (G4LogicalVolume*);`
- `void setSensDet (G4LogicalVolume*);`

[3] Note:**MDTSD**, **ScintillatorSD** and **StreamerSD** are not only **Tool**s but also **G4VSensitiveDetector**s!

[4] A hint lightens the reading of the following lines: every class from within the Geant4-context starts with "**G4**...", while GeoModel classes start with "**Geo**..."

It follows a recursive approach: the `Construct()`-method, which is accessed from inside **MuonCTGeoToG4Svc**, mainly consists of the following statement:

```
return new G4PVPlacement(0, G4ThreeVector(),"experimentalHall",
          GeoPhysToG4Log( m_theExpt ),0, false, 0);
```

where `m_theExpt` is the pointer to the root node of the GeoModel tree to be converted. `GeoPhysToG4Log()` plays a central role in this recursive approach. First, it converts all geometrical properties of the GeoModel volume in the corresponding Geant4 description. Second, it looks for child volumes and converts them recursively. In a third step, properties that are independent of the GeoModel (such as Geant4 coloring and attaching sensitive detectors) are added to the volume:

1. `GeoPhysToG4Log()` decomposes the physical volume (which is represented by a GeoModel object) it retrieved as argument into its properties, represented by means of instances of classes defined in GeoModel: **GeoShape**, **GeoMaterial**. Pointers to these property-objects are handled over to `GeoMatToG4()` (which itself utilizes `GeoEleToG4()`) or `GeoShapeToG4()`. They return pointers to objects representing the same properties in terms of Geant4 (**G4Solid** and **G4Material**).

2. In the next step, `GeoPhysToG4Log()` examines the GeoModel physical volume it initially received as argument for possible child-volumes (e. g. as indicated in figure 2.3, the volume "Chamber" has two volumes named "Multilayer" as direct children). *Pointers to these children are taken as argument for a recursive call of* `GeoPhysToG4Log()` *itself.* The output of this recursive function call (a pointer to a **G4VPhysicalVolume**) is attached to the logical volume currently processed as its child volume.

3. In its constructor, **GeoToG4Conversion** gets two tables: one contains visualization details for the Geant4 visualization engine. The other maps logical volume names to pointers of sensitive detectors. These maps are utilized in `setVisOpts()` and `setSensDet()`, which simply set the visualization- and sensitive detector-properties of Geant4 logical volumes and are called from `GeoPhysToG4Log()`, too.

The outcome of all function calls from within `GeoPhysToG4Log()`, namely `GeoShapeToG4()`, `GeoMatToG4()`, `setVisOpts()`, `setSensDet()` and the recursive `GeoPhysToG4Log()` function call(s) is taken to build a new instance of a Geant4 logical volume. The pointer to this "fully configured" object is returned from `GeoPhysToG4Log()`.

Incompatibility of Geometry Models

At this stage, a fundamental incompatibility between Geant4 and GeoModel geometry description comes up that is shown in figure 4.1. The different approaches of geometry description are presented in a simplified way. In Geant4, substructure is represented by means of pointers to the daughter logical volumes, held by the mother logical volume. GeoModel in contrast introduces an object called physical volume (that can be identified with the **GeoPhysVol** discussed in 2.4.1) that holds pointers to either the properties of the volume it represents (in terms of a logical volume, just as in Geant4) as well as the pointers to its substructure - pointers to daughter physical volumes. The big difference between the two approaches is clear: given, a logical volume appears several thousand times in a detector setup, as for drift tubes in the ATLAS detector. Drift tubes are considered as "daughter volumes" of their drift chamber. Lets

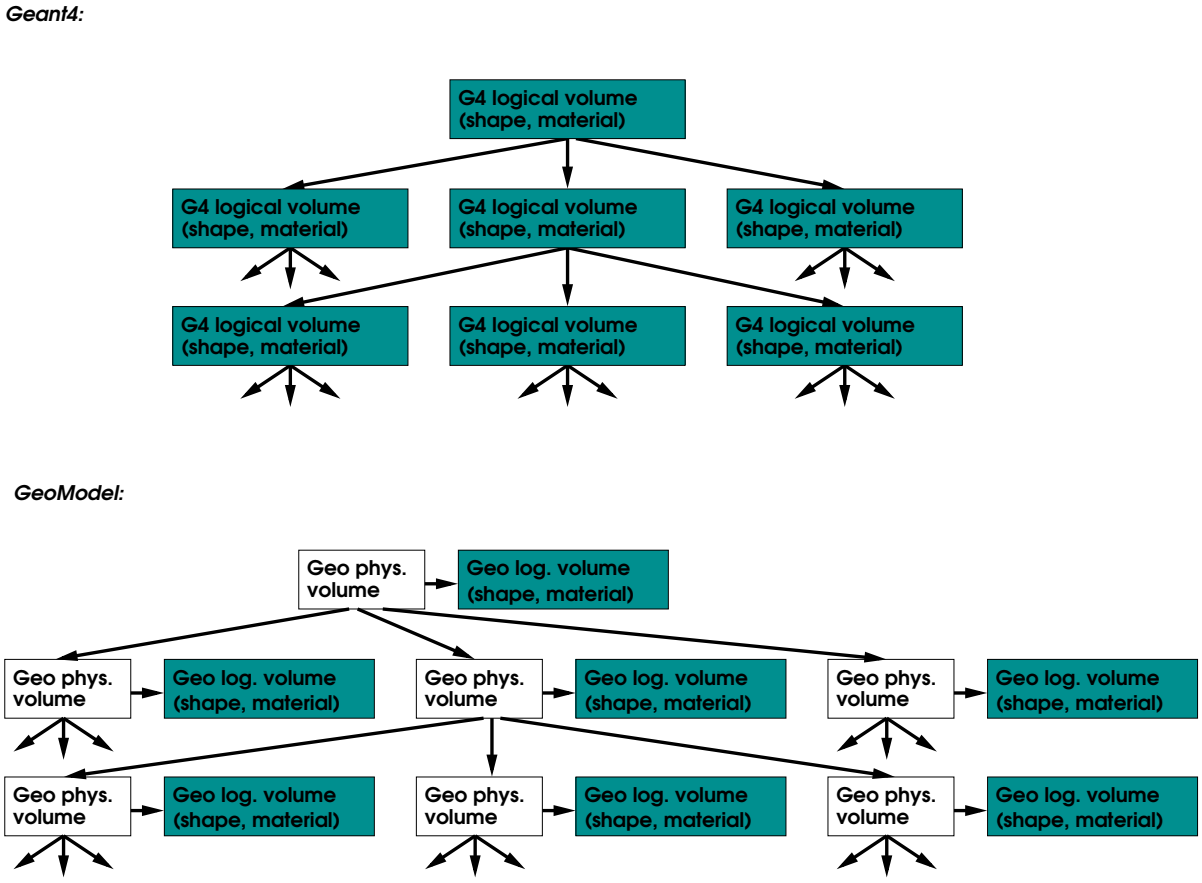

Figure 4.1: Incompatibility between Geant4 and GeoModel geometry description: in Geant4, substructure is associated with a logical volume while in GeoModel, substructure is associated to a so-called physical volume. Filled boxes represent memory-consuming objects. On the GeoModel-side, logical volumes need not to be instantiated multiple time. Physical volumes, that have the same geometrical properties, simply point to one single instance representing this property. (The latter is not shown in the picture and left to the reader's imagination.)

assume all drift tubes are slightly misaligned in their chambers, such that no chamber resembles another. With GeoModel, the tree designer defines the properties of chambers *only once*. He instantiates exactly one copy of a logical volume object describing the shape/material properties of a drift chamber. Every physical volume representing a drift chamber gets a pointer to this single instance associated, as well as pointers to the physical volumes representing the drift tubes below.[5]

With Geant4, this approach is impossible: when sharing a logical volume (including shape and material information), all substructure is shared, too. But this is not wanted in the considered example, since all chambers carry slightly (different) misaligned configurations of tubes, and thus the substructure is not the same. In consequence, it is not possible to share the properties of drift chambers in Geant4. Every drift chamber has to be represented by its own instance of a logical volume, including all shape and material information. This can be rather memory-consuming and contradicts the third point given in the design principle list above.

If the chambers follow an ideal geometry, sharing of logical volumes in Geant4 makes perfectly sense as the reader can imagine, and it would be even better in terms of memory saving then GeoModel. But, even if the chambers considered in the example above are all the same, a problem arises. Sensitive detectors attached to a shared volume have no chance gathering information about which volume they are attached to. But, a hit object produced by a sensitive detector, needs to be assigned to a channel, e. g. an identifier of a drift tube. Otherwise it is not possible to determine the coordinates at which this hit occurred.

Of course, identification of volumes is not impossible in Geant4 stand alone applications, but it contradicts the Athena way. Identification in Geant4 is subdued to the geometry layout, namely the mother-daughter relationship. For example, if a drift tube is the 25th daughter of an MDT chamber, which itself is the second daughter of the teststand setup, the Geant4 identification for this drift tube would be "2/25" (of course, this example is very simple, and more complex mother-daughter relationships would lead to more complex tuples).

In Athena, identification is carried out via grouping logical units (which, in the current layout, in a simplified way is a tuple representing subsystem/detector element/readout channel). This logical grouping is completely independent of mother-daughter relationships in GeoModel, even though it is possible for the GeoModel tree designer to resemble this logical grouping in the way the geometry model is designed, but this would contradict the first point of the three design priciples above.

Several approaches have been thought of that could solve this problem. One was to decide on the GeoModel side, which volume has to be shared in Geant4 and which not. Unfortunately, this contradicts the first principle given above.

For the Munich Test Facility, the situation is quite different compared to the ATLAS experiment. The experiments complexity is orders of magnitude below the ATLAS one. Therefore, **GeoToG4Conversion** simply never shares logical volumes on the Geant4 side. Instead, it builds individual instances of logical volumes for every volume found in the GeoModel, if it could be shared or not. This is not passable for the ATLAS detector since it would exhaust the machines memory it is run on. But, for MCT, this approach does work.

At the time this thesis is being written, the problem described above is clearly recognized by the Athena community, but no solution exists so far. It will be interesting to follow the devel-

[5]Readers used to Geant4 might argue that even Geant4 offers a class **G4VPhysicalVolume**. This is true, but its name is misleading. In the internal representation, substructure is still associated to logical volumes, even if it has explicitly been assigned to **G4VPhysicalVolume**s. The name **GeoPhysToG4Log** has been chosen to make this point clear: *physical volumes* of GeoModel are converted to *logical volumes* of Geant4.

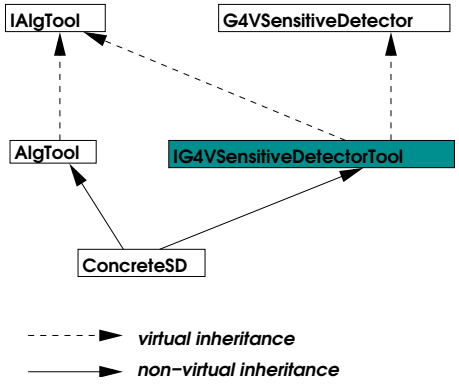

Figure 4.2: Inheritance diagram for **IG4VSensitiveDetectorTool**.

opment on this topic, since a possible solution might require changes in the Geant4 geometry, even though Geant4 is not maintained by the Athena developers.

Sensitive Detector Integration

The last section already introduced the method `setSensDet()`, where sensitive detectors are attached to logical volumes following directives of a `std::map`[6] received of **GeoToG4Conversion** in its constructor. The map itself is build from **MuonCTGeoToG4Svc** according to its joboptions. **MuonCTGeoToG4Svc** requests the tools from the tool service and inserts pointers to them in that map.

`setSensDet()` is rather straightforward, it simply matches the name of logical volumes to the map and possibly attached sensitive detectors.

Figure 4.2 shows an inheritance diagram for **IG4VSensitiveDetectorTool**. Instances of **IG4VSensitiveDetectorTool** have all properties of either **Tool**s or **G4VSensitiveDetector**s. Concrete sensitive detectors inherit from both, **AlgTool** and **IG4VSensitiveDetectorTool**.

4.2 Obtaining Calibration Data

With the **IOVSvc**, Athena offers a comfortable scheme for calibration data handling. Calibration "constants" of different type vary in time and need to be easily accessible. The following sections describe the data types that are already integrated in the **IOVSvc** mechanism as well as how they are obtained.

Additional time varying data can be integrated in the future, such as cable plugging information, drift time relations and RasNiK [1] information about dynamic chamber movement and transformation.

[6] A `std::map` is a container that is comparable to an array. An extension of C++, the so-called Standard Template Library (STL), provides several array-like containers. An overview of STL extensions can be found in [11]

4.2.1 Problem Definition: Time Varying Data

Drift Time Spectra

Muons passing through the gas mixture of an MDT tube cause atoms to ionize. The positively charged nuclei drift towards the tube wall, while the negatively charged electrons drift towards the anode wire. Close to the wire, the electric field is strong enough to impart the electrons an energy high enough to ionize further atoms. A charge cloud arises in the neighborhood of the wire. The burst influenced on the anode wire is measured together with a global time information of its incidence. The scintillator trigger of the test stand measures the global time of the muons passing through the lower two scintillator layers (plus a non-varying constant). The difference between these two global times is the *raw drift time*. After applying some corrections on these raw drift times, which have been described in section 3.2.1 (description of the **Algorithm TDCDelayAdjust**), one yields a *drift time* τ that (ideally) only varies with the radius at which a muon passes by an anode wire.

A spectrum of these drift times for individual tubes not necessarily starts at $\tau = 0$. The start- and endpoints (t_0 and t_m) of the spectrum depend on cable lenght and electronic internals. However, the length of the spectrum, $t_m - t_0$, should be nearly the same for all tubes. Deviations in the lenght of a drift time spectrum may indicate problems with the gas tightness of a tube.

Figure 3.1 on page 22 shows a typical drift time spectrum. The rising edge has already been corrected to be zero.

With a perfect tube resolution, the rising and falling edges of the spectrum would be step-like functions. Due to a limited resolution, the transition on the edges becomes continous and is described rather good by two fermi-like functions,

$$G(\tau) := z_0 + \frac{A_0}{1 + e^{-\frac{\tau - t_0}{\tau_0}}}$$

for the left and

$$H(\tau) := z_m + \frac{A_m - \alpha_m \tau}{1 + e^{\frac{\tau - t_m}{T_M}}}$$

for the right edge. Both are restricted on a limited interval around their edge. z_0 and z_m describe the permanent background of the drift time distributions. τ_0 and T_M indicate the slope, at which the edges rise respectively fall. τ_0 or T_M being close to zero indicates a step-like rise respectively fall of the edges. Fermi functions are flat, except for the transit interval. This applies for the left edge, while it does not for the right edge. Here, the spectrum is described by a negatively inclined linear function for τ being left with respect to the transition interval. The term $\alpha_m \tau$ accomodates this observation.

Chamber Positions

Chambers can be statically or dynamically misaligned. Statical misalignment is unavoidable and due to imperfect adjustment when inserted into the test facility. Dynamical misalignment is due to fluctuations in the outer conditions and should be monitored by the RasNiK system.

In both cases, a chamber is considered as a stiff body, whose misalignment can be expressed with a transformation consisting of a translation and a rotation.

4.2.2 Concept and Design

Drift Time Spectra

A dedicated **Algorithm** situated in the sub package MuonCTCalibAlgs, **T0Fitter**, is used to determine the drift time spectra information. A joboptions file for a calibration job is configured such that **T0Fitter**s `execute()`-method is invoked after **TDCDelayAdjust**s one. In this method, it collects run time corrected drift times from **StoreGate**. At the end of a job, **T0Fitter** tries to fit modified Fermi functions to each edge of the spectrum in its `finalize()`-method. Further information about this fit can be found in [12].

In a calibration run, this fit can performed automatically for every single tube. If the statistic is very low, it might be a good choice not to fit single tubes but to fit the sums of histograms of groups of tubes. The number of tubes that are taken for a fit can be selected via joboptions. The transient data structure of drift spectrum information needs to flexible enough to accomodate every choice of spectrum gouping.

The result is written to the calibration database, which is done by converters.

Chamber Alignment

The procedure of determining static misalignment using reconstructed muon tracks results in two sets of five parameters describing translation and rotation of the reference chambers relative to their ideal position. The sixth parameter, a translation along the x-axis, is not considered since the chambers are not sensitive to this direction (the coordinate system was defined in figure 1.3). The calculation described below is performed by a dedicated **Algorithm**, **ChamberPosition**.

The procedure gathers information from **StoreGate** of locally reconstructed muon tracks event by event in the `execute()`-method of the **Algorithm** (mainly slope and intercept of the track following the parametrization defined in section 3.2.4 on page 22) and determines the true position of the chamber later in the `finalize()`-method by minimizing certain residua. The test chamber is assumed to be perfectly aligned in the origin of the setup while relative displacements of the reference chambers are determined individually.

ChamberPosition performs a sequence of the following calculations *iteratively* within a loop in its `finalize()`-method. Once a transformation as a first candidate for the displacement of a chamber is determined, all track parameters (of all events) undergo this transformation. This leads to new local track parameters, with which a new candidate for the transformation is calculated and so on.

In fact, not every parameter (shift in y, z or rotation around x, y, z) is considered in each iteration step. The algorithm turned out to be more robust, especially in the case of low statistics, if translations are corrected in every iteration step, while the rotations are not. In the present configuration, translations are considered in every step, while rotations around x, y and z are considered only every 2nd, 4th and 8th step, respectively.

Once the final transformation for both reference chambers is found, it is merged in the IOV/conditions database (currently manually) and it is recommended to do even a second run of **ChamberPosition**, since the result of **ChamberPosition**, once integrated in the IOV service, transforms chambers, while the internal iteration loop of **ChamberPosition** transforms track parameters. The results might differ slightly since a transformation of chambers, even at such a small scale, might affect track fitting and pattern recognition algorithms.

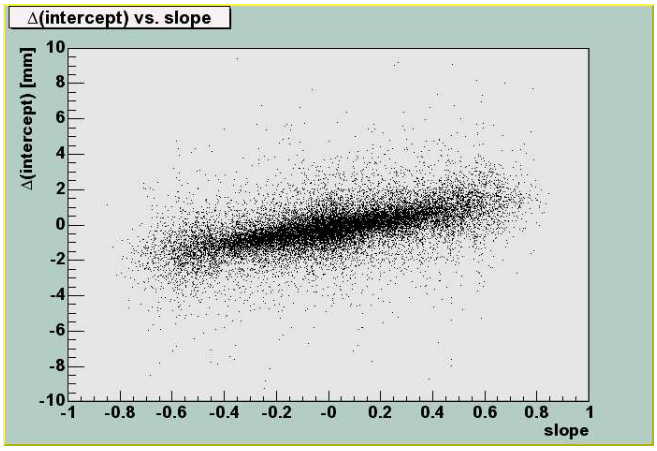

Figure 4.3: How chamber misalignment is measured: the x-axis shows the slope measured in the test chamber, while the y-axis shows the differences in intercept seen by test respectively reference chamber.

Chamber Alignment: Translations along y and z

Translations along y and z of each of the reference chambers are determined in a rather straightforward fashion. For each event, the *difference of the intercepts* b_y seen by the test chamber or reference chamber, Δb_y, as well as the *slope seen by the test chamber*, $m_{y,\text{testchamber}}$, are collected. They are fitted later assuming a linear relationship between Δb_y and $m_{y,\text{testchamber}}$. Figure 4.3 shows a plot of these two variables. The slope and intercept of this linear regression correspond to z-shift and y-shift of the reference chamber relative to the test chamber .

$\Delta b_y(m_{y,\text{testchamber}} = 0)$ reflects the misalignment along y, which corresponds to the intercept of the considered linear regression. A z-shift is measured as a Δb_y, that depends linear on the measured slope, $m_{y,\text{testchamber}}$. In case of no y-shift, Δb_y would be directly proportional to $m_{y,\text{testchamber}}$.

Chamber Alignment: Rotations around y and z

The rotation angle determination is based on a small angle approximation. Therefore, it is allowed to calculate three rotation angles independently of each other. Rotations around y and z are calculated utilizing the method used for y- and z-shift calculation described above. However, the y- respectively z-shift is not calculated globally for a whole chamber, but locally for tracks passing through the chamber in two well defined regions along x (a rough but sufficient track fit in the xz-projection is done using scintillator information).

By getting different results for the y- and z-shifts for different regions of a reference chamber, one in the front and one in the back, the rotation angles around y and z can easily be determined.

Chamber Alignment: Rotations around x

Rotations around x are determined by averaging the differences in slope m_y of the test, and reference, chamber.

4.2.3 Implementation

The implementation is derived straight forward from the concepts discussed above: collecting information in the `execute()`-method and performing numerical fits in the `finalize()`-method. In both cases, the minimization engine MINUIT (refer to [13]) is used, interfaced by Root (refer to [14]). It is not further discussed here.

The data structure of the drift spectrum information is introduced in 4.4, where the access to time varying data is discussed. It allows to group an arbitrary number of adjacent drift tube in one spectrum.

4.3 IOV and Conditions Databases

4.3.1 Problem Definition: Two Database Design of the IOV Service

Figure 2.4 on page 16 illustrates the interaction between the IOV and conditions database. The IOV database maps (run, event)-pairs on `std::string`s, so-called tags. The conditions database maps this tag on data that is interpreted by a converter and used to build the transient representation of a requested object.

Athena avoids dependencies on external technologies and makes them exchangeable via joboptions. In the case of IOV and conditions databases, the concrete **IIOVDBSvc** that reads the IOV database and a converter that reads conditions data need to be defined.

At the time this thesis is being written, some implementations of **IIOVDBSvc** and conditions data converters exist in the Athena repository. For the **IIOVDBSvc**, a MySQL implementation has been developed. Using it turned out to be inapplicable for the MCT project, since it presupposes a patched version of the MySQL server. Servers running this version of MySQL are situated at CERN and are in principle accessible from Munich, but utilizing them would mean storing data far away from the cosmic ray facility.

For the conditions database, an Athena wide solution might be NOVA. NOVA accommodates the fact that the ATLAS data output will exhaust present storage capabilities by far. It is an approach to store data decentrally, such that the user does not need to know where his data is stored physically. NOVA is being developed in the context of a world wide grid computing.

At the moment, very simple solutions for IOV and conditions databases are implemented (respectively deployed) for MCT. Considering the low complexity of the experiment and the low amount of data involved, they seem completely appropriate. In addition to that, they are easy to maintain.

For the IOV database, an alternative to the MySQL approach actually exists in the Athena repository. It was designed to be a test- and debugging database but turned out to be sufficient for MCT and therefore is used. Although it is just used and was not developed in the context of this diploma thesis, it is still described here in more detail, since the implementation of the conditions database was inspired from this.

It should be noted once again, that Athena's dependency on external technology is extremely loose, and therefore the simple ASCII solutions described below could easily be exchanged for more sophisticated alternatives.[7]

[7] A problem still, although not unsolvable, would be to move existing data itself to the new technology.

4.3.2 Concept and Design

Both databases exploit the capabilities of file systems to organize data in folders, so-called subdirectories.

Interval of Validity Database

The IOV database maps (run, event)-pairs to tags. A request is accompanied by the following information,

- a (run, event)-pair,
- a unique identification of the data objects class (CLID) and
- a unique identification of the data object itself (key).

The CLID is given mainly by an `integer` while the key is given by a `std::string`. In the **IOVASCIIDbSvc**, the CLID information marks a subdirectory that holds files named according to certain keys. Their location is chosen to be

$$\underbrace{\texttt{IOVDB}}_{\text{root dir}} / \underbrace{\texttt{CLID}}_{\text{e. g. } 4144} / \underbrace{\texttt{key}}_{\text{e. g. } 1\_0\_0\_0\_0}$$

(In this example, `4144` is the class identification number of a transformations that is computed as described in section 4.2.1 and `1_0_0_0_0` is the `std::string`-representation of the unique identifier of the test stand upper chamber.) These files contain, line by line, the interval of validity information: four numbers marking beginning and end of the interval and the corresponding tag.

Conditions Database

The conditions database utilized in MCT uses the Unix file system, too. The design is analogous to the IOV database above, except for some details. When creating a transient representation of an object, the converters interpret contents of one file. (The **IOVASCIIDbSvc** interprets one line of a file only.) The location of this file needs to be uniquely defined by the following set of information:

- a unique identification of the data object class (CLID),
- a unique tag (provided by the **IOVSvc**) and
- a unique identification of the data object itself (key).

This information is delivered by the **IOVSvc** that triggers the converter call. As the reader can see, the key- and CLID-information is redundant, since it is evaluated by each conditions database and IOV database. This, on the other hand, might lighten switching to an alternative conditions database due to compatibility reasons.

The designated file, as defined by the information above, is found via the following Unix file path:

$$\underbrace{\texttt{CondDB}}_{\text{root dir}} / \underbrace{\texttt{CLID}}_{\text{e. g. DeltaTransform}} / \underbrace{\texttt{tag}}_{\text{e. g. ChamberAlignment/my\_alignment}} / \underbrace{\texttt{key}}_{\text{e. g. } 1\_0\_0\_0\_0}$$

The difference to the IOV database described above is that the CLID information is not interpreted as a pure number, but as the class name that lies underneath: `DeltaTransform` has assigned CLID 4144.[8]

4.3.3 Implementation

This section contains implementation details for the databases. As the utilized IOV database is not part of the MCT project, it is not described here. The terminology introduced in section 2.4.2, where the interval of validity service was introduced, is used here.

Conditions Database

Once a converter is triggered in the IOV context, its

```
createObj(IOpaqueAddress* pA, DataObject*& pO)
```

method is invoked. `pA` essentially is the information that is stored in the proxy of an object (the term *proxy* was introduced in 2.4.2). It is decoded by the converters to achieve the tag and key,

```
std::string key = *(pA->par());
std::string tag = *(pA->par()+1);
```

The CLID need not to be retrieved explicitly. It is known by the converter itself, via its own `classID()`-method. (A converter only serves for one data class.)

Via interaction with its conversion service, the converter retrieves a pointer to a filestream object by invoking its

```
std::istream& file = m_cnvSvc->inFile( tag, this, key );
```

(`m_cnvSvc` holds a pointer to the conversion service.) In this method, the conversion service locates the file that contains the information necessary for the converter to create the transient representation of an object uniquely defined by tag and key, just as described in the design section of this chapter. The conversion service retrieves the CLID by calling the `classID()`-method of the converter itself. (The C++ keyword `this` indicates a pointer to the object itself. This way, the conversion service knows which converter triggers the `inFile()`-call.)

Finally, the converter can evaluate the data coming from the filestream and create a new instance of the desired object. This is done via stream interpretation, as in the case of chamber alignments,

```
file >> x >> y >> z >> phi >> theta >> psi;.
```

In this example, `file` holds the reference to the filestream object. One line in the file contains six parameters, namely three translation- and three rotation parameters (Euler angles). These are streamed into their variables `x`, `y`, `z`, `phi`, `theta` and `psi`.

Once the desired object is created, `pO` is assigned to point to this object and is registered with **StoreGate** by

```
pO = SG::asStorable( pointerToNewObject );
```

With this, the conversion process ends.

[8] It is helpful if the tag contains more, human-readable information of the underlying event sample, such as run number, date of calibration and raw data file name. A more realistic tag would be `ChamberAlignment/1042457993_run_BOS_4C_16_full_setup_sc_stable_0.MT_Rawdata`.

4.4 Accessing Calibration Data from inside MCT

In this chapter, access to the two types of calibration data, drift spectra information and chamber alignment, is described. Both types differs in many ways, such that two different approaches needed to be implemented. Chamber alignment information is integrated in the GeoModel tree, while spectra information is made available via an additional instance, the **MuonCTCondManager** (the conditions data manager).

4.4.1 Problem Definition: Different Views of Time Varying Data

Accessing calibration data from inside MCT is a crucial part. The main goal of calibration data access is to keep it as transparent to the programmer[9] as possible. Whoever needs to access time varying data should not be aware of it changing behind the scenes. This approach benefits from the assumption that time varying data does not vary during one event. By this, it is sufficient to update data only at the beginning of an event.

The integration of time varying data with a special emphasis on the alternative views of the data is subject of the following sections.

Access to Drift Spectra Information

There are many ways on accessing drift spectra information in the MCT context. Of course, one wants to prevent the programmer from interacting with the **IOVSvc** directly, since this would imply unnecessary overhead. Instead, the idea is to provide a dedicated instance, the **MuonCTCondManager**, that manages drift time spectra information (and possibly further data in the future). The user simply retrieves a pointer to the manager and asks for drift spectrum information of a certain tube.

Access to Geometry Information

The programmer does not access the GeoModel directly, even though it would be possible. As the GeoModel is designed to be lightweight with respect to memory consumption, it is complicated to access and rather slow. E. g., as discussed in previous sections, one has to traverse the whole tree from top to bottom, in order to get the absolute transformation of a leaf node. This task should not be left to the programmer. In addition to that, the GeoModel offers a huge variety of information that is uninteresting to the programmer in usual cases, such as the precise position information of support structures.

Instead, MCT introduces a new object, the **DetDescrManager**. This manager provides the programmer with a new view of the setup, so-called detector elements (**MuonCTDetectorElement**). In the current implementation, scintillator layers, drift chambers and streamer tube layers each correspond to one **MuonCTDetectorElement** (or rather the complete implementation of this base class, namely **ScintiDetectorElement**, **MDTDetectorElement** and **StreamerDetectorElement**). Instances of these objects have a one-to-one correspondence to the real world detector elements. They can answer manifold, subsystem-specific requests, such as the individual positions of individual readout channels (e. g. tubes in the case of MDT detector elements). Figure 4.4 gives an idea of the programmers view of the setup geometry. Changes in the geometry have to be made available within this view of the detector.

[9] A *programmer*, in this case, is somebody who develops software that uses the concepts described here, not the developer of these concepts.

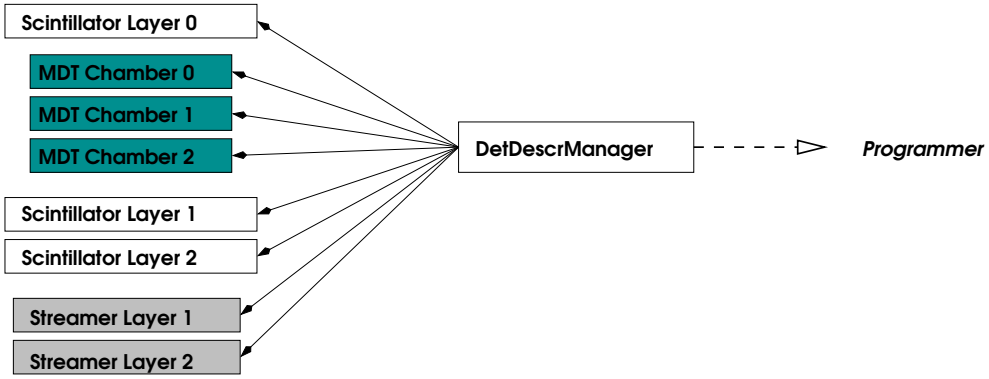

Figure 4.4: The programmers view of the detectors geometry: in this simplified view, only a couple of instances, each corresponding to one detector element, exist. They endue further methods that allow achieving further information, such as individual read out channel positions. Access to detector elements is mediated by the detector description manager.

4.4.2 Concept and Design

Drift Spectra Information

The programmer requests a pointer to an object that can answer questions on drift spectra. The **MuonCTCondManager** could be made an Athena tool or service. Or it could be a simple C++-object, that is stored in the detector store (which is a derivative of **StoreGate** with a different life time policy). The latter has been chosen for MCT, since it is consistent with the way that other manager objects, e. g. the identifier helpers, are made available to the programmer.

The way drift time spectrum information is held needs to be flexible: when gathering information about drift time spectra, one could in principle fill a histogram for *every single tube* and perform individual fits. But, due to statistical limitations, **T0Fitter** could be advised to group tubes and add up their histograms.[10] The data structure needs to accommodate this. The solution is rather simple and involves the `lower_bound()`-method of the standard template library container `std::map`. Details are given in the implementation section.

Another design choice that has to be made concerns the organization of time-dependent information. Given, the drift spectrum data for one single tube has changed. Should the IOV service only replace information for one single tube in memory, or for a set of tubes (multilayer, chamber, ...)? Using the data structure described in the implementation section, it turns out to be convenient to organize the data according to their chambers. This means that once the information for one single tube changes, a whole set of information corresponding to one chamber is replaced.

Although the IOV service mechanism would allow to access drift spectrum data in terms of single tubes, the approch made is justified. First, the calibration of drift spectra is usually performed in a single, dedicated job, leading to a set of drift spectrum information for a whole test stand setup. This way, it is excluded that drift spectrum information changes in between. Second, it is much more convenient for the manager of the IOV database to refer to subsets of

[10] Usually, this is not necessary when evaluating data of one chamber of the series test - the statistic is high enough to fit individual tubes.

drift spectrum information than to individual tubes.

Geometry Information

No canonical solution exists on how time varying data should be integrated in the geometry description of the test facility. With GeoModel and **DetDescManager** respectively its **MuonCTDetectorElement**s (from now referred to as the DetDescr), already two instances exist that provide information of the geometry. They have different purposes:

- *GeoModel* is supposed to be the single authorized source of geometry information, for dead material as iron absorbers as well as for sensitive detector components as MDT chambers.
- In *DetDescr*, the **DetDescrManager** usually interprets the GeoModel in the beginning of a job in order to gain information about the detector elements. Afterwards, clients can request pointers to **MuonCTDetectorElement**s, which contain further information of the specific read out systems.

Introduction of the IOV mechanism should take place on the GeoModel side in order to have consistent descriptions of the geometry. The DetDescr side on the other hand needs to be informed immediately of geometry changes in order to be able to update the **MuonCTDetectorElements**.

Fortunately, the IOV service already foresees such a constellation and offers the concept of call-back functions. The IOV service allows registering functions with an object that is updated via the IOV mechanism. Whenever such an object changes in memory, registered call-back functions are invoked. In addition to that, the IOV service allows registering call-back functions with already registered call-back functions. This later turns out to be the connecting interface between the GeoModel- and the DetDesc-side: a function on the GeoModel side, triggered by the IOV service, triggers a function on the DetDescr side.

The concept of IOV integration on the two geometry description sides is as follows in MCT. On the *GeoModel side*, a **IOVGeoAlignableTransform** replaces the **GeoAlignableTransform** of chambers (as in figure 2.3). This newly introduced class holds pointers to two transformations (instead of only one), one reflecting the nominal position of a volume. The other transformation is a small correction on the nominal position. (This correction might in fact be the result of the calculations described in section 4.2.1, the static chamber misalignment.) The latter is registered with the IOV service in **IOVGeoAlignableTransform**s constructor[11]. For that, the position of the volume is held up-to-date automatically. On the *DetDescr side*, the idea is to register call back functions whenever a **IOVGeoAlignableTransform** is seen in the GeoModel, once it is interpreted at the beginning of a job. The calculation of the positions of leaf nodes needs then to be re-performed whenever the local position information of a volume, that lies in the path from the root node to the leaf node, changes.

4.4.3 Implementation

Drift Spectra Information: Data Structure

The information belonging to one single tube is described by a **TDCSpectCond** object. It has several methods comparable to `t0()` or `t0Err()` which deliver information about t_0-value and its error.

[11] A constructor of a class is a method that is called automatically whenever a new instance of this class is created.

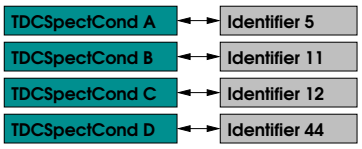

Figure 4.5: Example of a **DataIdMap** filled with four different sets of spectrum information, A-D. The interpretation of this constellation is given in the text.

TDCSpectCond are organized in an extension of a `std::map`, a **DataIdMap**. A `DataIdMap<TDCSpectCond>` is in principle a `std::map` instantiated with an**Identifier** (an object that uniquely identifies units, as e. g. single tubes, in the test stand setup) and a **TDCSpectCond** as its template arguments,

```
std::map< Identifier, TDCSpectCond >.
```

In addition to the STL (standard template library) container, the **DataIdMap** accommodates several **StoreGate**-specific issues that deal with ownership. As mentioned, **StoreGate** takes over ownership for objects placed in it. **DataIdMap** resides in the MCT sub package MuonCT-Conditions and is inspired by the **DataVector**, which was introduced by Athena developers.

The data structure above is sufficient even for the case that spectrum information was gathered for groups of tubes instead of single tubes. Grouping of tube spectra is inevitable in case of low statistics, as stated above. This depends on how the data organized in the map is *interpreted*. Figure 4.5 illustrates the interpretation. Here, the **DataIdMap** is filled with four different sets of spectrum data, A-D, that corresponds to Identifiers 5, 11, 12 and 44.[12] Instead of requesting objects from this map by using the `find()`-method of a `std::map`, one uses the `lower_bound()`-method! `find()` would only return a result if the argument is either 5, 11, 12 or 44. On the contrary, `lower_bound()` returns a **TDCSpectCond** for every argument that is smaller or equal 44. The definition of `lower_bound()` is as follows: `lower_bound()` returns the value of the biggest index, that is smaller or equal the one given as its argument. With this, the example given in the figure is interpreted as in the following table,

Identifier	**TDCSpectCond**
0..5	A
6..11	B
12	C
13..44	D

The described approach benefits from the fact, that **Identifiers** have a *defined order*, and insertions within a `std::map` are automatically ordered.

Drift Spectra Information: Data Organization in the IOV Context

As described in the design section, the goal is to have one set of data corresponding to one chamber. In order to achieve this, **MDTTDCSpectCollection** is introduced. **MDTTDC-SpectCollection** is an extension of **DataIdMap** in the sense, that it also derives from an

[12] An **Identifier** is slightly more than just a number, but for the purpose of this example, it can be considered as one.

Athena class **Identifiable**, which makes it, as the name suggests, identifiable. With this, **MDTTDCSpectCollection** can answer to identification requests in its `identify()` method. For the TDC spectrum informations of the test stand, three **MDTTDCSpectCollection**s, one for each chamber, reside in memory. They are identified by their corresponding chamber's id.

Drift Spectra Information: MuonCTCondManager and IOV Service

In its constructor, **MuonCTCondManager** registers in a loop **DataHandles** to **MDTTDCSpectCollection**s,

```
m_IOVSvc->regHandle( m_tdcSpectCollectionMap[chamberID], chamberIdString);.
```

(`m_IOVSvc` points to the IOV service, `chamberID` is the variable that is looped over and `chamberIdString` is the key of the drift spectrum information of one chamber. `m_tdcSpectCollectionMap[chamberID]` is a reference to an object of type **MDTTDCSpectCollection**). After that, the IOV service ensures that all **MDTTDCSpectCollection**s are held up-to-date.

The **MuonCTCondManager** has a single method that allows clients to gain access to spectrum information,

```
TDCSpectCond* tdcSpectCond( Identifier id );
```

The client only needs to know the **Identifier** of a single tube in order to get a pointer to the **TDCSpectCond** object.

Geometry Information: GeoModel side

The newly introduced **IOVGeoAlignableTransform**, residing in MuonCTGeoModel, has three essential member functions in its interface,

- `HepTransform3D getTransform()` returns the correct local position information of the regarding volume,
- `const HepTransform3D* getDelta()` returns the correction of the position with respect to the nominal position and
- `virtual StatusCode clearDelta()` performs operations necessary when the small correction is updated.

A **HepTransform3D** is the common CLHEP (see [15]) representation of an arbitrary transformation consisting of a translation and a rotation. **DeltaTransform** is a **HepTransform3D** and has, in addition to that, a unique class identification, CLID, which is necessary for any **StoreGate** operation.

In its constructor, **IOVGeoAlignableTransform** not only registers a **DataHandle** to a **DeltaTransform** with the IOV service. With the line

```
m_iovSvc->regFcn(&IOVGeoAlignableTransform::clearDelta, this, m_delta, key);
```

two things are achieved. First, the class member `m_delta`, which is essentially a pointer to a **DeltaTransform**, is registered for automatic update with the IOV service (which `m_iovSvc` points to). Second, a member function of the class **IOVGeoAlignableTransform**, namely

`clearDelta()`, is registered with the IOV service for automatic call-back. (The C++ keyword `this` indicates, that it is the method `clearDelta()` belonging to the instance of **IOVGeoAlignableTransform** performing the registration with the IOV service.)

With the registration with the IOV service already arranged in the constructor, subsequent calls of `getTransform()` will return the correct **HepTransform3D**, representing the local transformation of the regarding volume.[13]

With this, the automatic update of geometry information in principle is accomplished on the GeoModel side. One problem remains, though. It was mentioned in the GeoModel introduction (section 2.4.1), that **GeoFulPhysVol**s (unlike **GeoPhysVol**s) keep cached copies of their global positions. For that, `clearDelta()` provides a way to inform nodes below. The mechanism involves a **GeoClearAbsPosAction** (which is a **GeoNodeAction**) in `clearDelta()`, that traverses through the tree and invalidates all affected **GeoFulPhysVol** position caches.

Geometry Information: Detector Description Side

In the initialize phase of a job, the **MuonCTDetDescrManager** sends a **GeoNodeAction** through the GeoModel tree that looks for physical volumes corresponding to an MDT chamber, a scintillator or a streamer layer. The results of this **GeoNodeAction**, which are pointers to **GeoFulPhysVols**, are used as arguments for the constructors of **MDTDetectorElement**, **ScintiDetectorElement** or **StreamerDetectorElement**, respectively. For each **GeoFulPhysVols** a **MuonCTDetectorElement** is created which leads to a constellation as shown in figure 4.4.

Given the pointer to the GeoModel physical volume counterparts in their constructor, **MDTDetectorElement**, **ScintiDetectorElement** or **StreamerDetectorElement** now can send **GetGeoTransformAction**s (which are implementations of **GeoNodeAction**s) through the GeoModel tree and collect **GeoTransform**s that lie on the path in between the root node and their according physical volume. The transformations then can be connected in order to retrieve the global position of the detector element (or its representing physical volume). This global transformation is cached. In fact, this calculation does not take place in **MDTDetectorElement**, **ScintiDetectorElement** or **StreamerDetectorElement**, but in their common base class, **MuonCTDetectorElement**. (To be precise, this is done in **MuonCTDetectorElement**s constructor.)

Once a **GetGeoTransformAction** is applied on a node of a tree, it returns pointers to all **GeoTransform**s that lie in between the node where it is applied and the root node of the tree. Since **IOVGeoAlignableTransforms** are derived from **GeoTransform**s, **GetGeoTransformAction** does not see a difference between them and **GeoTransforms**. In **MuonCTDetectorElement**, where all found transformations are multiplied, a test is performed to decide whether a transformation is a **GeoTransforms** or a **IOVGeoAlignableTransforms**,[14]

```
const IOVGeoAlignableTransform* ciovgt =
dynamic_cast< const IOVGeoAlignableTransform* >(gt);
if(ciovgt!=0) { ...
```

[13] While small corrections of the nominal positions are read from a database, namely the conditions database in interaction with the IOV database, it should be noted that the nominal positions themselves are still hardcoded in MCT. In the future, the whole geometry should be dynamically built from a database.

[14] Usually, a `dynamic_cast` should be avoided in clean C++ programming. In order to do so, small changes need to be applied to the **GeoNodeAction** base class. This might happten, in case the presented **IOVGeoAlignableTransform**-concept is accepted by the Athena developers.

(`gt` is meant to be one of the **GeoTransform**s found by the **GetGeoTransformAction**.) Once this test turned out to be positive, **MuonCTDetectorElement** can register its own `update()`-method with the `clearDelta()`-method of the **IOVGeoAlignableTransform**,

```
m_iovSvc->regFcn(&IOVGeoAlignableTransform::clearDelta, iovgt,
                 &MuonCTDetectorElement::update, this);
```

(This registers the `update()`-method with the already registered `clearDelta()`-method, while `iovgt` is the pointer to the **IOVGeoAlignableTransform**.) In `update()`, the global position cache of the **MuonCTDetectorElement** is invalidated and a new calculation triggered.

This way, the global position cache of the **MuonCTDetectorElement** (and, deriving from this base class, **MDTDetectorElement**, **ScintiDetectorElement** and **StreamerDetectorElement**, too) always hold global position information that is up-to-date with respect to the GeoModel information. In addition, the position information of single read out channels that is made available from **MDTDetectorElement** via two methods,

```
HepTransform3D globalTubeTransform(const Identifier& id) const;
```

or

```
HepTransform3D localTubeTransform(const Identifier& id) const;,
```

which return the global or local position information of individually identified tube transformations, is updated correctly.

Chapter 5

Wire Position Measurements

One of the design goals of the Munich ATLAS Cosmic Test Facility is the determination of individual wire positions. MTOffline, the offline reconstruction package developed by Oliver Kortner and Felix Rauscher, accomplishes this goal within a precision of 8.3 μm for the measured displacement of wires in y- and 27 μm in z-direction (see [16]). Comparison of MCT, MTOffline and tomograph measurements is a touchstone for the MCT project and is the subject of this chapter. The procedure of determining individual wire positions is closely related to the chamber position determination discussed in 4.2.1.

The analysis is based on an n-tuple produced by an MCT Athena reconstruction job that evaluates raw data and performs track fits. The data file containing the results of [16] was made available for comparison purposes by Felix Rauscher.

5.1 Data Sample and Event Selection

The sample of muon events used for this measurement is a set of 6.9 million events, recorded by the Munich Cosmic Ray Measurement Facility in January 2003. During the data-taking period, evaluation of RasNiK data shows no significant changes in the geometry parameters (the RasNiK calibration data is not yet considered in MCT). The same set of data is the basis for the results of MTOffline, presented in [16]. This allows a comparison of the two software packages, MTOffline and MCT, later in this chapter.

The test chamber has been scanned with an X-ray tomograph at CERN. As a result, the wire positions are known at a position of 30 cm distance to the two ends of the chamber (i. e., near the readout side of individual tubes and its opposite). The precision of this measurement is 2 μm.

Cuts on Events

The MCT n-tuple contains track parameters for 75% of all recorded events. Most of the loss of 25% is due to ambigous scintillator information. An event is dropped, if the number of signal carrying clusters in the scintillator layers is not equal to one. In such a case, no fit for the xz-projection is performed and therefore no signal run time (see 3.2.1) corrections can be applied.

Further cuts are applied on the n-tuple, which conform to the cuts of the analysis presented in [16].

- The local track parameters measured by the reference chambers must match roughly. The slope difference, $|m_{\text{ref, up}} - m_{\text{ref, low}}|$, needs to be less then 15 mrad while the intercept difference, $|b_{\text{ref, up}} - b_{\text{ref, low}}|$, needs to be less than 4 mm.

- The wire position is determined for two sections along the wire, each of 1 m length. Only muons that pass the test chamber in one of these two sections are accepted for the position measurement. Due to an asymmetric trigger setup, the fractions of muons passing through each of these intervals is not identical.

The efficiency of these cuts is given in the following table.

cut	number of events after cut	efficiency
all events	$5.1 \cdot 10^6$	100%
track match	$4.7 \cdot 10^6$	92%
wire section (back/front)	$1.7 \cdot 10^6 / 1.3 \cdot 10^6$	32%/25%

Cuts on Hits in the Test Chamber

With the track parameters of the global fit of the two reference chambers, a radius prediction for hits in the test chamber is calculated. Among all hits in the test chamber, a maximum of 6 is selected, one hit per layer.

- A hit in the test chamber is required to have a deviation between the measured radius and the predicted radius, $|r_{\text{drift}} - r_{\text{ref}}|$, of less then 1.5 mm.

- The radius prediction for tubes near the edge of the setup can, due to the geometry of the setup, be performed only with tracks that have a limited range of angle of incidence in the yz plane. This limits the resolution. Therefore, the 16 outer tubes of each layer of the test chamber are not considered for the analysis.

- Four tubes of the test chamber show less than 20% of the hits of their direct neighbour tubes. These tubes are omitted.

No local fit is performed for the test chamber. A cut that compares hits in the reference chamber with its local trackfit, is applied in MTOffline but not in MCT.

5.2 Wire Displacements

For each selected tube, the measured radius r_{drift} in the test chamber is compared to the predicted radius r_{ref} from the reference chambers based on a global fit. This leads to a slope-dependent offset Δy in y direction, as illustrated in figure 5.1. Δy corresponds to the y shift estimated with the help of one single muon reference track. In case there is no z shift δz at all, Δy would reflect the real y shift δy, independently of the slope of the reference muon track. If δz is not equal to zero, Δy varies with the slope, and $\Delta y = \delta y$ is only true for $m = 0$. Therefore, the values δy and δz are obtained by a linear regression performed on all $(m, \Delta y)$-pairs,

$$\Delta y(m) = \delta z + \delta y \cdot m.$$

Due to the limited range of the angle of incidence, the resolution for the y-shift measurement is expected to be better than for the z-shift measurement.

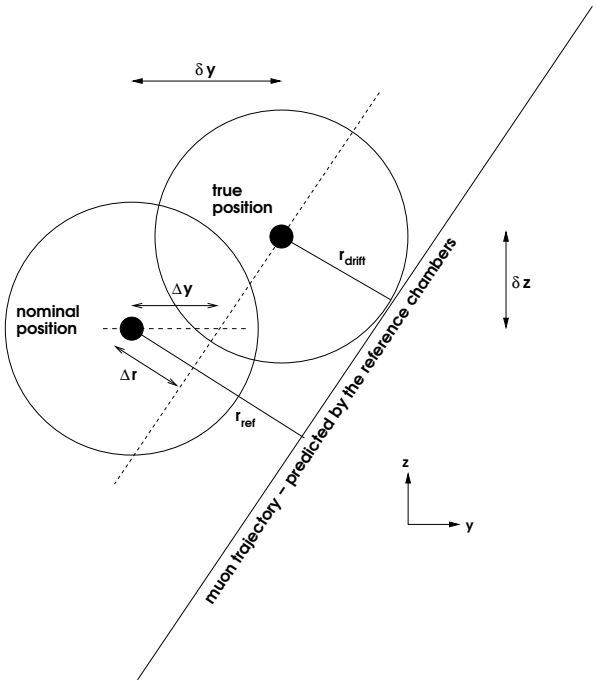

Figure 5.1: For each selected hit, the slope m of the muon track predicted by the reference chambers and the measured offset in y, Δy, is saved. A linear regression performed on the selected $(m, \Delta y)$-pairs leads to the real displacements of the wire, δy and δz.

5.2.1 Displacements in Y Direction

The result of the determination of wire displacements in the y-direction is presented in figure 5.2. It is compared to the X-ray tomograph results, which have a precision of 2 μm. The distributions of the difference of the MCT result and the tomograph result have widths of 10.5 μm and 12.5 μm, respectively.

5.2.2 Displacements in Z Direction

Figure 5.3 shows the results for measurements in the z-direction. The resolution is significantly lower than for the y shifts. The distributions of the difference of the MCT result and the tomograph result have widths of 24.0 μm and 26.0 μm, respectively. The two wires which are known to deviate from their nominal position by more then 300 μm are clearly recognized.

5.3 Comparison with MTOffline

5.3.1 Comparison of the Results

Figures 5.4 and 5.5 show the results for the measurements of MTOffline. The distributions of the difference between the MTOffline result and the tomograph scan for the front and the back side have a maximum width of 25.5 μm for the z direction and 8.5μm for the y direction. The corresponding widths achieved with MCT are larger by 4.0 μm in y and 0.5 μm in z. Both

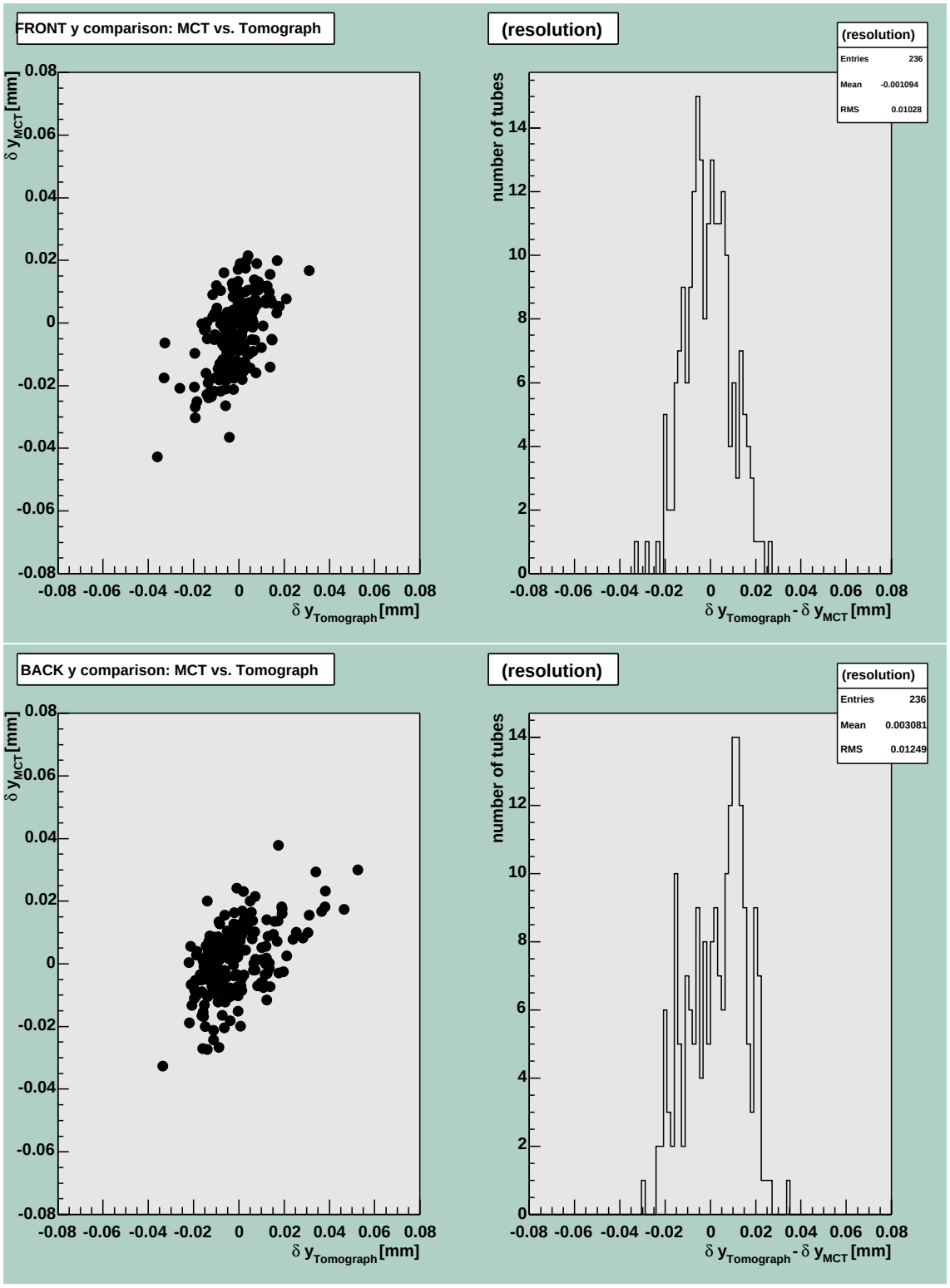

Figure 5.2: Comparison of the y displacements, determined for two sections of one meter length near the ends of the test chamber. The left side shows the displacements measured by MCT on the y-axis, and the tomograph measuerements on the x-axis for each side. The right side shows the distribution of the difference of the measurements. The maximum of the precision widths is 12.5 μm.

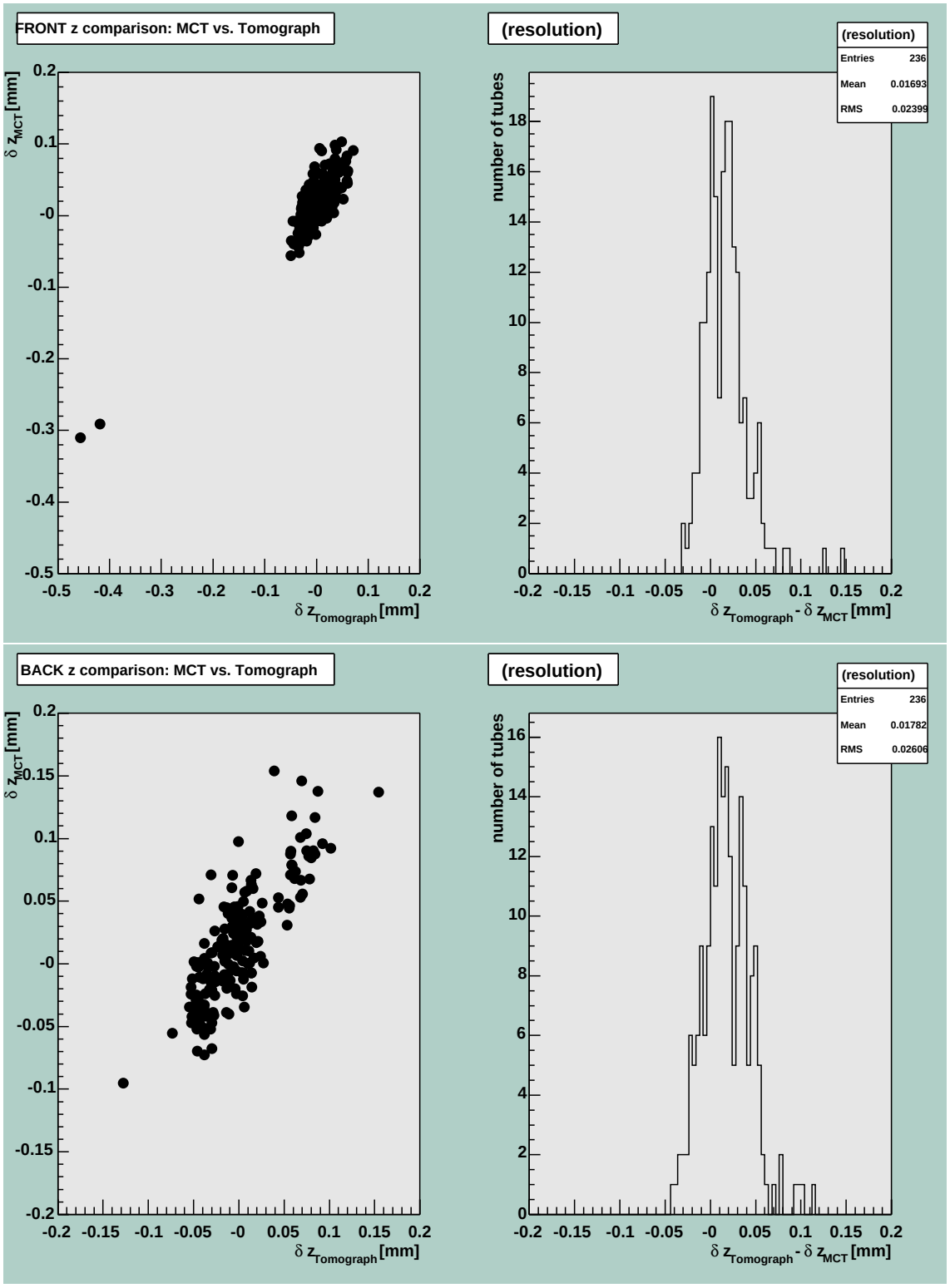

Figure 5.3: Comparison of the z displacements, determined for two sections of one meter length near the ends of the test chamber. The left side shows the displacements measured by MCT on the y-axis and the tomograph measuerements on the x-axis for each side. The right side shows the distribution of the difference of the measurements. The maximum of the precision widths is 26.0 μm.

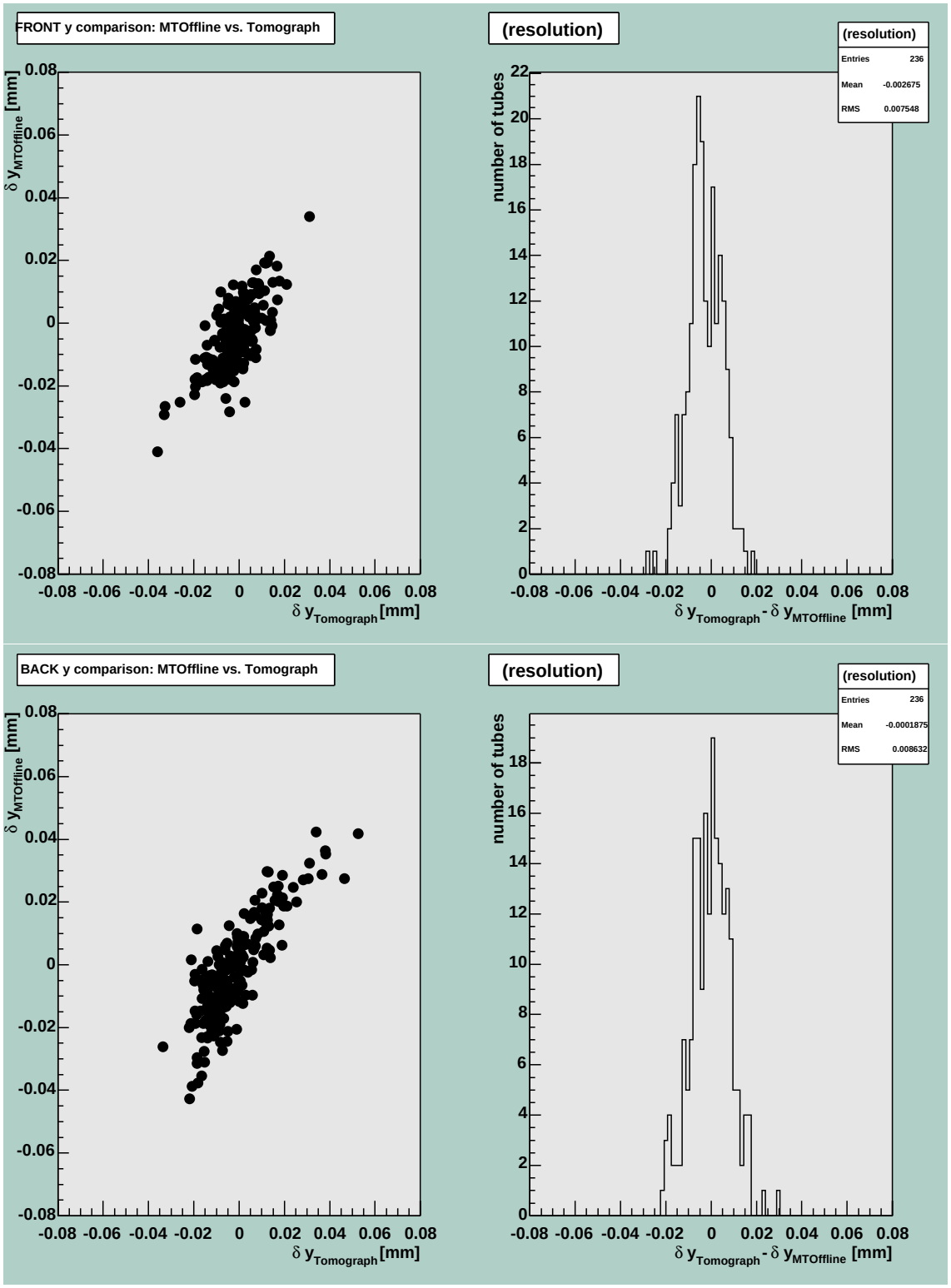

Figure 5.4: Comparison of the y displacements for the results of MTOffline. The distributions of the difference between MTOffline and tomograph results have a maximum width of 8.5μm.

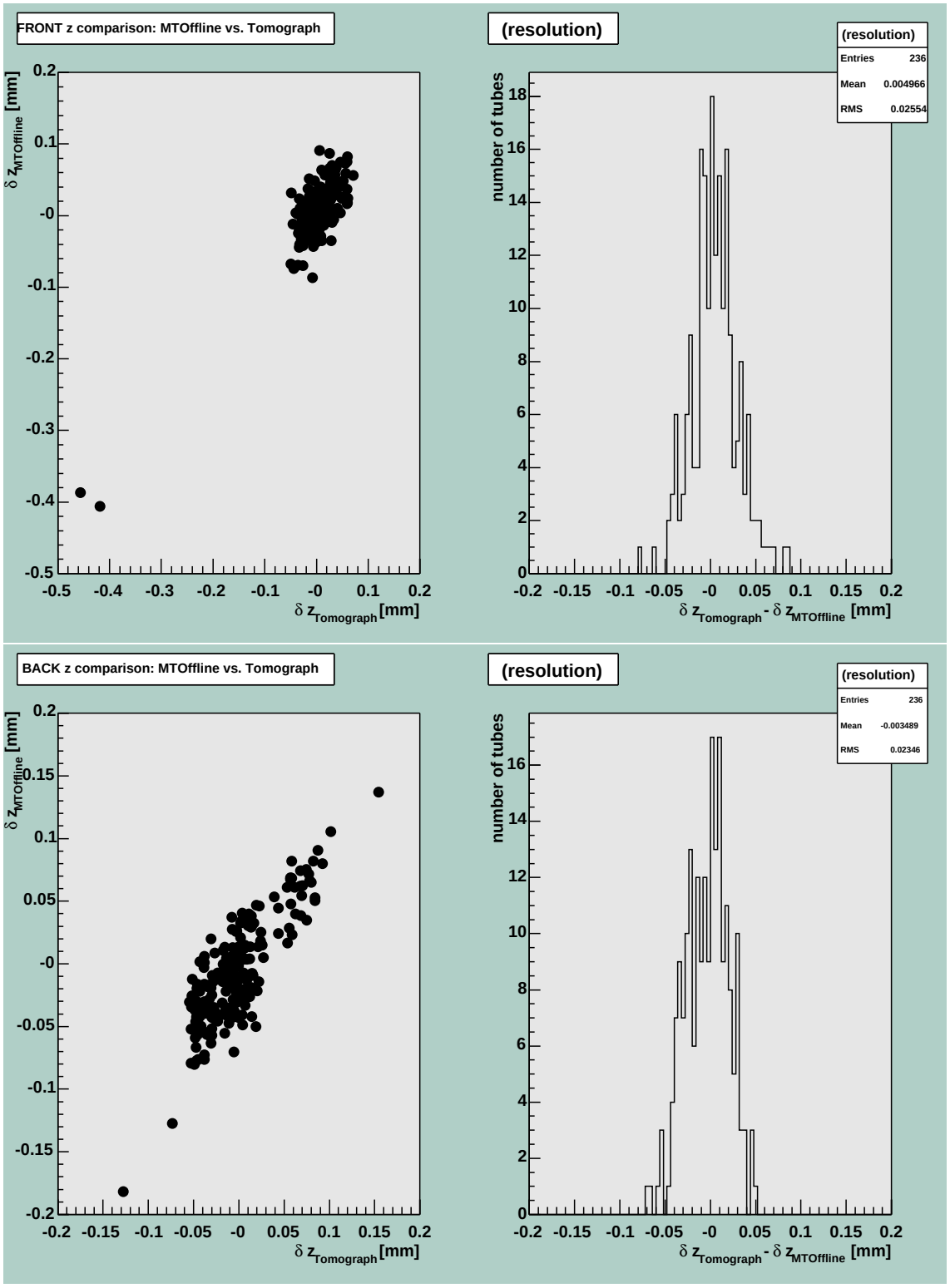

Figure 5.5: Comparison of the z displacements for the results of MTOffline. On the upper graph, the scale is modified with respect to the other z measurements in order to show to outlier that are detected with MTOffline. The distributions of the difference between MTOffline and tomograph have a maximum width of 25.5 μm.

program packages were able to recognize outliers in the wire displacement of the front side of the test chamber.

5.3.2 Comparison of the Methods

The development of the MCT reconstruction algorithms was inspired by the program package MTOffline. However, three main differences exist between the two packages.

- The procedure of determining the transformations that describe the deviations of the *overall* chamber positions with respect to their nominal position is identical for MCT and MTOffline. However, there is a difference in the application of these transformations: in MCT, as described in section 4.4, this information is inserted in the geometry description tree, and a new trackfit is performed with the aligned geometry. In MTOffline, these transformations are applied on the local track fit parameters. No further fit is performed here. As a consequence, the new alignment may influence the pattern recognition in MCT while it does not in MTOffline.

- Due to the last point, MCT has the ability to perform a global track *based on the hits of both reference chambers* with the aligned geometry. This is impossible in MTOffline. Therefore, radius prediction in MCT is based on the *global track fit*, while in MTOffline, it is based on the weighted average of the two predictions of the local track fits.

- MTOffline considers deviations of the nominal positions of the wires of the reference chambers, while MCT does not. Since these are distributed Gaussian with the production accouracy of $10\,\mu$m, this should not influence the result too much. Determination of individual wire displacements takes into account a large number of tracks, based on drift circles seen by a large number of tubes of the reference chambers. The effect of misaligned wires of the reference chambers should be negligible in the average.

Two main differences exist concerning the n-tuple based analysis

- The energy cut of 2.5 MeV that is applied in MTOffline analysis is not used in MCT. This cut decreases errors due to multiple scattering of the muons. On the other hand, it limits the statistic. Both effects balance.

- In the analysis based on the MTOffline n-tuple, the deviation of whole wire planes from their nominal z position is determined. This is taken into account for the determination of the y displacement. This is not consireded in the presented analysis.

The integration of deviations of the overall positions of the chambers into the geometry tree is made possible by the fact that Athena offers a comfortable management of time varying data. This allows to perform global track fits based on a misaligned geometry.

By now, MCT does not implement its own determination of the drift-time relation. While all other calibration constants are determined by MCT itself, the drift-time relation is taken from MTOffline and is identical to the one used for the MTOffline analysis.

5.4 Conclusions

The resulting wire position measurements are in general in good agreement with the tomograph results. A few discrepancies remain, which need further investigation. One result, the determination of the z deviation of the front side, even shows a slightly better resolution than the

MTOffline measurement. Considering the resolution of the X-ray tomograph, 2 μm, the resulting resolution for the wire position measurement for MCT is 26.0 μm for the z and 12.5 μm for the y direction. Even though these are still in the region of the production accuracy of the chambers, 10 μm, MCT has prooven to be able to recognize outliers.

Chapter 6

Summary

The MCT project was launched as an alternative software solution for the Munich Cosmic Ray Test Facility in order to gain experience with the young Athena Framework, to benefit from many of its features and to collaborate with an international community of developers by giving feetback and sharing source code.

Within a bit more than one year[1], it developed towards a broad software solution for the Munich Cosmic Ray Measurement Facility, covering big parts of the data processing steps necessary for reconstucting muon trajectorys. Among these are the interpretation of raw test stand data, calibration of the detector (fits of single drift spectra, determination of the deviation of the chambers from ther nominal position) and the determination of muon track parameters utilizing numerical and analytical minimization schemes. In addition to that, MCT provides a simulation based on Geant4, whose boundary conditions are derived consitently from one single source of geometry information, which itself is the starting point for track reconstruction, too.

The adoption of Athena not only accelerated the development, it faciliated clear program structures and readable code. It furthermore made MCT a comparative user-friendly, but profoundly flexible program package. Considering the geometry conversion described, it also showed that the adoption of Athena for complex experiments is limited, still.

The highly sensitive determination of the wire positions demonstrated that the muon track parameters determined by MCT already are very precise and that the predictions are qualitatively in good agreement with the MTOffline program package, even though some quantitative improvements are possible.

Parts of the work for the MCT project was the development of infrastructure that eases the commissioning of further program parts. With the introduced IOV service and its two databases, further calibration data, as for example the drift-time relation or RasNiK geometry data, can implemented without much effort.

[1] In the context of the doctoral thesis of Philipp Schieferdecker and this diploma thesis.

Chapter 7

Zusammenfassung

Das MCT Projekt wurde, als eine alternative Softwarelösung für den Münchener Höhenstrahlmessstand, begonnen, um Erfahrung mit dem noch jungen Athena Software Framework zu sammeln, um von den vielen Vorzügen Athenas zu profitieren und um mit der internationalen Entwicklergemeinde durch Rückmeldung und Austausch von Quellcode zusammen zu arbeiten.

Es hat sich innerhalb eines Zeitraums von etwas mehr als einem Jahr[1] zu einer umfassenden Softwarelösung für den Münchener Höhenstrahlmesstand entwickelt, welche bereits einen Großteil der zur Rekonstruktion von Myonspuren notwendigen Datenverarbeitungsschritte enthält. Dazu zählen die Interpretation von Rohdaten aus dem Teststand, Kalibration des Detektors (Fit von einzelne Driftzeitspektren, Bestimmung von Abweichungen der Kammerpositionen von ihrer Nominalposition) und Durchführung der Bestimmung der Spurparameter unter Anwendung analytischer beziehungsweise numerischer Minimierungsverfahren. Darüber hinaus enthält MCT eine Simulation auf Basis von Geant4, dessen Randbedingungen sich konsistent aus einer einzigen Quelle von Geometrieinformationen herleiten lassen, welche auch den Ausgangspunkt für die Rekonstruktion von Spurparametern bildet.

Der Einsatz von Athena hat nicht nur die Entwicklung beschleunigt, sondern auch klare Programmstrukturen und gut lesbaren Code gefördert. Zudem hat er ermöglicht, MCT zu einem vergleichsweise benutzerfreundlichen und dennoch hochgradig flexiblen Programmpaket werden zu lassen. Unter anderem am Beispiel der Geometriekonversion hat sich jedoch auch gezeigt, dass dem Einsatz von Athena für umfangreiche Detektoren derzeit noch Grenzen gesetzt sind.

Die hochsensible Drahtpositionsbestimmung hat gezeigt, dass die Spurparameter, die von MCT berechnet werden, bereits sehr präzise sind, und dass die Vorhersage der Drahtpositionen qualitativ mit der des Programmpaketes MTOffline übereinstimmt, wenngleich quantitativ sicher noch einige Verbesserungen möglich sind.

Ein Teil der Arbeit, die bisher an MCT geleistet wurde, ist das Bereitstellen von Infrastruktur, welche das zukünftige Einbinden von Programmteilen erleichtert. Mittels des vorgestellten Zusammenspiels des IOV service und der beiden dazugehörigen Datenbanken lassen sich mit vergleichtsweise geringen Aufwand weitere Kalibrationsdaten, wie etwa die Orts-Driftzeitbeziehung oder Geometriedaten aus dem RasNiK System, integrieren.

[1] Im Rahmen der Doktorarbeit von Philipp Schieferdecker sowie dieser Diplomarbeit.

Appendix A

Physical Streamer Digit Production

If in a simulated event a muon passes through the streamer tube layers (streamer tubes reside beneath the iron absorber, refer to figure 1.3 on page 8), the only information essentially stored is the unique identifier of the streamer bars readout channel. This section will describe physical process that need to be simulated when producing streamer digits.

Gaseous Detectors in the Proportional Mode

Streamer tube cells share several properties of the MDT chambers drift tubes. One streamer tube cell[1] is, as drift tubes, a detector based on the principle of gaseous detectors.

Drift tubes are operated in the so-called proportional mode. Charged particles passing through the gas volume cause atoms to ionize. The electric field causes the electrons to drift towards the anode wire, while the nuclei drift towards the tube wall. The electric field is distributed inhomogenious in the tube. Close to the wire, it is strong enough to impart the electron an energy high enough to ionize further atoms. This leads to electron avalanches near the wire. The burst influenced by the charge cloud is measured.

In the proportional mode, the anode signal is much higher than the energy loss of the initial particle but still proportional (approximate gain for ATLAS MDT chambers: 10^4).

The time difference between the initial ionization and the measurement of the influenced charge burst is called drift time. It changes with the radius, at which the inital particle passed by the anode wire. The radius resolution of a single tube is about $80\,\mu m$. For small radii, the resolution decreases due to the avalanche forming near the wire.

Gaseous Detectors in the Limited Streamer Mode

For streamer tubes, the electric field is increased. This causes the electron avalanches to grow even stronger and to form non-negligible space charge distributions, which shield the electric field of the anode wire. As a consequence, the electron-ion pair attraction is strong enough to recombine and thereby emit photons. The photons cause further atoms to ionize, which causes the charge cloud to grow in the opposite direction of the anode wire. This would, if the voltage is sufficient enough, lead to a breakdown of the gas. CO_2 as a quenching agent with a high photon absorption rate is added to the tube gas. This avoids the electric charge distribution

[1]The terminology might be confusing: a streamer tube groups eight streamer tube cells with a individual anode wire each. A MDT chamber groups 432 drift tubes with individual wires.

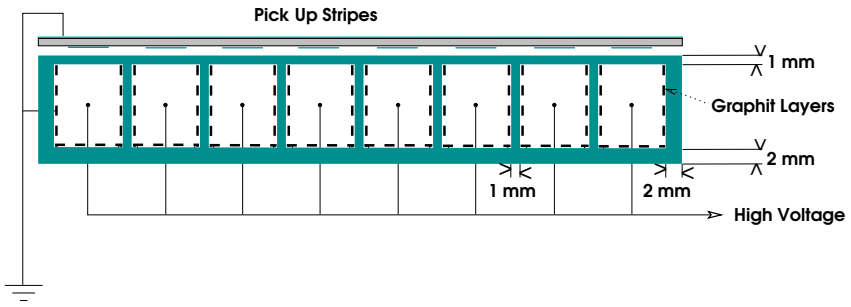

Figure A.1: Schematic view of one streamer tube with eight cells. (Thanks to Oliver Kortner for this picture!)

over the whole volume. The geometry of the tubes and the high resistance of the kathodes avoid a breakdown of the gas.

The streamer tubes utilized in the Munich Cosmic Facility have eight rectangular cells each, consisting of synthetics with a thin graphite layer on three pages of the cavities acting as cathode. Thin induction stripes stuck on the cells outside, opposite the cathodes, are used to record the electric signal influenced by the charge distribution inside the streamer tube cells. Figure A.1 illustrates the streamer tube layout.

Once a gas ionization process occurs, not only the pick-up stripe attached to the affected cell records a signal. A consideration based on a simple electrostatic model predicts the ratio of the influenced neighbor strip charge to the originating cell strip charge to be

$$\frac{Q_{\mathrm{i}+1}}{Q_{\mathrm{i}}} = \frac{\arctan\left(\frac{y+10\,\mathrm{mm}+\frac{b}{2}}{a_{\mathrm{q}}}\right) + \arctan\left(\frac{y+10\,\mathrm{mm}-\frac{b}{2}}{a_{\mathrm{q}}}\right)}{\arctan\left(\frac{y+\frac{b}{2}}{a_{\mathrm{q}}}\right) + \arctan\left(\frac{y-\frac{b}{2}}{a_{\mathrm{q}}}\right)},$$

where Q_{i} and $Q_{\mathrm{i}+1}$ denote the charges influenced at position y and $y + 10\,\mathrm{mm}$, measured by pickup strips of with width b. $10\,\mathrm{mm}$ is the spacial distance between two adjacent pickup strips. a_{q} is the distance of the initial charge causing the signal, it is assumed to reside directly underneath the strip at position y. An elaborate discussion can be found in [12].

Appendix B

Athena Components and Data Classes in MCT Sub Packages

The following sections describe all sub packages of MCT. Athena components and Data Classes that have a CLID assigned are explained if present. Reading the following pages does not substitute reading the source code, [9]! A, T, S and C are used to abbreviate **Algorithm**, **Tool**, **Service** and **Converter**, respectively.

MuonCTAlgs

Short Description

Implementation of helper **Algorithm**s for debugging or n-tuple production.

Athena Components

DigitMonitor (A) collects digit collections, either **MDTDigitCollection** (CLID 4103), **ScintillatorDigitCollection** (CLID 4104) or **StreamerDigitCollection** (CLID 4105) and utilizes the histogram service for histogram production.

DumpDigits (A) collects digit collections (**MDTDigitCollection** (CLID 4103), **ScintillatorDigitCollection** (CLID 4104) or **StreamerDigitCollection** (CLID 4105)) from **StoreGate** and dumps them on the screen.

DumpHits (A) collects hit collections (**MDTHitCollection** (CLID 4100), **ScintillatorHitCollection** (CLID 4101) or **StreamerHitCollection** (CLID 4102)) from **StoreGate** and dumps them on the screen.

MuonCTNTuple (A) collects digit, pattern, driftcircle and track collections (**MDTDigitCollection** (CLID 4103), **ScintillatorDigitCollection** (CLID 4104), **StreamerDigitCollection** (CLID 4102), **MDTDriftCircleCollection** (CLID 4111), **MDTPatternCollectio** (4113), **TrackCollection** (4115)) from **StoreGate** and utilizes the **NTupleSvc** for n-tuple production.

TrackFitMonitor (A) was intended to collect track information for performance comparison of different track fits but was not updated.

Data Classes

None.

MuonCTAthenaRoot

Short Description

Read/write converters for digits based on the Athena Root converter (**AthenaRootConverter**) base class.

Athena Components

MDTDigitAthenaRootCnv (C) is a read/write converter for **MDTDigitCollection**s (CLID 4103) using Root technology.

ScintiDigitAthenaRootCnv (C) is a read/write converter for **ScintiDigitCollection**s (CLID 4104) using Root technology.

StreamerDigitAthenaRootCnv (C) is a read/write converter for **StreamerDigitCollection**s (CLID 4105) using Root technology.

Data Classes

None.

MuonCTCalibAlgs

Short Description

Implementation of calibration algorithms. Calibration constants are needed for precise track fitting.

Athena Components

ChamberPosition (A) collects track collections (**TrackCollection** (CLID 4115)) of all chambers and uses local track fit information to determine small corrections to the nominal position of the reference chambers relative to the test chamber. The six parameters of the transformation are dumped to the screen and need to be inserted in the conditions database manually.

T0Fitter (A) collects drift times (**DriftTimeCollection** (CLID 4117)) and performs fits on both edges of the drift time spectra. The result is written to the conditions database utilizing the **MDTTDCSpectCollectionConv** from sub package MuonCTConditionsCnvSvc. It is possible to group tube spectra in order to increase statistics.

Data Classes

None.

MuonCTConditions

Short Description

MounCTConditions contains the data classes for the transient representation of the calibration constants. Usually, calibration constants are made available via the **IOVSvc**.

Athena Components

MCTCondMgrTool (T) is a tool utilized by the **GeoModelSvc**. Its only purpose is to instantiate the **MuonCTCondManager** (CLID 4133) and store it in the detector store.

Data Classes

DeltaTransform (CLID 4144) represents a six parameter transformation and is used by **IOVGeoAlignableTransform**. A converter exists in **MuonCTConditionsCnvSvc**. It is intended to be a small correction on the nominal position.

MDTTDCSpectCollection (CLID 4143) is an identifiable collection of **TDCSpectCond**s (no CLID) which represent drift spectrum fit information. One **MDTTDCCollection** belongs to one mdt chamber. A converter is found in **MuonCTConditionsCnvSvc**.

MuonCTCondManager (CLID 4133) is not really an data class. It had to be assigned a CLID in order to be able to store it in the detector store (done by **MCTCondMgrTool**). **MuonCTCondManager** is used by clients in order to gather information about drift spectrum fits. It implements the IOV service mechanism.

MuonCTConditionsCnvSvc

Short Description

Converter that implements the MCT conditions ASCII database. Warning: the converters described below are template-instantiations of the general converter template class `MuonCTConditionsConv<T>`.

Athena Components

DeltaTransformConv (C) is a read only converter for **DeltaTransform**s (CLID 4144). The persistent representation of a **DeltaTransform** is given by its three translations and its three Euler angles. Access to files is gained by interaction with the conversion service.

MDTTDCSpectCollectionConv (C) is the read and write converter for **MDTTDCSpectCollection**s (CLID 4143). The persistent representation of one **MDTTDCSpectCollection** is given by a file containing several lines of numbers, each line representing fit information of one drift spectrum. Access to files is gained by interaction with the conversion service.

MuonCTConditionsCnvSvc (S) is the conversion service for converters in this class. It opens files and handles filestream references to its converters on demand.

Data Classes

None.

MuonCTDetDescr

Short Description

Implementation of the programmer view of the test stand setup. Information about the geometry is gathered by evaluating the GeoModel and some hardcoded numbers. It contains several GeoModel-specific classes, such as **GeoNodeAction**s, which explore the geometry tree.

Athena Components

MuonCTDetDescrTool (T) is utilized by **GeoModelService**. It instanciates the MuonCTCondManager and stores it into the detector store at the beginning of a job.

Data Classes

MDTDescriptor (CLID 4125) is the source for primary numbers of the MDT chamber detector elements, e. g. number of multilayers, length of tubes and speed of light along the wire. (At present, it is instantiated from **MDTDetectorElement** based on hardcoded numbers. This needs to be replaced by an external source in the future.)

MDTDetectorElement (CLID 4122) represents one MDT chamber. It provides methods that deliver information about single read out channels.

MuonCTDetDescrManager (CLID 4121) is not a data class but has a CLID, which allows it to be made available via the detector store. **MuonCTDetDescrManager** has pointers to all detector elements. Clients can request these pointers from **MuonCTDetDescrManager**.

ScintiDescriptor (CLID 4126) is the source for primary numbers of the scintillator detector elements, e. g. read out side or speed of light. (At present, it is instantiated from **MDTDetectorElement** based on hardcoded numbers. This needs to be replaced by an external source in the future.

ScintiDetectorElement (CLID 4123) represents one scintillator layer. It provides methods that deliver information about single read out channels.

StreamerDescriptor (CLID 4127) is the source for primary numbers of the streamer detector elements, e. g. width of pickup strips or number of gas volumes per bar. (At present, it is instantiated from **MDTDetectorElement** based on hardcoded numbers. This needs to be replaced by an external source in the future.

StreamerDetectorElement (CLID 4124) represents one streamer layer. It provides methods that deliver information about single read out channels.

MuonCTDigitization

Short Description

Algorithms situated in MuonCTDigitization are executed in a simulation job. Their puropose is to calculate digits from hit information provided by simulation engines as Geant4, as well as simulation of the trigger logic.

Athena Components

MDTDigitizer (A) collects MDT chamber hits (**MDTHitCollection**s (4100)) from **StoreGate**, calculates digits, **MDTDigitCollection**s (CLID 4103), and stores them in **StoreGate**.

ScintiDigitizer (A) collects a `std::map` of **Discriminator**s (CLID 4140) from **StoreGate** and produces scintillator digits, **ScintillatorDigitCollection**s (CLID 4104).

ScintiPreDigitizer (A) collects scintillator layer hits and stores a `std::map` of **Discriminator**s in **StoreGate**. The **Discriminator**s are later evaluated by **TriggerSim** and **ScintiDigitizer**.

StreamerDigitizer (A) collects streamer layer hits (**StreamerHitCollection**s (CLID 4102)), calculates digits, **StreamerDigitCollection**s (CLID 4105) and stores them in **StoreGate**.

TriggerSim (A) collects a `std::map` of **Discriminator**s (CLID 4140) from **StoreGate** and simulates the trigger logic of the setup. If the event is not triggered, a flag is set and the sequence of algorithms is stopped.

Data Classes

MuonCTEvent

Short Description

Data classes for transient hit and digit representation of the setup. These data classes contain no information of the setup itself. The only connection between transien hit/digit representation and the setup is via **Identifier**s: each hit/digit carries an **Identifier** that uniquely corresponds to one read out channel. Warning: hit collections are instantiated template classes of `DataVector<T>`.

Athena Components

None.

Data Classes

MDTDigitCollection (CLID 4103) is a collection of **MDTDigit**s (no CLID). One collection represents one chamber. **MDTDigit**s have methods to deliver information about TDC count an their identification.

MDTDigitContainer (CLID 4106) organizes **MDTDigitCollection**s (three in MCT, one for each chamber).

MDTHitCollection (CLID 4100) is a collection of **MDTHits** (no CLID). **MDTHits** have methods to deliver information about a global time, the drift radius, the distance of the hit from the readout and their identification.

ScintillatorDigitCollection (CLID 4104) is a collection of **ScintillatorDigit**s (no CLID). **ScintillatorDigit**s have methods to deliver information about ADC and TDC count and their identification.

ScintillatorDigitContainer (CLID 4107) organizes **ScintillatorDigitCollection**s (three in MCT, one for each layer).

ScintillatorHitCollection (ClID 4101) is a collection od **ScintillatorHit**s (no CLID). **ScintillatorHit**s have methods to deliver information about a global time, the distance from the left and right edge, an energy deposition and their identification.

StreamerDigitCollection (CLID 4105) is a collection of **StreamerDigits** (no CLID). **StreamerDigits**s have methods to deliver information about their identification only.

StreamerDigitContainer (CLID 4108) organizes **StreamerDigitCollection**s (two in MCT, one for each layer).

StreamerHitCollection (CLID 4102) is a collection of **StreamerHit**s (no CLID). **StreamerHit**s have methods to deliver information about a global time, the distance from the read out, a pulse height and their identification.

MuonCTG4Sim

Short Description

Implementation of all simulation related issues. Physics lists to be used in the simulation of Geant4 are implemented as well as the conversion algorithms that derives the Geant4 geometry from the GeoModel geometry. Geant4 sensitive detectors are implemented as Athena tools, using multiple inheritance from Geant4 sensitive detector and Athena tool base class.

Athena Components

MDTSD (T) implements the hit producing algorithm of drift tubes. **MDTSD** is attached to logical volumes representing drift tubes. Produced hits, **MDTHitCollection**s (CLID 4100), are stored in **StoreGate**.

MuonCTG4Sim (A) is the algorithm, in whose `initialize()` method the boundary conditions of the simulation are established. It uses the **MuonCTGeoToG4Svc** to build the Geant4 geometry from the GeoModel description.

MuonCTGeoToG4Svc (S) implements automatic conversion of GeoModel to Geant4 geometry conversion, including the instantiation and activation of sensitive detectors.

ScintillatorSD (T) implements the hit producing algorithm of scintillator bars. **ScintillatorSD** is attached to logical volumes representing scintillator bars. tubes. Produced hits, **MDTHitCollection**s (CLID 4101), are stored in **StoreGate**.

StreamerSD (T) implements the hit producing algorithm of streamer tueb cells. **StreamerSD** is attached to logical volumes representing streamer tube cells. tubes. Produced hits, **MDTHitCollection**s (CLID 4102), are stored in **StoreGate**.

Data Classes

None.

MuonCTGeoModel

Short Description

This sub package is full of classes which build the GeoModel tree. Only the top node has a CLID assigned, which is sufficient, since all daughter nodes are owned by their parent node. Building of the GeoModel tree involves several builder-classes which are not mentioned here.

Athena Components

MuonCosmicTeststandTool (T) is utilized by the **GeoModelSvc** in the beginning of a Job, following directives in joboptions. Its only purpose is to instantiate the GeoModel tree and store the root node in the detector store.

Data Classes

MuonCosmicTeststandNode (CLID 4120) represents the root node of the MCT GeoModel tree.

MuonCTGraphics

Short Description

Both **Algorithm**s residing in this sub package fetch track information from the event store and write out either a XML or ASCII file for each event that is interpreted later by a graphical front end.

Athena Components

AsciiTrackConverter (A) was intended to read track information from **StoreGate** and write it to ASCII files but was not properly updated.

AtlantisXMLTrackConverter (A) collects track information (**TrackCollection** (CLID 4115)) from **StoreGate** and writes to disk XML-files that are to be interpreted by the Atlantis event display (see [8]).

Data Classes

None.

MuonCTIdentifier

Short Description

Identifiers are objects that uniquely Identify sensitive elements (such as MDT chambers) or read-out channels (such as drift tubes), like numbers identify individual seats in a cinema. Identifier helpers are tools, that are capeable of performing several calculations, such as decomposing a seat number into its row and column number, or decomposing a tube identifier into its chamber, multilayer, layer and tube number. In addition to that, they utilize internal hash tables built from dictionary-files in the beginning of a job that allow very fast operations.

Athena Components

MuonCTIdHelperTool (T) instantiates all MCT identifier tools and puts them into the detector store. The mechanism uses the **GeoModelSvc** just as in the case of the **MuonCosmicTeststandTool**. (A little misuse is done here since the **GeoModelSvs** is not intended just to store objects in the detector store.)

Data Classes

MDTIdHelper (CLID 4130) performs calculation on MDT chamber subsystem identifiers (chamber, multilayer, layer and tube numbers).

ScintiIdHelper (CLID 4131) performs calculations on scintillator subsystem identifiers (layer and bar number).

StreamerIdHelper (CLID 4132) performs calculations on streamer subsystem identifiers (layer, bar and volume number).

MuonCTRawDataCnvSvc

Short Description

This sub package implements read converter for raw data originating of the test stand. With these converters, MCT is able to access the same data format as the MTOffline software. Several classes were simply adapted from the MTOffline package.

Athena Components

MuonCTRawDataCnvSvc (S) is the service for the storage technology that is used for the real teststand.

EventInfoRawDataConverter (C) mainly converts event and run number originating from raw test stand data into its transient representation, the **EventInfo** object.

[**MDTRawDataConverter** (C) converts MDT chamber digits.

ScintiRawDataConverter (C) converts scintillator digits.

StreamerRawDataConverter (C) converts streamer digits.

Data Classes

MuonCTRawDataEventSelector

Short Description

The event selector is responsible for single event selection of data sets. In case of MCT this is not crucial, since a single muon passage is always identified with one single event.

Athena Components

MuonCTRawDataEventSelector preloads proxies at the beginning of an event. Dereferencing of proxies triggers the converter mechanism.

Data Classes

None.

MuonCTRawEvent

Short Description

This package contains the slightly modified raw data classes implemented by the MTOffline project. They are utilized by the MCT raw data converter.

Athena Components

None.

Data Classes

None.

MuonCTReco

Short Description

MuonCTReco contains fitting related **Algorithm**s. In some cases, they are not concrete implementation of fit algorithms but utilize easy exchangable **Tool**s that perform the fit.

Athena Components

DetDescrTestAlg (A) is a test **Algorithm** that was intendet for validation of IOV-GeoModel related mechanisms.

ExNTupleMaker (A) is not used anymore.

GlobalTrackFitter (A) is not used anymore.

MDTPatternFinder (A) fetches drift circles (**MDTDriftCircleCollection** (CLID 4113)) from **StoreGate** and discovers if subsets of drift circles build a pattern according to a set of parameters defined in **MDTPatternFinder**'s joboptions. These subsets are placed in **StoreGate** as **MDTPatternCollection**s (CLID 4113).

MDTTimeToRadTransform (A) retrieves run time corrected drift times from **StoreGate** (**MDTDriftTimeCollection** (CLID 4117)) and subtracts the the t_0 value. Then, the drift time is converted into a drift radius. At this stage, geometry information is consulted and stored into the resulting **MDTDriftCircleCollection**s (CLID 4111).

MDTTrackFinder (A) is not used anymore.

MDTTrackFitter (A) evaluates patterns (**MDTPatternCollection**s (CLID 4113)) by performing fits. The fit algorithms are not implemented in **MDTTrackFitter**. Instead, **MDTTrackFitter** utilizes a tool with the concrete algorithm.

ScintiTrackFinder (A) is not used anymore.

TDCDelayAdjust (A) reads **MDTDigitCollection**s (CLID 4103) and **ScintillatorDigitCollection** (CLID 4104) from **StoreGate**, performs rudimentary fits and calculates run time corrections of the measured drift times. The result is stored in **StoreGate** as **MDTDriftTimeCollecion** (CLID 4117).

Data Classes

None.

MuonCTRecoEvent

Short Description

MuonCTRecoEvent defines the transient representation of reconstructed event data.

Athena Components

None.

Data Classes

MDTDriftCircleCollection (CLID 4111) is a collection of drift circles (**MDTDriftCircle** (no CLID)). Drift circles already contain geometry information.

MDTDriftCircleContainer (CLID 4112) organize **MDTDriftCircleCollection**s.

MDTDriftTimeCollection (CLID 4117) is a collection of drift times (**MDTDriftTime** (no CLID)). **MDTDriftTime**s are run time corrected drift times, that have not yet the t_0 subtrackted. They contain no geometry information.

MDTDriftTimeContainer (CLID 4118) organizes **MDTDriftTimeCollection**s.

MDTPatternCollection (CLID 4113) is a collection of patterns of the MDT chambers (**MDTPattern** (no CLID)). Patterns are sub sets of hit tubes that might form a muon trajectory.

MDTPatternContainer (CLID 4114) organizes **MDTPatternCollection**s.

TrackCollection (CLID 4115) is a collection of tracks (**Track** (no CLID)), either local chamber or global tracks.

TrackFitTimes (CLID 4119) is an object that maps track pointers to the time it took to perform the fit. It is evaluated by **MuonCTNTuple**.

TrackInfo (CLID 4134) extends the information held by **TrackCollection**. It is supposed to connect global tracks with its corresponding local tracks.

TrackPointCollection (CLID 4109) is a collection of **TrackPoint**s (no CLID).

TrackPointContainer (CLID 4110) organizes **TrackCollection**s.

MuonCTRecoUtils

Short Description

This packages hold several components, all **Tool**s, that are used to perform small fitting related tasks. Many of them are utilized from **Algorithm**s of the MuonCTReco sub package. There are several **Tool**s that solve identical problems using different approaches. The end user is able to switch between these alternatives easily by modifying the joboptions file. To clarify the use-relationships of some **Tool**s in this package: **MuonCTReco** uses **StraightLineDCFitter** or **StraightLineTPFitter**, respectively (the choice is made in the joboptions). Everything containing "DCLineFitter" in its name is utilized by **StraightLineDCFitter**, while everything containing "LineFitter" (without "DC") is used by **StraightLineTPFitter**.

Athena Components

IronScatterAngle (T) is used to calculate the offset between a muon trajectory and the measured streamer tube layer impact point.Therefore, it fetches **TrackCollection** (CLID 4115) and **StreamerDigitCollection**s (CLID 4105) from **StoreGate**.

LazyDCLineFitter (T) is used by **StraightLineDCFitter**. It does nothing but returning the suggestion for slope an intercept stored in a **MDTPattern** (no CLID) as result for the reconstructed track parameters. It is used for debugging.

LinearDCLineFitter (T) implements a linear, analytical fit method. It is used by **StraightLineDCFitter**.

MinuitDCLineFitter (T) implements a fit algorithm, that minimizes residuas utilizing MINUIT and is used by **StraightLineDCFitter**.

NumRecLineFitter (T) implements a linear regression based on an algorithm found in [7] and is used bt **StraightLineTPFitter**.

RootLineFitter (T) is used by **StraightLineTPFitter**. It performs a numerical linear regression based on ROOT.

ScintiTrackFitter (T) is a **Tool** that can be used to perform a track fit based on the scintillator information. It fetches **ScintillatorDigitCollection**s (CLID 4104) from **StoreGate**. The resulting fit parameters describe the muon trajectory in a plane perpendicular to the plane fitted based on drift tube digits.

SemiAnalyticDCLineFitter (T) implements a semi-analytical method to minimize residuas. It is not completely tested. It is used by **StraightLineDCFitter**.

StraightLineDCFitter (T) is used by **MuonCTReco**. It utilizes a helper **Tool** (e. g. **SemiAnalyticDCLineFitter** or **NumRecDCLineFitter**) that implements a fit based on residua minimizing. The residuas are defined as the difference between the measured radius and the radius prediction of the fitted track for single tubes.

StraightLineTPFitter (T) is used by **MuonCTReco**. Based on a **MDTPattern**, it estimates points which lay on the muon trajectory. These points are subject to a linear regression, which is performed utilizing a helper **Tool** (e. g. **RootLineFitter** or **NumRecLineFitter**).

TangentsOnCirclesSvc (T) is not used anymore.

Data Classes

None.

MuonCTRelease

Short Description

MuonCTRelease contains no code at all. In its current version, Athena uses CMT [17] as its code management tool. Sub packaging is subdued to this system, every package contains a directory named *cmt* that defines its use-relationships. MuonCTRelease's cmt directory defines a dependence on all MCT packages. Usually, MCT-Athena jobs are executed after processing all directives in this subdirectory.

Athena Components

None.

Data Classes

None.

MuonCTUtils

Short Description

MuonCTUtils contains code that does not really fit in any of the other categories. At the moment, it contains neither Athena components nor data classes. In MuonCTUtils, helper classes are found, that help calculating a drift tube's wire sag or define a simple drift time relation.

Athena Components

None.

Data Classes

None.

Appendix C

Detailed Example: Joboptions

In this chapter, a detailed example of a joboptions file is given. This can be starting point for different Athena applications based on the MCT package. The language of joboptions files is closely related to the C++ language. The job below is capeable of performing the static chamber alignment as described in chapter 4.2.1.

```
//////////////////////////////////////////////////////////////////////////////
//
// MCTFindChamberPostionOptions.txt
// --------------------------------
//
//////////////////////////////////////////////////////////////////////////////

Mandatory for every job, except simulations...
#include "$ATHENACOMMONROOT/share/Atlas.UnixStandardJob.txt"

comments...
//--------------------------------------------------------------
// configure application
//--------------------------------------------------------------

Here, all libraries are defined that are used for the following job - their name indicates their purpose.
E. g. "IOVSvc" denotes that file "libIOVSvc.so" contains class implementations that are needed throughout this job.
ApplicationMgr.DLLs   += { "GaudiAlg",
                           "GeoModelSvc",
                           "GeoModelGraphics",
                           "RootSvcModules",
                           "RootHistCnv",
                           "MuonCTDetDescr",
                           "MuonCTGeoModel",
                           "MuonCTIdentifier",
                           "MuonCTConditionsCnvSvc",
                           "MuonCTAlgs",
                           "MuonCTCalibAlgs",
                           "IOVSvc",
                           "MuonCTReco",
                           "MuonCTRecoUtils",
                           "MuonCTGraphics"};
```

External services - they need to be implemented in the libraries above. AtRndmGenSvc generates random numbers.
```
ApplicationMgr.ExtSvc += { "AtRndmGenSvc",
```
IOVSvc is external, too.
```
                           "IOVSvc",
```
IOVASCIIDbSvc is the concrete choice of an IOV database.
```
                           "IOVASCIIDbSvc",
```
GeoModelSvc is responsible for instantiating the geometry tree in the beginning of a job.
```
                           "GeoModelSvc",
```
This is the concrete converter choice for drift spetrum information - it implements the ASCII conditions database.
```
                           "MuonCTConditionsCnvSvc"};
```

This tells the ApplicationMgr to instantiate a Sequencer with the name "TopSequence" as the top algorithm. Instead, arbitrary Algorithms could be instantiated here. Sequencers are a way to organize Algorithms.
```
ApplicationMgr.TopAlg   = { "Sequencer/TopSequence" };
```

The following lines define how the instance named "TopSequence" of Sequencer shall be configured. All members are Athena Algorithms. Some of them are described in the text.
```
TopSequence.Members     = { "TDCDelayAdjust/TDCDelayAdjust",
                            "MDTTimeToRadTransform/MDTTimeToRadTransform",
                            "MDTPatternFinder/MDTPatternFinder",
                            "MDTTrackFitter/MDTTrackFitter",
                            "ChamberPosition/ChamberPosition"};
```
GeoModelDisplay is the Algorithm, that generated the test stand picture 2.3. It is outcommented here.
```
                            //"GeoModelDisplay/GeoModelDisplay" };//,
                            // "AtlantisXMLTrackConverter/AtlantisXMLTrackConverter"};
```

Define the level of output...
```
MessageSvc.OutputLevel        =    3; // 2=DEBUG, 3=INFO, 4=WARNING, 5=ERROR, 6=FATAL
```
Number of events to be processed...
```
ApplicationMgr.EvtMax         =    100000000;

// Pattern Finder
// --------------
```
Defines properties of the instance of MDTPatternFinder...
```
MDTPatternFinder.MinPerChamber   = 5;
MDTPatternFinder.RoadWidth       = 5.;
MDTPatternFinder.GlobalDeltaInter= 40.;
MDTPatternFinder.GlobalDeltaSlope= 0.2;
MDTPatternFinder.MaxResidualSum  = 10.0;
MDTPatternFinder.MaxResidualAvg  =  2.0;
```
GlobalPatternIgnoresMiddleChambers indicates whether a pattern has to be built for the middle chamber or not. If one is interested in chamber positions, local track fits for all chambers are needed. If one wants to predict wire positions, only the reference chambers are used.
```
MDTPatternFinder.GlobalPatternIgnoresMiddleChambers = false;

// Track Fitter
// ------------
```
Configuration of the TrackFitter. First the strategy tool is defined...
```
MDTTrackFitter.TrackFitter = "StraightLineDCFitter";
```
Next, the concrete algorithm that is used by the strategy tool is defined...
```
MDTTrackFitter.StraightLineDCFitter.LineDCFitter = "MinuitDCLineFitter";
MDTTrackFitter.MinGlobalProb = 0.1;

// GeoModelSvc
// -----------
```
The following lines define helpers of GeoModel, whose main purpose is to store objects of different type in the detector store. To a certain extend, abuse is done here, since the helpers are used to store several things that are not connected to GeoModel. MuonCTIdHelperTool stores the id helpers in the detector store.

```
GeoModelSvc.Detectors = { "MuonCTIdHelperTool",
```
MuonCosmicTeststandTool instanciates the MCT GeoModel and stores it.
```
                          "MuonCosmicTeststandTool",
```
MuonCTDetDescrTool instantiates and stores the MuonCTDetDescrManager.
```
                          "MuonCTDetDescrTool",
```
MCTCondMgrTool instantiates and stores the MuonCTCondManager
```
                          "MCTCondMgrTool"};

// Conditions
// ----------
```
Activate MuonCTConditionsCnvSvc as a conversion service...
```
EventPersistencySvc.CnvServices += { "MuonCTConditionsCnvSvc" };
```
Instruct IOVASCIIDbSvc to be responsible for filling proxies...
```
ProxyProviderSvc.ProviderNames += { "IOVASCIIDbSvc" };
```
Define the root directory of the IOV ASCII database. This path points to the local directory from where the job is started, what indicates, that a local, private version of the IOV database is used here.
```
IOVASCIIDbSvc.DBname = "IOVDB";
```
Define, which CLIDs the IOVASCIIDbSvc is responsible for...
```
IOVASCIIDbSvc.ClassID = { 4143, 4144 };
```
This tells the IOVASCIIDbSvc, what to do, if no entry is found in the IOV database...
```
IOVASCIIDbSvc.UseDefaultRange = false;
```

Root directory of the conditions database...
```
MuonCTConditionsCnvSvc.CondDBRoot = "CondDB";
```
Feature of the conditions data converter - do not overwrite data that has been produced once...
```
MuonCTConditionsCnvSvc.overwriteConditionsData = false;

// RawData / MonteCarlo
// --------------------
```
If not outcommended, this would install Monte Carlo as input source...
```
//#include "MCTAthenaRootReadOptions.txt"
```
But we want to read real data here...
```
#include "MCTRawDataReadOptions.txt"

// AtlantisXMLTrackConverter
// -------------------------
```
Destination of Atlantis-XML files...
```
AtlantisXMLTrackConverter.Directory      =   "event/atlantis";

////////////////////////////////////////////////////////////////////////////////
```

Appendix D

Description of the MCT N-Tuple

The following lines describe the entries of the n-tuple produced by the **Algorithm MuonCT-NTuple**. The *name*, via which the entries are made available, their *type* and their *description* is given. A type followed by a square bracket [] indicates that the type is an array with upper bound index given by the value in brackets.

Name	**Type**	**Description**
event parameters		
"run"	long	run number
"event"	long	event number
global fit results		
"chi2"	float	χ^2-sum
"slope"	float	slope
"sigSlope"	float	error on slope
"intercept"	float	intercept
"sigIntercept"	float	error on intercept
"covariance"	float	covariance of slope and intercept
"scatterOffset"	float	offset of streamer prediction w. r. t. the fitted track in y-direction
"xzSlope"	float	slope of hodoscope fit
"xzIntercept"	float	intercept of hodoscope fit
"nTube"	long	number of tubes consulted
upper chamber local fit results		
"chi2UC"	float	χ^2-sum
"slopeUC"	float	slope
"sigSlopeUC"	float	error on slope
"interceptUC"	float	intercept
"sigInterceptUC"	float	error on intercept
"covarianceUC"	float	covariance of slope and intercept
"chamberUC"	long	identification number of upper chamber *redundant*
"xzSlopeUC"	float	slope of hodoscope fit *redundant*
"xzInterceptUC"	float	intercept of hodoscope fit *redundant*
"nTubeUC"	long	number of tubes consulted
test chamber local fit results		
"chi2TC"	float	χ^2-sum

"slopeTC"	float	slope
"sigSlopeTC"	float	error on slope
"interceptTC"	float	intercept
"sigInterceptTC"	float	error on intercept
"covarianceTC"	float	covariance of slope and intercept
"chamberTC"	long	identification number of test chamber *redundant*
"xzSlopeTC"	float	slope of hodoscope fit *redundant*
"xzInterceptTC"	float	intercept of hodoscope fit *redundant*
"nTubeTC"	long	number of tubes consulted *redundant*
lower chamber local fit results		
"chi2LC"	float	χ^2-sum
"slopeLC"	float	slope
"sigSlopeLC"	float	error on slope
"interceptLC"	float	intercept
"sigInterceptLC"	float	error on intercept
"covarianceLC"	float	covariance of slope and intercept
"chamberLC"	long	identification number of lower chamber *redundant*
"xzSlopeLC"	float	slope of hodoscope fit *redundant*
"xzInterceptLC"	float	intercept of hodoscope fit *redundant*
"nTubeLC"	long	number of tubes consulted *redundant*
upper chamber hits		
"nHitUC"	long	number of drift circles that contribute *redundant*
"yPosUC"	float["nHitUC"]	array of y-positions of drift circle centers
"zPosUC"	float["nHitUC"]	array of z-positions of drift circle centers
"rUC"	float["nHitUC"]	array of radii of drift circles
"tUC"	float["nHitUC"]	array of drift times of corresponding to drift circles
"sigRUC"	float["nHitUC"]	array of errors on radii of drift circles
"hit_chamberUC"	long["nHitUC"]	array chamber number of drift circles *redundant*
"hit_multilayerUC"	long["nHitUC"]	array of multilayer number of drift circles
"hit_layerUC"	long["nHitUC"]	array of layer number of drift circles
"hit_tubeUC"	long["nHitUC"]	array of tube number of drift circles
test chamber hits *in case the test chamber was excluded from the track fitting, the hit section contains information about all hits, and not about hits that belong to a local track*		
"nHitTC"	long	number of drift circles that contribute *redundant*
"yPosTC"	float["nHitTC"]	array of y-positions of drift circle centers
"zPosTC"	float["nHitTC"]	array of z-positions of drift circle centers
"rTC"	float["nHitTC"]	array of radii of drift circles
"tTC"	float["nHitTC"]	array of drift times of corresponding to drift circles
"sigRTC"	float["nHitTC"]	array of errors on radii of drift circles
"hit_chamberTC"	long["nHitTC"]	array chamber number of drift circles *redundant*
"hit_multilayerTC"	long["nHitTC"]	array of multilayer number of drift circles
"hit_layerTC"	long["nHitTC"]	array of layer number of drift circles
"hit_tubeTC"	long["nHitTC"]	array of tube number of drift circles
lower chamber hits		
"nHitLC"	long	number of drift circles that contribute *redundant*

"yPosLC"	float["nHitLC"]	array of y-positions of drift circle centers
"zPosLC"	float["nHitLC"]	array of z-positions of drift circle centers
"rLC"	float["nHitLC"]	array of radii of drift circles
"tLC"	float["nHitLC"]	array of drift times of corresponding to drift circles
"sigRLC"	float["nHitLC"]	array of errors on radii of drift circles
"hit_chamberLC"	long["nHitLC"]	array chamber number of drift circles *redundant*
"hit_multilayerLC"	long["nHitLC"]	array of multilayer number of drift circles
"hit_layerLC"	long["nHitLC"]	array of layer number of drift circles
"hit_tubeLC"	long["nHitLC"]	array of tube number of drift circles

Bibliography

[1] H. van der Graaf, H. Groenstege, F. Linde, P. Rewiersma, *RasNiK, an Alignment System for the ATLAS MDT Barrel Muon Chambers - Technikal System Description*, NIKHEF/ET38110, 2000

[2] The Gaudi Collaboration, `http://proj-gaudi.web.cern.ch/`

[3] E. Gamma, R. Helm, R. Johnson, *Design Patterns. Elements of Reusable Object- Oriented Software*, Addison-Wesley, 1997

[4] J. Boudreau, D. Quarrie, M. Shapiro, *Geometry Kernel Classes*, `http://atlas.web.cern.ch/Atlas/GROUPS/DATABASE/detector_description/`

[5] C. Leggett, *IOVSvc Documentation*, `http://annwm.lbl.gov/~leggett/Atlas/IOVSvc/main.shtml`

[6] J. Beck, M. Dobbs, *HepMC a C++ Event Record for Monte Carlo Generators*, `http://cern.ch/mdobbs/HepMC/`

[7] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*, Camebridge University Press

[8] H. Drewermann, G. Taylor, *Atlantis Event Display*, `http://cern.ch/atlantis/`

[9] A. Brandt, G. Duckeck, P. Schieferdecker, *(access to MuonCosmicTeststand sorce code via the US ATLAS LXR-Code Browser)*, `http://atlassw1.phy.bnl.gov/lxr/source/atlas/MuonSpectrometer/MuonCosmicTeststand/`

[10] The Geant4 Collaboration, `http://cern.ch/geant4/`

[11] D. R. Musser, A. Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1997

[12] O. Kortner, *Schauerproduktion durch hochenergetische Myonen und Aufbau eines Höhenstrahlprüfstands für hochauflösende ATLAS Myonkammern*, Doctoral Thesis, LMU Munich, 2002

[13] F. James, *MINUIT, Function Minimization and Error Analysis, Reference Manual*, CERN Program Library Long Writeup D506

[14] R. Brun, F. Rademakers, *ROOT System Homepage*, `http://root.cern.ch/`

[15] *CLHEP - A Class Library for High Energy Physics*, `http://wwwinfo.cern.ch/asd/lhc++/clhep/`

[16] O. Biebel et al., *A Cosmic Ray Measurement Facility for ATLAS Muon Chambers*, arXiv:physics/0307147, LMU-ETP-2003-01.

[17] C. Arnault, *CMT*, `http://www.cmtsite.org/`

[18] *ATLAS Detector And Physics Performance*, CERN/LHCC 99-14

[19] *ATLAS posters*, `http://atlas.web.cern.ch/Atlas/documentation/poster/HTML/`

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt und nur im Literaturverzeichnis angegebene Quellen verwendet habe.

München, 4. November 2003

(Alexander Brandt)