Die approbierte Originalversion dieser Diplom-/ Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.

http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.

http://www.ub.tuwien.ac.at/eng

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DIPLOMARBEIT

A coherent framework for full, fast and parametric detector simulation for the FCC project

ausgeführt am

Atominstitut
der Technischen Universität Wien

unter der Anleitung von

Privatdoz. Dipl.-Ing. Dr.techn. Michael Benedikt
(Atominstitut, TU Wien)
Dr. Andreas Salzburger (CERN)

durch

Julia Stefanie Hrdinka
2721 Bad Fischau-Brunn, An der Wasserleitung 26

September 14, 2015

Abstract

The outstanding success of the physics program of the Large Hadron Collider (LHC), including the discovery of the Higgs boson, shifted the focus of part of the high energy physics community onto the planning phase for future collider projects. The FCC (Future Circular Collider) is a five year international design study to explore post-LHC possibilities. Hadron and electron-positron based collider technologies are considered as potential LHC successor project branches.
Common to both branches is the need of a coherent software framework, in order to carry out simulation studies to establish the potential physics reach or to test different technology approaches.
Detector simulation is a particularly necessary tool needed for both, design studies for different detector concepts and the establishment of relevant performance parameters. In addition, it allows to generate data as input for the development of reconstruction algorithms needed to cope with the expected future environments.
A coherent framework will be presented, that combines full, fast and parametric detector simulation embedded in the Gaudi framework and based on the FCC Event Data Model. Detector description is based on DD4hep and the different simulation approaches are centrally steered through the Geant4 simulation.
A geometry for reconstruction was integrated into the framework. This geometry will also be used for fast simulation. The DD4hep geometry is automatically translated into the different geometries needed by the different simulations. A prototype example of a simple tracking detector is demonstrated for the different simulation approaches and the detector geometries together with the materials are compared. Furthermore, a potential workflow to use full simulation based on Geant4 and fast simulation techniques alongside is presented and the results are compared.

Acknowledgments

My diploma thesis was only possible with the help of many people and I want to thank at this point to all my colleagues and my family.

Special thanks to:

The FCC Software group for the support and the pleasant working environment.
The ATLAS group, for the excellent collaboration and working climate.
I want to thank Michael Benedikt, my university supervisor, for guidance and for making this whole experience possible for me.
Thank you Andreas Salzburger for being a great supervisor, giving me interesting tasks and promoting me into the right direction. I learned a lot this year and I am looking forward to working with you in my doctors.
Another person who made this project possible is Werner Riegler. He supported me in the context of the FCC detector group.
Benedikt Hegner was a big help for many tasks and always gave me good advice.
Thanks to Anna Zaborowska, for being a good co-worker and helping me in various situations.

Thank you Michael and Andi for the great Tea-Times.

Mama, thank you for your unconditional love and for raising me to a strong, feminist woman.
Thank you Papa for the moon-landing-bedtime stories and our endless talks about the universe and "the nothing".
Patrick, thank you for trying to hold me back from "vakopfa".

Contents

Chapter 1

Introduction

The Standard Model of particle physics (SM) is one of the most established theories in science. Over more than forty years it has been very successfully applied to describe the interactions and types of fundamental particles and has been tested to extremely high precision. No experiment has managed to conclusively disprove the SM to date. The recently discovered Higgs boson, announced by the ATLAS [2] and CMS [3] collaborations at the Large Hadron Collider (LHC) [1] located at CERN, Switzerland, has filled the puzzle piece of electro-weak symmetry breaking and confirmed the existence of the Higgs-mechanism.
However, phenomena as for instance the baryon asymmetry, quark confinement or the nature of dark matter raise questions that can not be addressed within the Standard Model. Moreover, the SM can only describe three out of the four fundamental forces and since masses remain free parameters within the SM, it includes a huge set of model parameters [11].
Since so-called *Beyond Standard Model* (BSM) phenomena have not been discovered in Run1 at the LHC, future experiments focus on higher energy scales to search at a new energy frontier for rare processes and new particles.
As high-energy experiments require decades of planning and construction, attention is drawn to experimental facilities for the post-LHC era in order to adress the open questions that may not be answered with the LHC physics program. This led to the formulation of a common european strategy for the future of High Energy Physics beyond 2025 when the LHC is planned to be decommissioned.
Based on the huge success of the LHC and its predecessor, the Large Electron-Positron-Collider (LEP 1989-2000) [5], the focus is drawn onto a new circular collider factory.

1.1 FCC – Future Circular Collider

The FCC (Future Circular Collider) is an international five-year design study to explore post-LHC particle accelerator options. Until the end of 2018 a conceptual design report will be delivered as an outcome of this study. This report will consider all technical aspects, concerning the machine and infrastructure as well as the physics opportunities, the discovery potentials, the experimental concepts and the detector designs. First cost estimates will be included in this design report [7].
Extrapolations of the current available technology, together with the need to keep the

synchrotron radiation under control, require an effective increase of the circumference of the accelerator with respect to the current LHC in order to reach the desired energies and luminosities. Building the tunnel with a circumference of 80-100 km in the Geneva area, with the LHC as an injector is being considered as an option, aiming to make best use of already existing facilities (see Fig 1.1). The first step would be the realization of

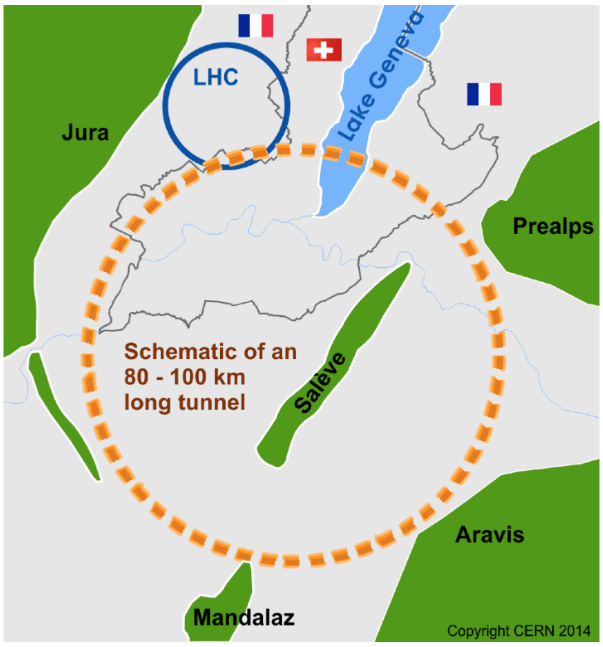


Figure 1.1: Possible realization of the FCC in the Geneva area, using the LHC as an injector.

an electron-positron collider with a center of mass energy of 90-400 GeV. The ultimate goal is a proton-proton (FCC-hh) collider, which will be installed in the same tunnel, with a center of mass energy of 100 TeV and a peak luminosity of $5 * 10^{34} cm^{-2} s^{-1}$. The option of a positron-electron is also being considered (FCC-he), which could operate at the same time as FCC-ee/FCC-hh [6].
The main physics goals for the FCC-hh are the exploration of electro-weak symmetry breaking by studying the Higgs boson in high precision (higher luminosity, much increased rates, access to very rare processes) and the exploration of BSM phenomena. An emphasis is also put on better understanding W/Z physics and Quantum chromodynamics (QCD). However, also the exploration and high precision measurement of known Standard Model particles, such as the top quark, the bottom quark or the tau lepton is intended [8].

1.2 Detector Simulation

Monte Carlo (MC) simulation, in general, is needed to allow studies that demonstrate the physics reach of future facilities. It is used to explore e.g. the possible interactions, the cross sections or possible production rates. Moreover, detector simulation is essential for the exploration of possible detector concepts in the upcoming environment. It simulates the interactions of the particle with the detector material and the response of the readout electronics.

The first step of the MC chain (see Fig 1.2) is the generation of the process induced

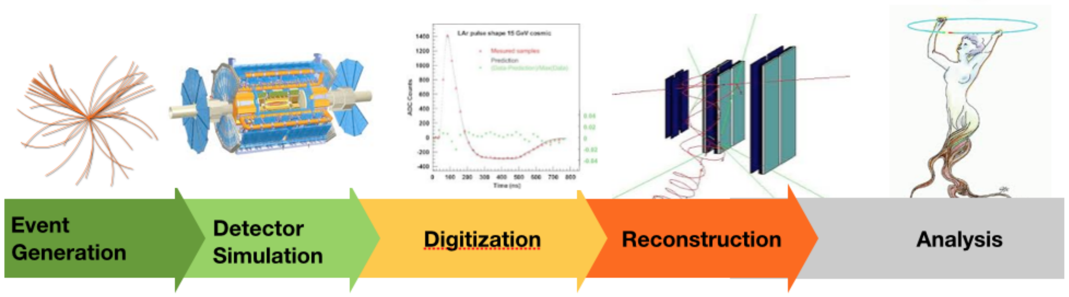


Figure 1.2: Illustration of the Monte Carlo simulation chain in high energy physics: physics event generators are used to simulate the primary process from the beam-beam collision. This input is successively processed by detector simulation, digitization and reconstruction modules, before finally being used for analyses.

by the initial beam-beam interaction. This is usually carried out by external physics event generators such as PYTHIA [9], HERWIG [10]. The particles are then handed over to the detector simulation, which emulates the interaction of the particle with the detector. When a particle passes through the detector, it interacts with the material. All kinds of interactions can occur, depending on the particle type and energy: Charged particles can undergo multiple scattering, can cause ionization and excitation or can lose energy due to Bremsstrahlung or the Cherenkov effect. Basically three effects dominate the interactions of photons with matter: the photoelectric effect, Compton scattering and pair production. Hadrons underlie the strong interaction and can undergo nuclear interactions and neutrinos only interact weakly [11].

One usually distinguishes between passive and sensitive material in the detector. Passive elements are used to describe support structures, cooling facilities, readout electronics or simply cables. Sensitive material, on the other hand, describes the detection devices, such as e.g. the silicon sensors in a pixel detector, or the ionization gas in gas-filled detectors. In detector simulation, energy deposits of particles in sensitive detectors are recorded, as they are used to emulate the readout signal of the detector. This detector response is then described by the digitization and aims to transform the raw simulation input into signals as being read out by the detector readout system. The last step is the reconstruction, which describes the set of algorithms that are responsible to build

meaningful physics objects, such as particles or jets from the detector output. Reconstruction algorithms often perform local or global pattern recognition to find associate hit patterns and to connect them to build higher level objects.
There are different types of detector simulation strategies concerning accuracy and time consumption. To study the detector design or to undertake technology studies, a detailed simulation is needed. This so-called *full simulation* uses the full and very detailed detector geometry. The particles are transported through the material taking the magnetic field into account. Interactions of the particle with the detector material are simulated precisely. This simulation technique is the most accurate kind of simulation, but on the other hand also the most time consuming one. The full simulation needs to be followed by digitization and reconstruction. Full simulation describes the data from the LHC experiments with an outstanding accuracy. Figure 1.3 shows an example of the ATLAS experiment for the good agreement between data and simulation.

However, the use of full simulation was already limited by the available computing resources during Run-1 of the LHC data taking campaign and fast simulation techniques had to be used in order to achieve the necessary statistics needed for certain analyses. Fast simulation techniques try to emulate the full simulation output by applying different approaches. They usually implement simplifications or parameterisations of the full simulation effect. The simplification can be done for the geometry, the physics processes or can be implemented by skipping certain steps or parameterising the whole process.
In the *parametric simulation* approach the different detector regions are described by simple envelopes. Different fast simulation models are attached to this detector regions, using the parameters of the full simulation as an input. The particle is transported from the entry to the exit point of the envelope, while its momentum or energy and track resolutions are smeared. For each detector model, new resolutions need to be obtained from the full simulation, however this can be done with a reasonable smaller number of events.
Other *fast simulation* approaches (used in ATLAS and CMS) use the reconstruction geometry. This is a simplified detector geometry, consisting of the sensitive material and a simplified material description. Hence, a material approximation is needed. In this way the simulation is much faster and less precise but still reflects the detector effects accurately.
Fast simulation can produce higher level output data, which does not need to be followed by digitization and can be directly used for reconstruction. The fast simulation can create the needed track parameters already during the simulation.
An example of a fast simulation approach from the ATLAS experiment is ATLFAST-II. It gains one order of magnitude in speed compared to the full Geant4 simulation by using a fast simulation approach in the calorimeter (FastCaloSim) [13]. In Fig 1.4 one can see a validation plot of ATLFAST-II in comparison with the full Geant4 simulation and data [14].

1.2.1 Simulation Framework Requirements

To facilitate the analysis and the comparison of the data, a coherent simulation framework that allows the combination of the different simulation types is essential. This simulation framework should only have one input for the detector geometries, needed

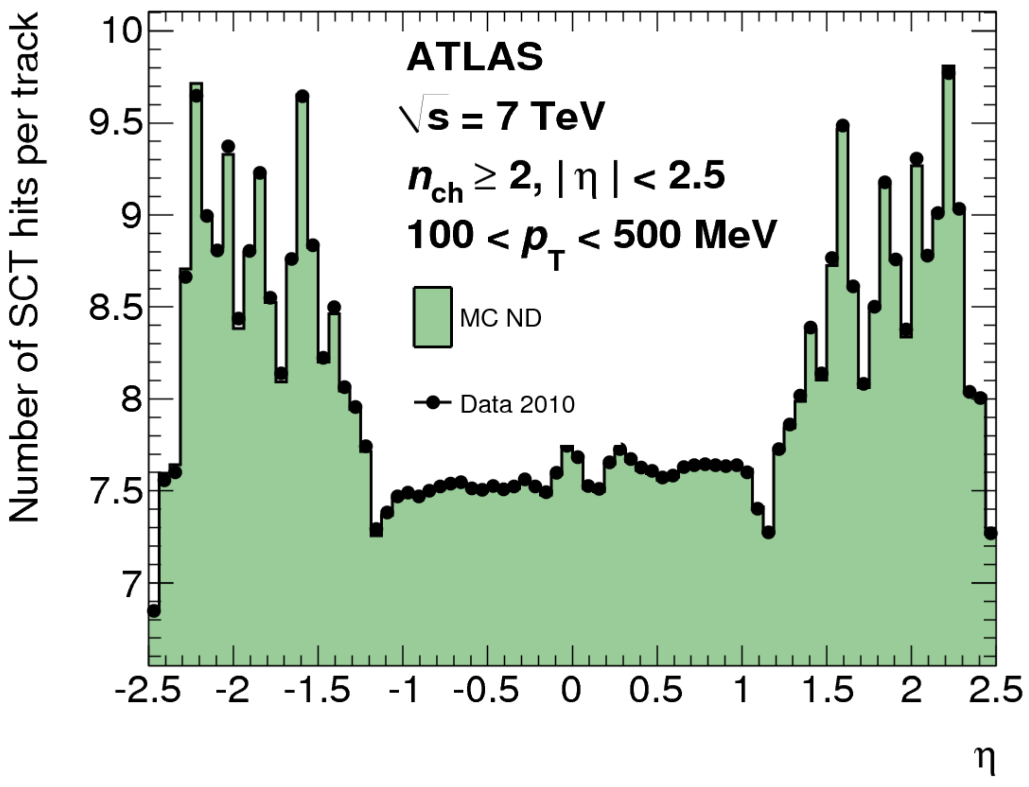


Figure 1.3: Comparison between data and simulation at sqrt(s) = 7 TeV for tracks with transverse momentum p_T between 100 and 500 MeV for the ATLAS experiment: the average number of silicon hits on reconstructed tracks as a function of pseudo rapidity η in the Semiconductor Tracker (SCT) is shown. The p_T distribution of the tracks in non-diffractive MC is re-weighted to match the data and the number of events is scaled to the data [12].

by the different simulation types and an identical analysis event data model. We aim to create a flexible framework, following the approach of the ATLAS ISF (Integrated Simulation Framework) [16]. This method allows to mix different kinds of simulations for different parts of the detector in one event by special particle routing algorithms. Thus, accuracy and speed can be balanced in different combinations, using the detailed full simulation where needed and a fast simulation for regions of less interest for physics studies.

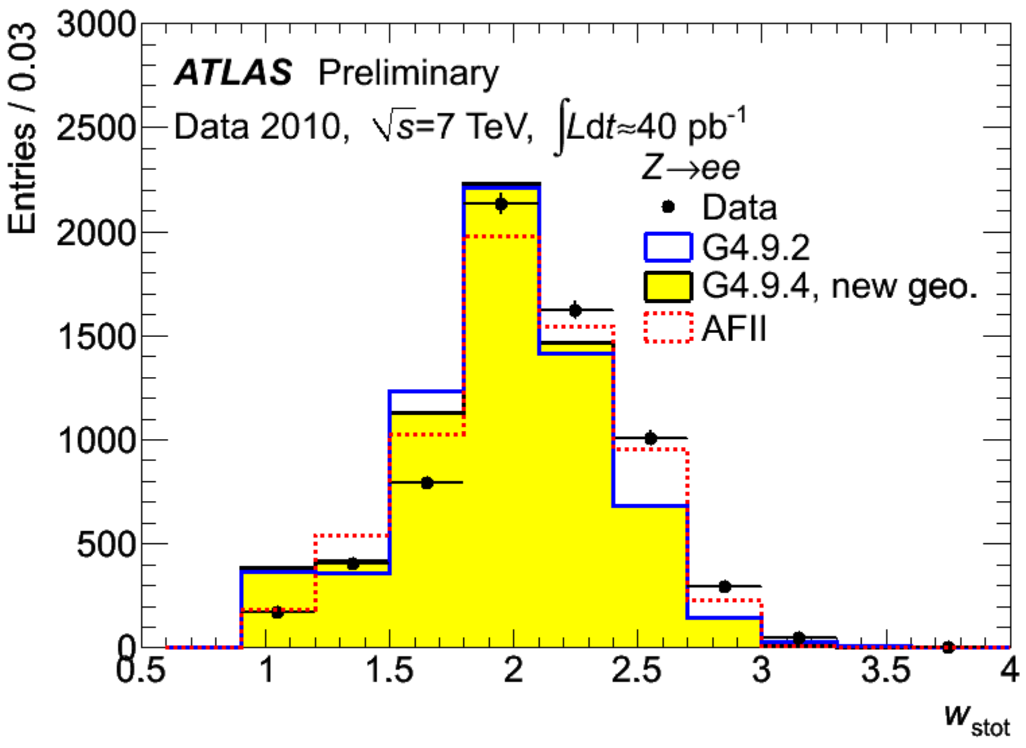


Figure 1.4: Distribution of the particle shower width w_{stot} in the high granularity strip layer 1 of the electromagnetic calorimeter of ATLAS. Data taken from 2010 with a center-of-mass energy of 7 TeV is compared with Geant4 full simulation (yellow) and ATLFASTII (dashed red), for $Z \rightarrow ee$ events [15].

Chapter 2

FCC-Software

The FCC-Software group was formed to establish a common software suite usable for the three rather diverse FCC projects (FCC-ee, FCC-hh and FCC-eh). Many components for the event processing (including simulation and reconstruction) already exist as part of the software of either the LHC experiments or of other future collider studies, e.g. a future linear collider. In order to optimise the workload, a thorough review of existing software had initially been done to identify usable components for the FCC project.
To contribute to the development of the FCC software framework was a major part of this thesis, especially the integration of the components in a coherent way 2.5 to enable common steering of full, fast and parametric detector simulation. A single source of detector information and a common output guarantees consistency and comparability. The focus of this work is on tracking detectors, however, the overall design chosen is compatible with all detector technologies. In this Chapter an overall view of the framework and a short introduction to the main components is given.
To execute the event loop, an event processing framework is needed, which was chosen to be the Gaudi framework [17] (see Section 2.1). The geometry description of the detector, which is done in DD4hep [18] (Section 2.2) is a basic input for all detector descriptions used in simulation, digitization and reconstruction. Geant4 [22] is used as the simulation environment in which full and fast simulation is embedded (Section 2.3). The Event Data Model (EDM) is described in Section 2.4, which is essential for storing and accessing the data and finally for the event analysis. In Fig 2.1 the overall picture of the framework with a simplified work flow is displayed.

2.1 A Gaudi Based Framework

The FCC community agreed on using Gaudi as the event processing framework. Gaudi is an experiment independent software architecture and framework for building high energy physics data applications for simulation, reconstruction and analysis. It has a long tradition in serving as a data processing framework for LHCb [4] and as part of the Gaudi-Athena framework for the ATLAS experiment. Currently, a concurrent version of Gaudi, GaudiHive, is developed, to exploit modern processing architectures.
Gaudi is realised with an object oriented design pattern and follows the rules of C++ programming. The architecture consists of components, which have well defined interfaces, functionality and interactions with another. Single components can be easily

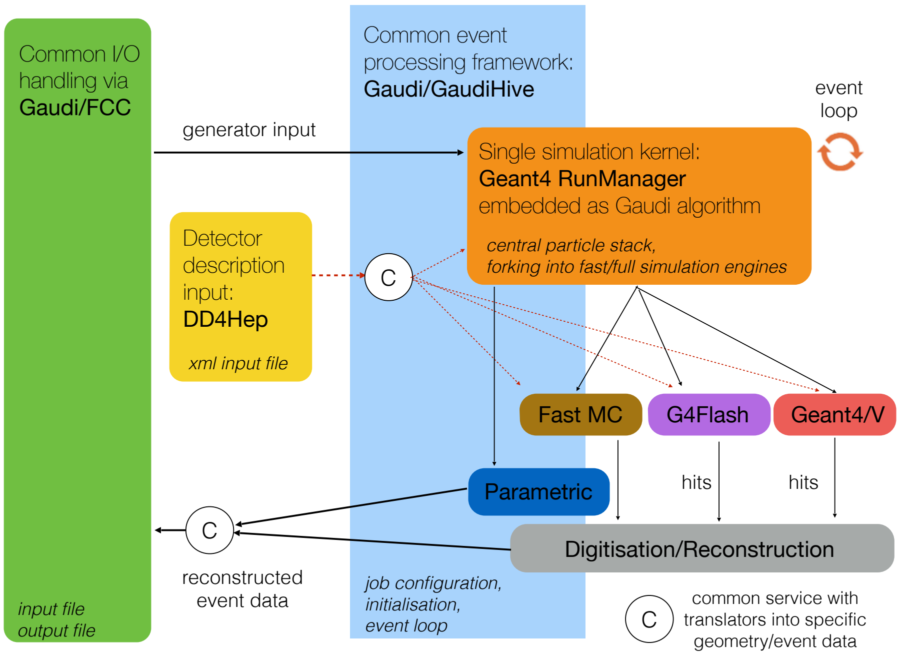

Common I/O handling via Gaudi/FCC
input file
output file
generator input
Detector description input: DD4Hep
xml input file
Common event processing framework: Gaudi/GaudiHive
job configuration, initialisation, event loop
event loop
Single simulation kernel: Geant4 RunManager embedded as Gaudi algorithm
central particle stack, forking into fast/full simulation engines
C
Fast MC
G4Flash
Geant4/V
Parametric
hits
hits
Digitisation/Reconstruction
reconstructed event data
common service with translators into specific geometry/event data

Figure 2.1: The overall picture of the simulation framework. Gaudi is used as event processing framework, which is configured by a job option file. For the geometry input DD4hep is used. Geant4 serves as a simulation kernel for all kinds of simulations (picture: A. Salzburger).

replaced or updated.
Gaudi has a central event data store, which distinguishes between transient and persistent data. The `EvtDataSvc` manages storing and retrieving objects in the event data store.
Gaudi distinguishes between objects and algorithms manipulating them. Every *algorithm* has to be invoked explicitly by the framework or other algorithms. This allows to structure the algorithms in a hierarchy. In this way top algorithms, invoking sub-algorithms can be created. The `IAlgorithm` base class represents the interface for every algorithm, providing the virtual methods `initialize()`, `execute()` and `finalize()`. Derived algorithms have to implement these three methods: the `initialize()` method is generally used to prepare the algorithm before the first execution. This can involve the retrieval of other components needed in the execution, or the setting of certain parameters. It is only called once in the program flow. The `execute()` method, however, is called once per event by the framework and thus usually carries the main processing code. Clearup and final statistics, e.g. are done in the `finalize()` method.
Services are components which provide the different features or data needed by algorithms throughout the job execution. Gaudi provides common services e.g. the services for managing the different transient stores (event data service, detector data service,...)

or the job options service, histogram service and the message service. Apart from these basic services, Gaudi offers also the possibility to introduce particular services, inheriting from the base class `IService`. Examples for services would be, e.g. a magnetic field service or access services to the conditions database.
Often, the same functionality is needed by several algorithms, probably with different configurations. For this purpose *tools* are provided [17].
A single event is processed by executing a sequence of algorithms. These algorithms read at first event data from the event data store and after the execution write back data to the store. In f Fig 2.2, an example for an event is illustrated.

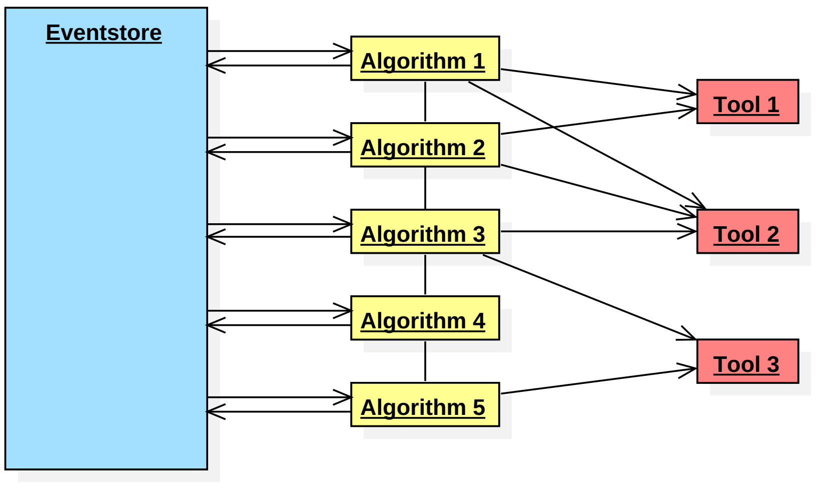


Figure 2.2: Simplified example for event processing in Gaudi. The algorithms read and write events via the `EvtDataSvc` in the Gaudi store gate, during the `execute()` call. Tools are encapsulated code blocks, which can be used by several algorithms.

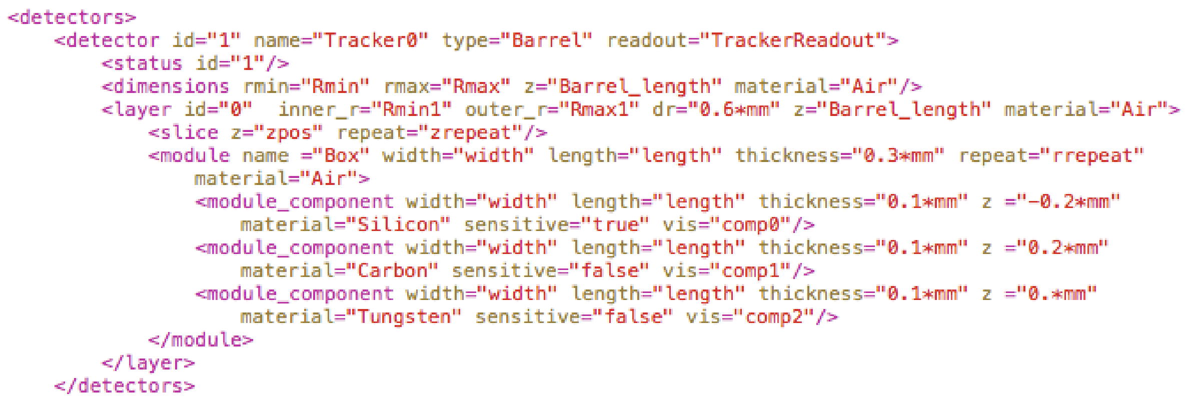
2.1.1 Job Configuration

The job option service offers the possibility to configure the program behavior via job option files. These are written in Python. In these files all services, tools and algorithms which should be used for the job have to be declared. The input and output of the event is also defined in this file. Furthermore, it offers the possibility to set the number of events or parameters and limits for algorithms [17].

2.2 The Geometry Input

Simulation, digitization and reconstruction need dedicated detector descriptions. Simulation needs a (more or less) detailed description of the detector to simulate the interactions of the particles with passive and sensitive detector material. The digitization, on the other hand, mainly needs access to detector readout properties. For the reconstruction the position of the detecting elements and a simplified description of the detector as a whole are essential. In order to facilitate the workflow for the different simulation approaches, digitization and reconstruction, a common detector description source is highly beneficial.
The detector construction of the the FCC project uses DD4hep (detector description for high energy physics) as a geometry input. It is being developed by the AIDA (Advanced European Infrastructures for Detectors and Accelerators) community [18] in the context of the linear collider experiment. DD4hep is a generic toolkit for the full detector description, including the geometry, the used material, visualization attributes, detector readout information, alignment, calibration and environmental parameters.
The basic input for building the detector are a detector description in the XML file format and a corresponding detector constructor written in C++. These two components have to be changed accordingly. The compact detector description allows a straightforward and simple declaration of the detector in the XML format. Detector constructors, supported in C++ or Python, use these XML files as an input and construct a detector in the DD4hep geometry format. The whole detector is segmented into different detector parts, e.g. barrels and endcaps which describe different sub-detectors. Since these sub-detectors are built differently, they need different detector constructors. In this way one detector description in XML can use various detector constructors, needed for the different kinds of detector parts.
The separation of the description in XML and the actual geometry building modules, implemented within the framework, has the advantage that geometry parameters can be quickly changed without the need of recompiling the detector builders. Furthermore, if the basic detector concept has been chosen and the constructors are made in a generic way, sub-detectors, layers, modules or components can be added easily without changing the code.
The different parts and properties of the detector are described and identified by tags in the XML file. Several sub-detectors, for example, are each described in the detector section, enclosed by the `detectors` tag (see Fig 2.3).

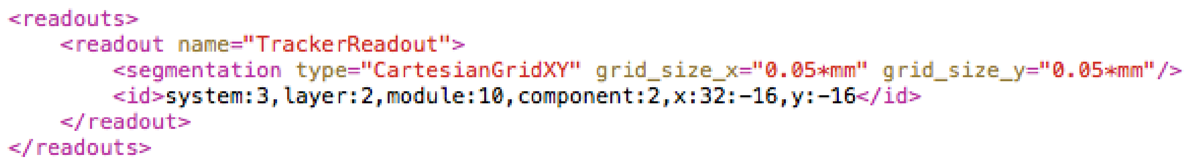
The sensitive components of a detector are segmented into smaller readout channels. This readout segmentation needs to be described within the detector description in order to assign the hits to a position on the component. In the XML file this is described in the section `segmentation` within the `readout` tag (see Fig 2.4). It can be described in several cartesian grids (xy, xz, yz, xyz), in a polar grid or in a cylindrical grid. In the XML file bit fields have to be reserved for the different kinds of detector elements and the grid. These bit fields are later combined to identify each detector element uniquely.
DD4hep reuses already existing components. The geometry, e.g. is based on the ROOT [21] geometry package, using logical volumes as a basis and partly extending it for fur-



```
<detectors>
    <detector id="1" name="Tracker0" type="Barrel" readout="TrackerReadout">
        <status id="1"/>
        <dimensions rmin="Rmin" rmax="Rmax" z="Barrel_length" material="Air"/>
        <layer id="0"  inner_r="Rmin1" outer_r="Rmax1" dr="0.6*mm" z="Barrel_length" material="Air">
            <slice z="zpos" repeat="zrepeat"/>
            <module name ="Box" width="width" length="length" thickness="0.3*mm" repeat="rrepeat"
                material="Air">
                <module_component width="width" length="length" thickness="0.1*mm" z ="-0.2*mm"
                    material="Silicon" sensitive="true" vis="comp0"/>
                <module_component width="width" length="length" thickness="0.1*mm" z ="0.2*mm"
                    material="Carbon" sensitive="false" vis="comp1"/>
                <module_component width="width" length="length" thickness="0.1*mm" z ="0.*mm"
                    material="Tungsten" sensitive="false" vis="comp2"/>
            </module>
        </layer>
    </detectors>
```

Figure 2.3: Code fragment of a detector description. The different parts of the subdetector are enclosed by tags (`detector`, `layer`, `module`, `component`,...). The type of the detector identifies, which detector constructor should be used.

ther functionality. Hence, an easy visualization of the DD4hep geometry is guaranteed via the ROOT visualization displays. In addition, a straight forward integration of the geometry in Geant4, which is used for the full simulation is already provided (see Section 2.5). The DD4hep geometry needs also to be translated into the simplified detector geometry used for the reconstruction and potentially for the fast simulation. An automatic translation mechanism between the geometry representations has been implemented and is further described in Chapter 4.1.
For this conversion the full detector information and the geometry information needs to be accessed. DD4hep has implemented the Detector Description Data Hub (LCDD), which provides management, bookkeeping and ownership to the model instances.
In the detector description model of DD4hep any detector is a tree of instances of the so-called `DetElement` class. This `DetElement` class provides all needed detector information, e.g. readout segmentation, geometrical information, environmental conditions. This tree is parallel to the volume tree, which provides the `TGeoVolumes` and their placements. The relation between these two trees is one-directional, i.e. every volume can be accessed via its corresponding `DetElement`, but not vice versa (see Fig 2.5). Not every



```
<readouts>
    <readout name="TrackerReadout">
        <segmentation type="CartesianGridXY" grid_size_x="0.05*mm" grid_size_y="0.05*mm"/>
        <id>system:3,layer:2,module:10,component:2,x:32:-16,y:-16</id>
    </readout>
</readouts>
```

Figure 2.4: Code fragment of a readout description. The bit fields to uniquely identify every position on a sensitive component have to be reserved in the section `id`.

supporting material will be declared as a detector element, hence, the geometrical tree can have a deeper hierarchy structure. In order to access both, detector and geometry information, one has to navigate through the detector tree during the translation process. The `DetElement` can also be extended, to add specific features or to access information. This extension mechanism is provided from DD4hep to attach additional information to certain objects to allow customized use for certain implementations. The LCDD, the `SensitiveDetector` class, describing the sensitive detector parts, and the `DetElement` can be extended. An essential tool for the geometry is the `VolumeManager`

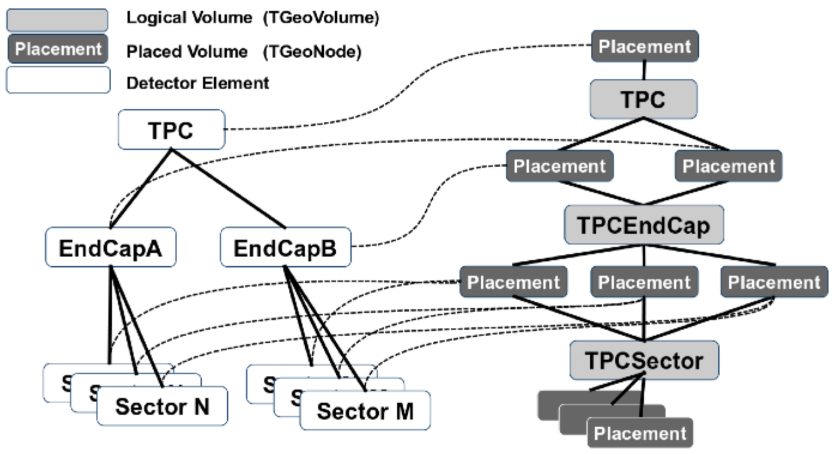


Figure 2.5: On the left side one can see the `DetElement` tree which holds the hierarchy of the detector elements. On the right the volume tree is illustrated, where nodes (volumes with placement) hold logical volumes. From the `DetElement` the placement and herewith the logical volume can be accesssed but not vise versa [19].

which combines the unique volume identifier with the placement of the certain volume. In this way it allows to access volumes by its volume ID [19].

2.3 The Simulation Environment

To enable common steering of full, fast and parametric simulation, Geant4 is used as the simulation kernel. The simulation kernel is steering the simulation by receiving the particles and sending them to the different simulations depending on the detector region and particle type defined by the user via the job configuration. It manages the particle stack and does bookkeeping within the Geant4 structure.
In Geant4 the entire simulation is managed via the so-called `G4RunManager`. It controls the Geant4 kernel, the program flow, the run and the event loop. A `G4Run` represents the highest instance in the process hierarchy and can contain several events.
To run a simulation in Geant4, one must create an instance of the run manager in the

user's main program. It is responsible for the initialization of a run, including detector construction, the physics processes, primary particle generation and afterwards the invocation of the run.
To integrate the simulation into the Gaudi based framework and to enable also external simulation toolkits to be used (based on other geometries and using different software) within the Geant4 environment, a dedicated run manager, inheriting from `G4RunManager` was implemented. It is embedded as Gaudi algorithm in order to invoke the simulation from Gaudi. The choice to make the whole simulation handling within the Geant4 environment was to allow a most coherent simulation and taking advantage of the already existing feature of detector envelopes, i.e. detector regions.
In the Geant4 environment a fast simulation can be realized by attaching a `G4FastSimulationManager` to different regions of the detector. It manages the different simulation models for the different particles. All daughter volumes of the defined detector region will use the parameterizations defined for certain particles from the current `G4FastSimulationManager` [23].

2.4 The Event Data Model

A coherent simulation framework only unfolds its full benefit, if it is supported by a common Event Data Model that is shared between the different simulation approaches. Even when fast simulation is used and higher level output objects such as tracks particles or jets are emulated, it is necessary that those objects are created in the standardized output format one would expect from running the full simulation–digitization–reconstruction-chain. This facilitates the switching between the simulation techniques while having little effect on the analysis chain.
For FCC, the Event Data Model consists of a description of the data structures stored in an event. These data structures are simple C structs or C++ classes, called POD (Plain Old Data). ROOT is used for handling the file input/output and event persistency [24]. The FCC software community has decided to implement a very clean, minimal EDM that allows every extension and flexibility at one hand, but also fulfills necessary performance figures on the other. The actual EDM classes are auto-generated, after being defined on user input: The classes within the data model are defined in yaml-file. A code generator translates them in the C++ code and the directory structure. For every datatype, there are three classes generated automatically: the POD itself, a handle (containing a pointer to an existing POD) and a collection of handles. The PODs are always accessed via its handles. References to another POD can be stored in a POD as handles.
The Collections inherit from the Gaudi class `DataObject` in order to make the data model compatible with the `EvtDataSvc` of Gaudi [24].

2.4.1 The Tracking Event Data Model

Track reconstruction requires in general the most complex Event Data Model of all reconstruction steps. Based on the experience of the EDMs of the LHC experiments and on the LCIO (Linear Collider I/O) a tracking Event Data Model has been proposed (see Fig 2.6) and the implementation of a first setup was part of this master thesis.
Extending the `BareHit` class, a `TrackHit` has been proposed, which has a start and an end point of the track hit in the sensitive material, plus a transient pointer to the corresponding `Surface` of the reconstruction geometry (described in Section 3.3). The latter is to enable local to global transformations and vice versa.
The `TrackCluster` class, extending the `BareCluster`, holds containers of `TrackHits` grouped together by a clustering algorithm, a local measurement and a local error covariance matrix. After a calibration the `CalibratedTrackCluster`, extending the untouched track cluster can hold a container of `TrackClusters` or one `TrackCluster`. Calibrated measurements have been proven to be a very good concept during LHC operation: they may take detector conditions, local deformations or even misplacements into account.
A `Track` class exists, holding a container of `TrackStates`. Each `TrackState` has a `TrackParameterisation`, describing the parameters needed for a track. This includes a local measurement on the surface, the charge in terms of momentum, the spatial angles θ and φ which describe the global orientation of the track and a covariance matrix to describe the uncertanties and correlations of the track parameters. Finally this leads to the `TrackParticle`, which is composed of a `BareParticle` and a `Track`.

2.5 Integration of the Components in the Framework

In this Section the communication between the different components and the combination to one coherent framework, which was a main task of this thesis, is described.
For retrieving the geometry input within the Gaudi framework a geometry service was created. This service `GeoSvc` inheriting from an interface `IGeoSvc` provides both, the Geant4 and the DD4hep geometry. The geometries are built in the initialization of the service, i.e. as soon as one component evokes this service, the geometry is built automatically.
After creating an instance of the `LCDD`, which is the interface to the DD4hep geometry and applying the XML file, the `DetElement` of highest order (world) is accessed via this instance, allowing to access the entire detector tree. In addition an instance of the volume manager is applied to set the unique volume identification.
To create the Geant4 geometry, the `Geant4DetectorConstruction` of DD4hep is invoked via the LCDD instance. It is then internally converted into the Geant4 geometry using the `Geant4Converter`, which is also a DD4hep class. The `Geant4DetectorConstruction` inherits from the Geant4 `G4VUserDetectorConstruction`, in order to satisfy the necessary integration into the Geant4 data flow.
For building the reconstruction geometry (see Chapter 3) a dedicated service was de-

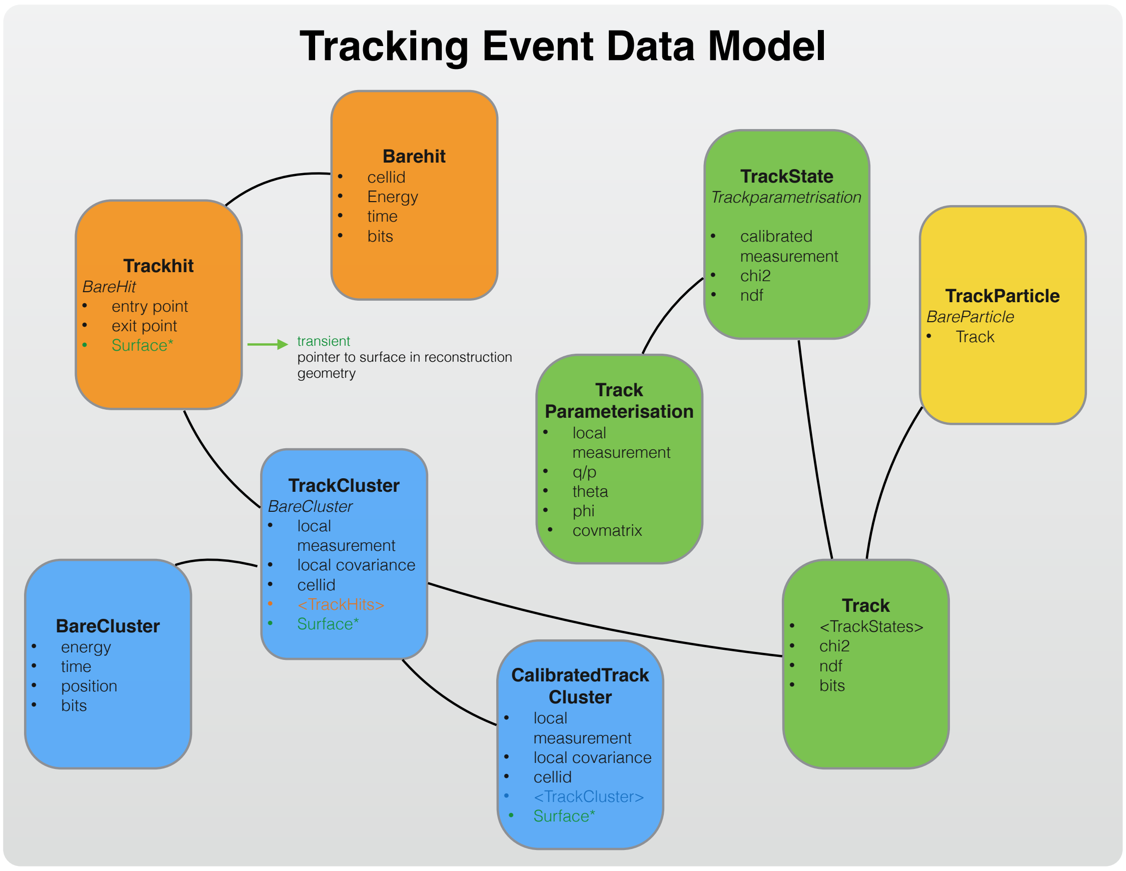


Figure 2.6: The current version of the FCC tracking Event Data Model. The bare classes, holding the basic information of the FCC EDM are extended by track classes, adding members needed especially for tracking. The `Surface` is the link to the reconstruction geometry.

signed (`ClassicalRecoGeoSvc`), which translates the DD4hep geometry into a simplified reconstruction geometry. The reconstruction geometry is built upon request. In order to build and access the reconstruction geometry the method `getRecoGeometry()` has to be called. It returns a pointer to a `ContainerVolume`, which is a class of the reconstruction geometry (see Section 3.5).
Since a complete general translation mechanism into the reconstruction geometry was not possible, the service is implemented with an interface (`IRecoGeoSvc`). In this way other implementations of a translation for other detector types can be introduced. For the current implementation (`ClassicalRecoGeoSvc`), special conventions have to be met (described in Chapter 4.1).
If only the reconstruction geometry is used in an algorithm, only the `ClassicalRecoGeo-Svc` has to be instantiated. It invokes the service for building and retrieving the DD4hep geometry itself (see Fig 2.7).

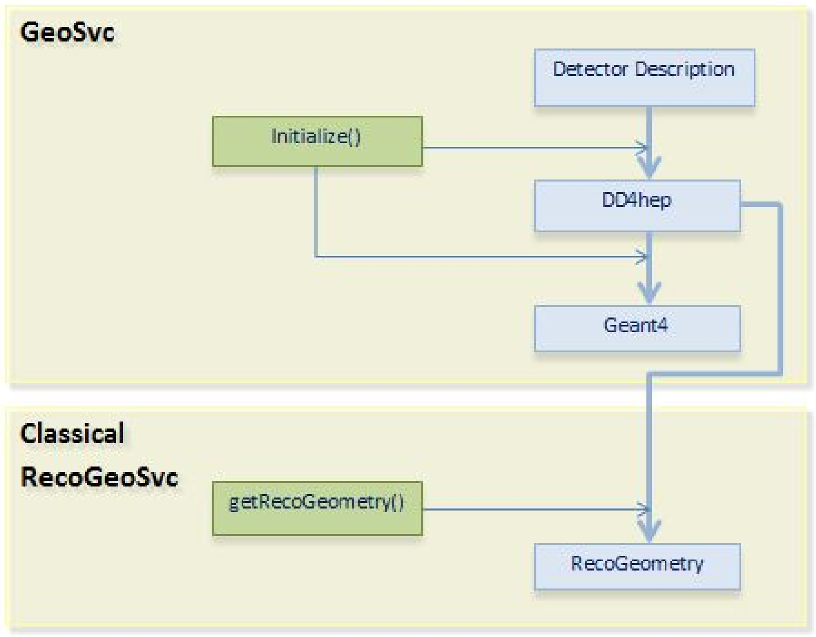


Figure 2.7: The Geant4 geometry can be received from the `GeoSvc`, which builds the DD4hep and the Geant4 geometry in the initialization phase. To obtain the reconstruction geometry an algorithm has to explicitly request to the `ClassicalRecoGeoSvc`. The DD4hep geometry is then internally retrieved and translated.

To fully understand the concepts of the translation an introduction into the general reconstruction geometry is needed, which follows in the next Chapter. A more detailed description of the translation mechanism is described in Chapter 4.1.

Chapter 3
Reconstruction Geometry

The reconstruction geometry is a simplified geometry which is used during reconstruction and for fast simulation. While in detector simulation an accurate description of the detector geometry is needed to correctly follow the path of the particle through the detector regions and to simulate its interaction with passive and active material, in the reconstruction application, geometry features can usually be simplified. This is done to improve the execution speed and to take into account, that only stochastic effects can be considered for reconstruction purposes. In simulation, e.g. when tracking a particle through the detector, it is a great importance where the particle hits what kind of material, while in (track) reconstruction only an averaged picture of the detector is needed in order to include potential disturbances from material interactions as process noise.
In general, the reconstruction geometry describes the sensitive material to exact detail in order to interpret hits and measurements. Passive material, however, is usually approximated. Having a fast navigation scheme through the reconstruction geometry is essential, since it is a very repetitive task and can end up easily in taking significant CPU resources. For this reason, having an initial design where the navigation is built into the geometry structure is highly beneficial.
The sensitive detector parts are described as surfaces (Section 3.3) in the reconstruction geometry. These are infinitesimal thin objects allowing intersection and build the reference frame for track parameterisation. On each surface the supporting material is described approximately, to enable material effects, e.g. energy loss or multiple scattering. Layers (Section 3.4) hold these surfaces at their specific position and the layers are surrounded by volumes. Volumes (Section 3.5) are enclosed by boundary surfaces, which fulfill two purposes. First, the possibility to describe material of supporting structures and second, navigation between the volumes can be enabled, because these boundary surfaces point to their next/previous volume. To navigate within one volume, the layers are pointing to their next/previous layer.
The main concepts of the reconstruction geometry have been taken from the ATLAS Tracking Geometry [25], but have been adapted to fit into the FCC software framework. To enable the translation from the TGeo based DD4hep geometry and to have a first simple setup, the general geometry has been implemented.
In order to ease the translation from the DD4hep geometry into the reconstruction geometry all the classes can be constructed of `TGeoShapes` (describing the shape) and `TGeoNodes` (describing the placement). For coherency reasons the readout segmentation

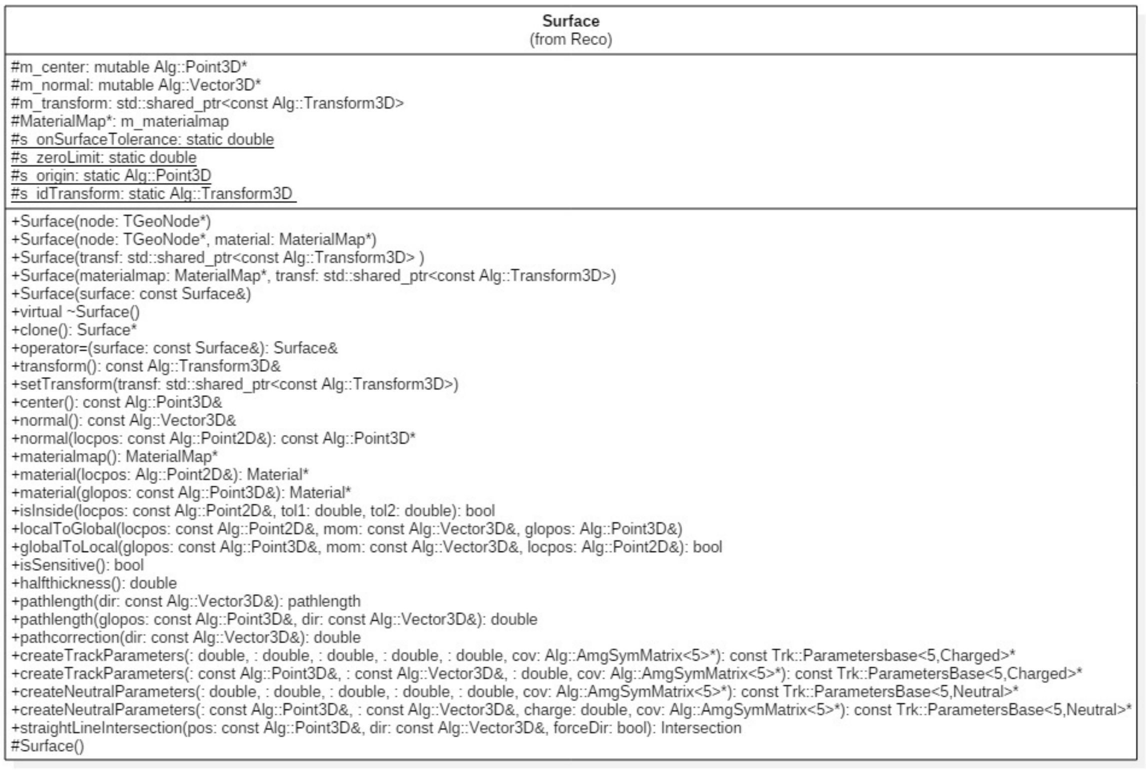
of DD4hep will be used.
For a proper memory management only `std::shared_ pointers` of the standard C++ library to the instances of the geometry are used.

3.1 Frame Definitions

In this Chapter the following frame definitions are used: The global frame is a 3D cartesian frame with origin at the center of the detector. The z-axis was chosen to be the beam tube–axis. The azimuthal angle θ describes the angle of a particle (i.e. its momentum vector) relative to the beam axis and the polar angle φ describes the rotation angle around the beam axis. The pseudo rapidity is described by $\eta = -\ln \tan \frac{\theta}{2}$ [27], which almost corresponds to rapidity and is easier to sustain in measurements.
Each geometrical node has a local frame and a transformation matrix which describes the relative positioning of this node to the global frame. A transformation matrix is a $4x4$ matrix containing a $3x3$ matrix, which describes the rotation of the object with respect to the global frame and a three dimensional vector which describes the translation of the node relative to the origin of the global frame. Local to global transformations transform a local vector or point into the global frame by applying the rotation of the transformation matrix to the vector and translate the point by the translation vector. The global to local conversion uses the inverse of the transformation matrix to transform directions and positions into the local frame of a placed geometrical object.

3.2 Helper Classes and Utilities

In this Section the basic classes needed later by the geometry classes are described. In the `Alg` (Algebra) namespace the algebra library and the geometric transformation library are declared, which were taken directly from ROOT. These include vector definitions, matrix representations and other algebra classes.
In order to store and access geometrical objects on a certain position the classes `BinUtility` and `BinnedArray` were directly implemented from the ATLAS software. The `BinUtility` is used to generate a given number of bins in an interval. The bins can be equidistant, bi-equidistant or arbitrary. In latter case, a vector with the bin values has to be provided to the constructor, which defines the boundaries of the array.
Out of a vector of objects together with the center position of this object and a corresponding `BinUtility` a `BinnedArray` can be created. Different implementations of the `BinnedArray` class, concerning the dimension have been provided, for one (`BinnedArray1D`) or two dimensional (`BinnedArray2D`) binning as well as for anti-symmetric two dimensional binning (`BinnedArray1D1D`). Every object in the binned array can be addressed over its bin. This two classes are basic components for the following classes.



```
Surface
(from Reco)

#m_center: mutable Alg::Point3D*
#m_normal: mutable Alg::Vector3D*
#m_transform: std::shared_ptr<const Alg::Transform3D>
#MaterialMap*: m_materialmap
#s_onSurfaceTolerance: static double
#s_zeroLimit: static double
#s_origin: static Alg::Point3D
#s_idTransform: static Alg::Transform3D

+Surface(node: TGeoNode*)
+Surface(node: TGeoNode*, material: MaterialMap*)
+Surface(transf: std::shared_ptr<const Alg::Transform3D> )
+Surface(materialmap: MaterialMap*, transf: std::shared_ptr<const Alg::Transform3D>)
+Surface(surface: const Surface&)
+virtual ~Surface()
+clone(): Surface*
+operator=(surface: const Surface&): Surface&
+transform(): const Alg::Transform3D&
+setTransform(transf: std::shared_ptr<const Alg::Transform3D>)
+center(): const Alg::Point3D&
+normal(): const Alg::Vector3D&
+normal(locpos: const Alg::Point2D&): const Alg::Point3D*
+materialmap(): MaterialMap*
+material(locpos: Alg::Point2D&): Material*
+material(glopos: const Alg::Point3D&): Material*
+isInside(locpos: const Alg::Point2D&, tol1: double, tol2: double): bool
+localToGlobal(locpos: const Alg::Point2D&, mom: const Alg::Vector3D&, glopos: Alg::Point3D&)
+globalToLocal(glopos: const Alg::Point3D&, mom: const Alg::Vector3D&, locpos: Alg::Point2D&): bool
+isSensitive(): bool
+halfthickness(): double
+pathlength(dir: const Alg::Vector3D&): pathlength
+pathlength(glopos: const Alg::Point3D&, dir: const Alg::Vector3D&): double
+pathcorrection(dir: const Alg::Vector3D&): double
+createTrackParameters(: double, : double, : double, : double, : double, cov: Alg::AmgSymMatrix<5>*): const Trk::Parametersbase<5,Charged>*
+createTrackParameters(: const Alg::Point3D&, : const Alg::Vector3D&, : double, cov: Alg::AmgSymMatrix<5>*): const Trk::ParametersBase<5,Charged>*
+createNeutralParameters(: double, : double, : double, : double, : double, cov: Alg::AmgSymMatrix<5>*): const Trk::ParametersBase<5,Neutral>*
+createNeutralParameters(: const Alg::Point3D&, : const Alg::Vector3D&, charge: double, cov: Alg::AmgSymMatrix<5>*): const Trk::ParametersBase<5,Neutral>*
+straightLineIntersection(pos: const Alg::Point3D&, dir: const Alg::Vector3D&, forceDir: bool): Intersection
#Surface()
```

Figure 3.1: The `Surface` class of the reconstruction geometry, the surface builds the core geometrical node of the tracking geometry. All other objects are either extensions of surfaces or built from those.

3.3 The Surface Classes

The surface representation is fundamental for the reconstruction and the simulation. They are needed to produce hits and to generate tracks. It will also be stored as a transient pointer in the Event Data Model, to enable local to global transformations and backwards.

The `Surface` class is a virtual class which serves as a parent for certain surface implementations. It contains a three dimensional global center position, a three dimensional global normal vector to the surface and a transformation matrix, describing the global orientation, containing the rotation and the translation of the surface.

The surface contains a material map, which is a two dimensional grid, offering the possibility to describe an existing material distribution with a good approximation. It is constructed from a three dimensional transformation matrix or to simplify the translation from the DD4hep geometry, from a ROOT `TGeoNode`. If the surface contains material also the material map must be provided at construction.

The `Surface` class provides the basic functionalities needed for tracking. It implements

the transformation functions which transform local to global coordinates and vice versa. A method checks whether a local given two dimensional position is inside the surface. It provides methods for creating track parameters. A straight line intersection method has been provided. It finds the closest possible intersection in a straight line, from the given global position in a specific global direction. The struct `Intersection` is returned, which contains the path length (the length from the given point to the surface), the global intersection position and a boolean, returning whether the position actually is on the surface.
Geometrically, a surface is a two dimensional construct, however in the detector description it can represent a physical volume, a sensor or detector module. Therefore a thickness can be assigned to the `Surface`. When a particle is traversing through the surface it passes a certain amount of material. The method `pathlength()` has been implemented to calculate this distance depending on the direction.
A detector module will have sensitive parts, i.e. sensors, which have electronic readout elements and are detecting the particles. To describe these sensitive parts the class `SensitiveSurface` has been introduced, described in the next Section. In order to distinguish between sensitive and non sensitive surfaces a boolean method `isSensitive()` has been introduced.
The basic `Surface` class has currently four implementations, for four different shapes: `PlaneSurface`, `CylinderSurface`, `DiscSurface`, and `TrapezoidSurface`. These classes hold their boundary description as additional members and are implementing the virtual methods of the `Surface` class in their local coordinate system. The plane and the trapezoid surface class are using cartesian coordinates x and y. The `CylinderSurface` uses a combination of cylindrical coordinates $r \cdot \varphi$ and z and the `DiscSurface` uses polar coordinates r and φ. All of these four classes can either be constructed by its dimensions or specially for the translation mechanism, directly from the ROOT `TGeoShape` classes. A `PlaneSurface` can be constructed from a `TGeoBBox`, a trapezoid from a `TGeoTrd2` and for the `CylinderSurface` and the `DiscSurface` a `TGeoConeSeg` is used.

3.3.1 Sensitive Surfaces and Readout Segmentation

The `SensitiveSurface` classes represent the sensitive (measuring) parts of the detector. This parts have a readout structure, i.e. depending on the kind of detector module and the granularity, they are divided into grids. Planar detectors are usually segmented as pixels or strip detectors, hence, they are segmented in two or one dimensional bins. Every bin has a unique identification on this surface, the so-called *cellID*, while the surface itself is also uniquely identified in the detector by its *volumeID*. In this way, when the detector receives a signal in a certain readout channel, one can automatically determine the position and the module where the signal occurred.
The global unique identification is a 64 bit integer. The first 32 bits are used for the global identification of the surface in the detector, the other 32 bit remain for the channel identification on the surface.
For coherence reasons, the readout segmentation was directly taken from DD4hep. The `SensitiveSurface` class holds a pointer to the DD4hep `Segmentation` class [20].This

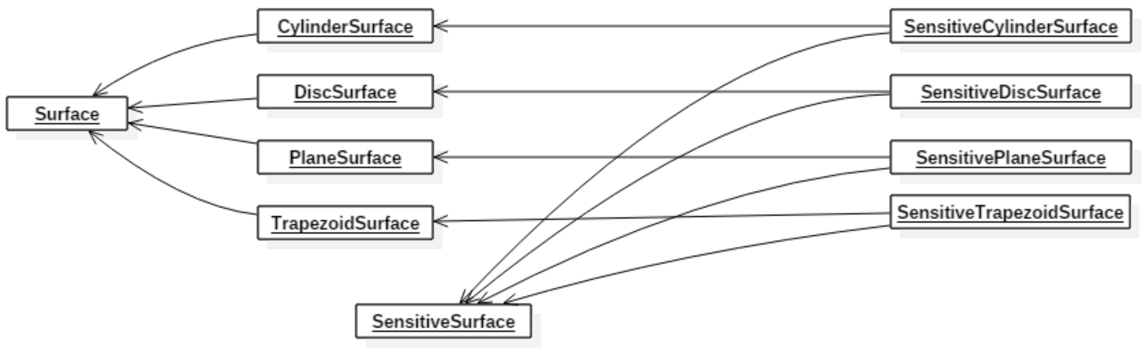


Figure 3.2: The `SensitiveSurface`, has four geometrical implementations, inheriting as well from the `Surface` class to realize a surfaces with a readout structure.

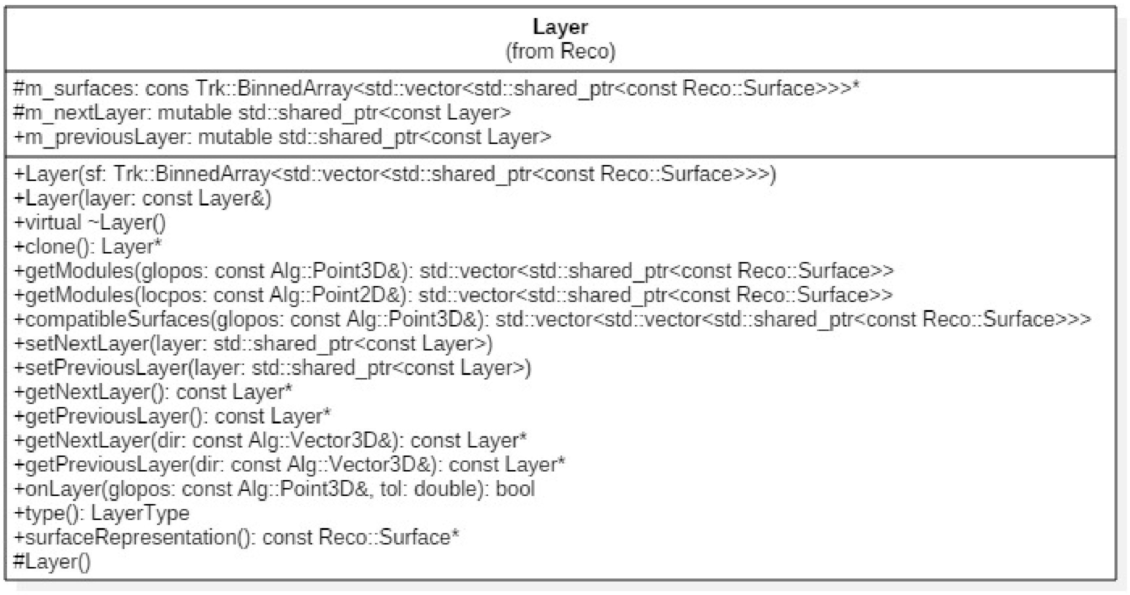
instance of the readout segmentation is retrieved over the translation mechanism, described in the next Chapter.
The sensitive surface class implements methods to calculate the cellID by its local position on the surface and to obtain the local position by providing the associated cellID. Furthermore, one can retrieve all surrounding neighbor cellIDs by calling the method neighbors(). This is usefull for e.g. channel clustering.
The `SensitiveSurface` class is a virtual class. It has currently four implementation, which again represent the geometries plane, cylinder, disc and trapezoid. This implementations are inheriting as well from the basic `Surface` class (see Fig 3.2), for instance the class `SensitivePlaneSurface` is inheriting from `SensitiveSurface` and `PlaneSurface`. In this way a surface can have a readout structure.

3.3.2 BoundarySurfaces

A `Surface` can be extended to be used as a boundary of a volume, described in the Section 3.5. This classes are needed for enabling navigation within the geometry. The boundary surfaces hold pointers to the next and previous volume. In case there is more than one volume attached, a `BoundarySurface` holds one dimensional `BinnedArray`s to the next and previous volumes attached.
Within a detector the direction to the next volume points always from the inside (the center point) to the outside – this was chosen as a convention.
It has methods to access these volumes. The `BoundarySurface` is as well realized in different geometrical shapes: disc, plane, cylinder and trapezoid. It contains a pointer to `Material` (see Section 3.6), to enable boundary surfaces to have support material.

3.4 The Layer Classes

The `Layer` class was introduced for navigation reasons. It contains `Surfaces` (detector modules) and pointers to the next and previous layer. The surfaces are held by a `BinnedArray` of a standard vector of surfaces. In this way it is possible to place more than one surface layered in one bin.
Methods to retrieve the surfaces have been implemented. Therefore the specific surfaces



```
Layer
(from Reco)

#m_surfaces: cons Trk::BinnedArray<std::vector<std::shared_ptr<const Reco::Surface>>>*
#m_nextLayer: mutable std::shared_ptr<const Layer>
+m_previousLayer: mutable std::shared_ptr<const Layer>

+Layer(sf: Trk::BinnedArray<std::vector<std::shared_ptr<const Reco::Surface>>>)
+Layer(layer: const Layer&)
+virtual ~Layer()
+clone(): Layer*
+getModules(glopos: const Alg::Point3D&): std::vector<std::shared_ptr<const Reco::Surface>>
+getModules(locpos: const Alg::Point2D&): std::vector<std::shared_ptr<const Reco::Surface>>
+compatibleSurfaces(glopos: const Alg::Point3D&): std::vector<std::vector<std::shared_ptr<const Reco::Surface>>>
+setNextLayer(layer: std::shared_ptr<const Layer>)
+setPreviousLayer(layer: std::shared_ptr<const Layer>)
+getNextLayer(): const Layer*
+getPreviousLayer(): const Layer*
+getNextLayer(dir: const Alg::Vector3D&): const Layer*
+getPreviousLayer(dir: const Alg::Vector3D&): const Layer*
+onLayer(glopos: const Alg::Point3D&, tol: double): bool
+type(): LayerType
+surfaceRepresentation(): const Reco::Surface*
#Layer()
```

Figure 3.3: The `Layer` Class of the reconstruction geometry. Layers are used for storing surfaces to access them at every given point and needed for navigation.

at a given global position as well as a vector of compatible surfaces, around a given global position can be accessed. It provides also a `Surface` representation, since the derived classes `DiscLayer` and `CylinderLayer` inherit from the `Layer` and the `Surface` class (see Fig 3.4). In this way the functionalities of the `Surface` class, e.g. the straight line intersection, the transformations, the normal vector, or the access to the center position are guaranteed. Finally a `NavigationLayer` class has been implemented. This shapeless class serves for navigation only, by holding pointers to the next and previous layers.

3.5 The Volume Classes

The `Volume` (Fig 3.5) class is the base class to describe geometrical volumes. It contains basic members like a transformation matrix, which exactly locates the volume globally to a world volume and also a rotation matrix. It is bordered by `BoundarySurfaces`, pointing to the next and previous `Volume`, stored in a vector of `BoundarySurfaces`. A

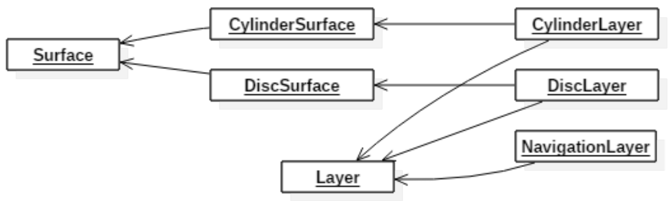


Figure 3.4: The `Layer` class has currently three implementations, representing the geometrical shapes disc and cylinder and a shapeless implementation for navigation reasons. These inherit as well from the `Surface` implementations to give layers surface functionality.

`Volume` can contain layers, implemented as a `BinnedArray` of Layers. To enable navigation at every position in the volume, the volume has to be completely filled with layers. The empty spaces between the material layers (layers containing modules), should be filled with navigation layers. The purpose of the navigation layers is, that every single point in the detector can be associated to a layer and embedding volume. This is necessary to guarantee that the navigation through the entire detector can be done.

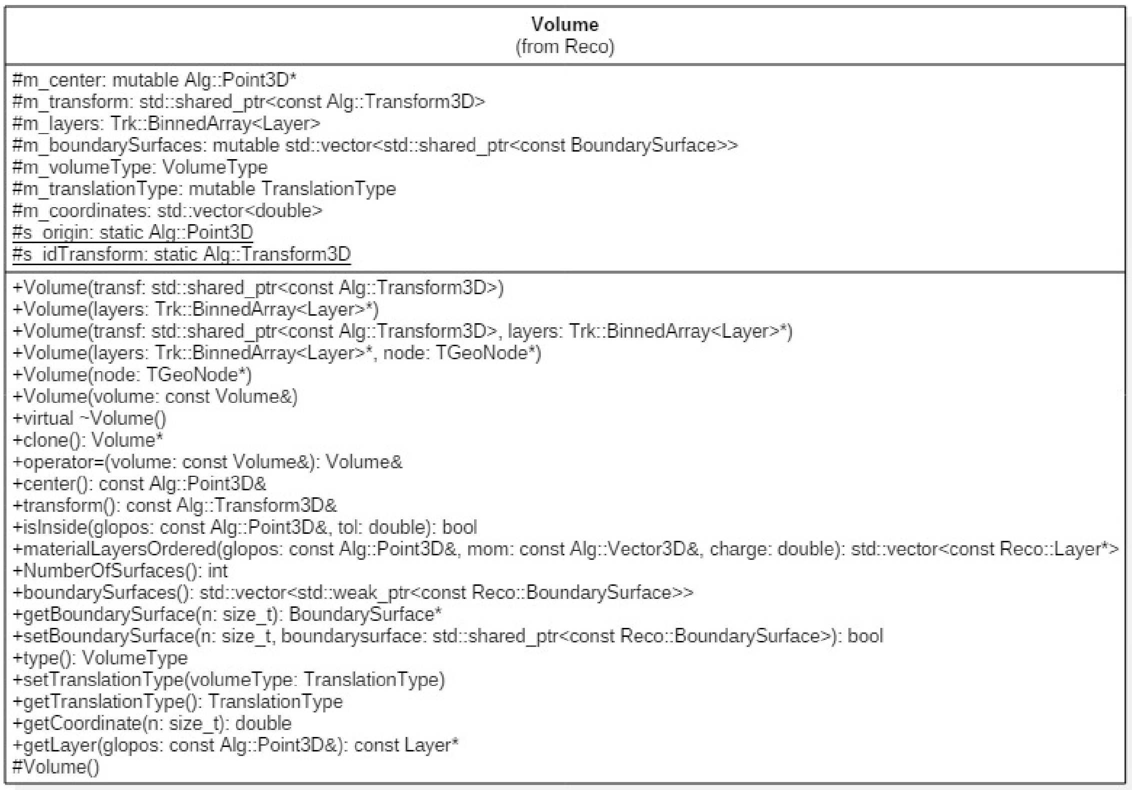
Basic methods returning the center position, the transformation matrix, the dimensions or a check whether a global position is inside a volume, within a certain tolerance, have been implemented. It is also possible to access and set the boundary surfaces. For the simulation the method `materialLayersOrdered()` is needed. It returns all material layers in a vector, starting from a point in a given direction.

To enable the translation from the DD4hep geometry (see Chapter 4.1), the volume can have a translation type. This type can currently be a barrel, an endcap at the negative or the positive side of the barrel or a container. It is needed to distinguish the different kinds of volumes during the translation phase.

Currently there only exists one shape implementation of the basic volume class, the `CylinderVolume`. This volume is sufficient to describe most classical collision experiments. The class `ContainerVolume` allows to group together several sub-volumes in one container volume. After nesting the sub-volumes in several containers the whole detector geometry can be accessed over one world volume, which will be a `ContainerVolume`. It inherits from the `Volume` base class to provide the basic volume functionalities and holds a `BinnedArray` of volumes. Furthermore the class `ContainerCylinderVolume`, inheriting from `ContainerVolume` and from `CylinderVolume` has been implemented (see Fig 3.6).

3.6 Material Description

A material description is needed to describe the physical interaction processes. Multiple scattering needed in reconstruction and simulation is described by thickness in terms of



```
Volume
(from Reco)

#m_center: mutable Alg::Point3D*
#m_transform: std::shared_ptr<const Alg::Transform3D>
#m_layers: Trk::BinnedArray<Layer>
#m_boundarySurfaces: mutable std::vector<std::shared_ptr<const BoundarySurface>>
#m_volumeType: VolumeType
#m_translationType: mutable TranslationType
#m_coordinates: std::vector<double>
#s_origin: static Alg::Point3D
#s_idTransform: static Alg::Transform3D

+Volume(transf: std::shared_ptr<const Alg::Transform3D>)
+Volume(layers: Trk::BinnedArray<Layer>*)
+Volume(transf: std::shared_ptr<const Alg::Transform3D>, layers: Trk::BinnedArray<Layer>*)
+Volume(layers: Trk::BinnedArray<Layer>*, node: TGeoNode*)
+Volume(node: TGeoNode*)
+Volume(volume: const Volume&)
+virtual ~Volume()
+clone(): Volume*
+operator=(volume: const Volume&): Volume&
+center(): const Alg::Point3D&
+transform(): const Alg::Transform3D&
+isInside(glopos: const Alg::Point3D&, tol: double): bool
+materialLayersOrdered(glopos: const Alg::Point3D&, mom: const Alg::Vector3D&, charge: double): std::vector<const Reco::Layer*>
+NumberOfSurfaces(): int
+boundarySurfaces(): std::vector<std::weak_ptr<const Reco::BoundarySurface>>
+getBoundarySurface(n: size_t): BoundarySurface*
+setBoundarySurface(n: size_t, boundarysurface: std::shared_ptr<const Reco::BoundarySurface>): bool
+type(): VolumeType
+setTranslationType(volumeType: TranslationType)
+getTranslationType(): TranslationType
+getCoordinate(n: size_t): double
+getLayer(glopos: const Alg::Point3D&): const Layer*
#Volume()
```

Figure 3.5: The `Volume` class of the reconstruction geometry. Volumes describe sub detector elements which surround detector modules belonging to a certain part of the detector.

radiation length t/X_0 (energy loss through radiative effects). Hadronic interactions are described in the simulation by thickness in units of the nuclear interaction length t/Λ_0 (mean free path), they are not taken into account at reconstruction, but serve as input for fast simulation. Finally the mass number A, the atomic number Z and the density are used to calculate the energy loss in the material.
The `Material` class describes the basic properties of material: A, Z, density, t/X_0, t/Λ_0 and the percentage of sensitive material.
A `MaterialMap` serves for describing the material distribution of a `Surface` on a two dimensional grid. It is implemented as a standard map of a pair of bins and the corresponding material. One can access the material on the surface by its local position.
Two implementations of the `MaterialMap` class exist, `MaterialMap2D` and `MaterialMap1D1D`. `MaterialMap2D`, for symmetric binning, must be constructed by a two dimensional `BinUtility` and a map of a pair of bins and a pointer to its corresponding material. The `MaterialMap1D1D` containing a one dimensional leading `BinUtility` and a vector of `BinUtilities`, for every bin of the leading `BinUtilities`, can be used for asymmetric two dimensional binning.

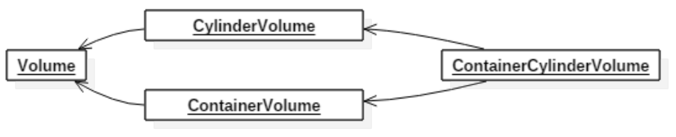


Figure 3.6: The virtual `Volume` class has currently one shape implementation, the `CylinderVolume`. The class `ContainerVolume` extends the `Volume`, to allow to group volumes together in one surrounding volume, with the `ContainerCylinderVolume` as a shape implementation.

Chapter 4

Common Geometry Building

In order to facilitate detector prototyping without the need for work-intensive adaptions, a single source geometry input for full, fast simulation and reconstruction is essential. Since the different steps require different geometry descriptions, dedicated converters from the DD4hep input have to be provided. For the Geant4 simulation toolkit, a converter into the Geant4 geometry library already exists as part of the DDG4 implementation, see Section 2.5. For the reconstruction geometry, however, this conversion had to be established. This Chapter describes the translation from the DD4hep geometry to the reconstruction geometry. To test this translation a first simple test tracker was introduced. Finally, the geometry is compared with the DD4hep and the full Geant4 geometry.

4.1 Translation from DD4hep Geometry into the Reconstruction Geometry

As mentioned in Section 2.5 the translation from the full DD4hep geometry into the simplified reconstruction geometry is done by the `ClassicalRecoGeoSvc`, which implements the `IRecoGeoSvc` interface. Since a complete general translation without any convention is not possible, specializations of the `IRecoGeoSvc` may exist for different detector designs. The only requirement for the geometry is, however, that it fulfills the navigation aspects as described in Chapter 3. The `ClassicalRecoGeoSvc` hereby allows building traditional tracker designs with cylindrical barrel and disc-like endcap structures. This translation can be extended or later be replaced by other translations, for other detector types, as long as the same interface is implemented and the layer and volume structure kept compatible with the concepts described in Chapter 3.
In the DD4hep geometry, volume and detector trees are parallel to each other (see Fig 2.5). To access both, the geometry and the detector specific information, the converter module has to parse the detector tree. Beginning at the world `DetElement`, it scans through the volumes, the layers, the modules, the components and translates them into the according objects of the reconstruction geometry. For this reason, every detector using this conversion has to have this structure, consisting of volumes containing layers, which hold modules and are made of components (see Fig 4.1). In order to distinguish between the different detector element types while scanning through the

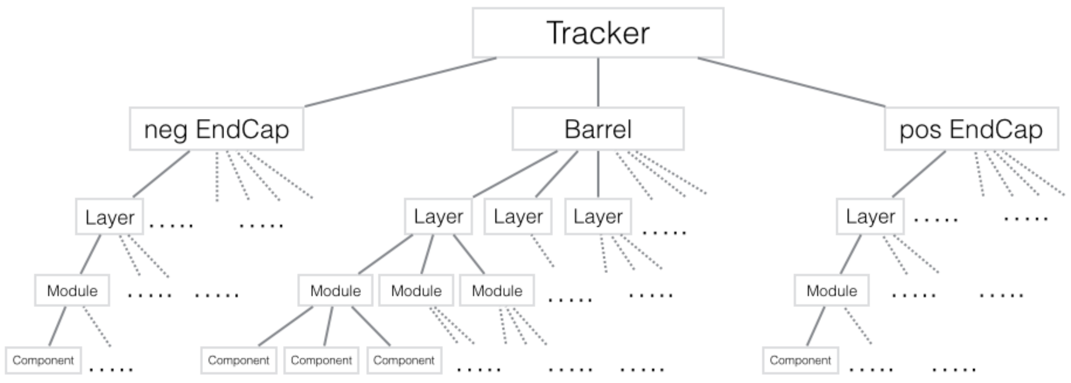
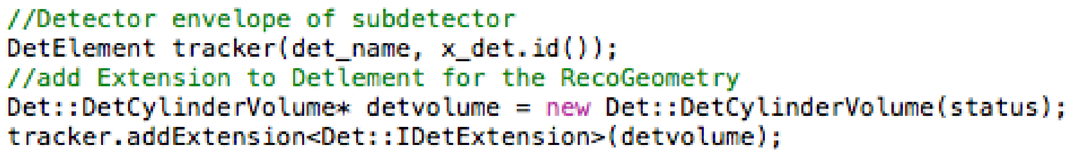


Figure 4.1: To use the `ClassicalRecoGeoSvc` every hierarchy of the tracker has to be composed of a barrel with its corresponding endcaps. These volumes itself are made of layers, modules and components.

volumes the `DD4hep::DetElement` was extended by volume, layer and module classes. All these classes have a common interface `IDetExtension`. The DD4hep geometry is based on ROOT TGeo classes. There is no possible distinction between a volume used as a barrel or a volume used as an endcap (both are implemented as `TGeoConeSeg`). Therefore, two different extension types for these kind of volumes have been implemented (`DetCylinderVolume` and `DetDiscVolume`). For the same reason the extension classes `DetCylinderLayer` and `DetDiscLayer` were introduced. For a module, the `DetModuleExtension` should be used and finally for a sensitive component of the detector the class `DetSensComponent` was created. The `DetSensComponent` must be constructed with the DD4hep class `Segmentation`. In this way the readout segmentation is provided to the reconstruction geometry. If there is no matching class, the general `DetExtension` must be used.
These extensions have to be applied to the `DetElement` during construction time of the DD4hep geometry. In the constructors of the various sub detectors of the detector one has to call the template function `addExtension()` (see Fig 4.2). In the

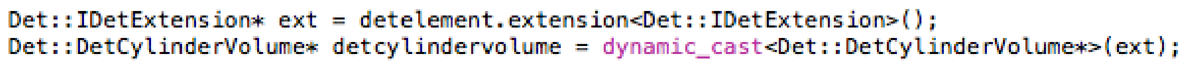


```
//Detector envelope of subdetector
DetElement tracker(det_name, x_det.id());
//add Extension to Detlement for the RecoGeometry
Det::DetCylinderVolume* detvolume = new Det::DetCylinderVolume(status);
tracker.addExtension<Det::IDetExtension>(detvolume);
```

Figure 4.2: Code snippet, showing how to add an extension to the `DetElement` at construction time.

`ClassicalRecoGeoSvc` the extensions are accessed via their interface and can then be casted into the various types (Fig 4.3).

When an algorithm calls `getRecoGeometry()` of the `ClassicalRecoGeoSvc` it first re-



```
Det::IDetExtension* ext = detelement.extension<Det::IDetExtension>();
Det::DetCylinderVolume* detcylindervolume = dynamic_cast<Det::DetCylinderVolume*>(ext);
```

Figure 4.3: Access of the `DetExtension` in the conversion module, the dynamic cast is used for distinguishing the different `DetElement` types.

trieves the DD4hep geometry from the `GeoSvc` and afterwards calls `translateDetector()` with the world `DD4hep::DetElement` as an input parameter. Beginning from this `DetElement` volume, at highest hierarchy level, the service scans the modules. At this stage, the simplification from the full into the reconstruction geometry takes place.
One module (consisting of different components) equals a `Surface` in the reconstruction geometry. Within the translation, an approximation of the detector material is needed in order to describe the material distribution of the components in a two dimensional grid (class `MaterialMap`) on every `Surface`. In every bin of the material map the material parameters are calculated as mean values of the parameters of the components in this bin (see Fig 4.4).:

$$\frac{t}{X_0} = \sum_{i=1}^{n} \frac{t_i}{x_i} \tag{4.1}$$

$$\frac{t}{\Lambda_0} = \sum_{i=1}^{n} \frac{t_i}{\Lambda_i} \tag{4.2}$$

$$\rho = \frac{\sum_{i=1}^{n} t_i \rho_i}{\sum_{i=1}^{n} t_i} \tag{4.3}$$

$$A = \frac{\sum_{i=1}^{n} \rho_i A_i}{\sum_{i=1}^{n} \rho_i} \tag{4.4}$$

$$Z = \frac{\sum_{i=1}^{n} \rho_i Z_i}{\sum_{i=1}^{n} \rho_i} \tag{4.5}$$

$$fsens = \frac{\sum_{i=1}^{n} sensV_i}{\sum_{i=1}^{n} V_i} \tag{4.6}$$

t...thickness, X_0...radiation length, Λ_0...interaction length, ρ...density, A...mass number, Z...atomic number, $fsens$...sensitive fraction, $fsensV_i$...fraction of sensitive volume

The density is averaged with the thickness to obtain the right weighting, since the different components can have different thicknesses. The mass and atomic number are weighted with ρ to take into account that the different atoms occur in different quantity according to their density. The sensitive fraction describes the percentage of the module, which is sensitive. This is useful for fast simulation, when the energy/charge deposit within the sensitive detector component has to be simulated. The module surfaces have to be structurally ordered on a layer in order to allow navigation towards them. For

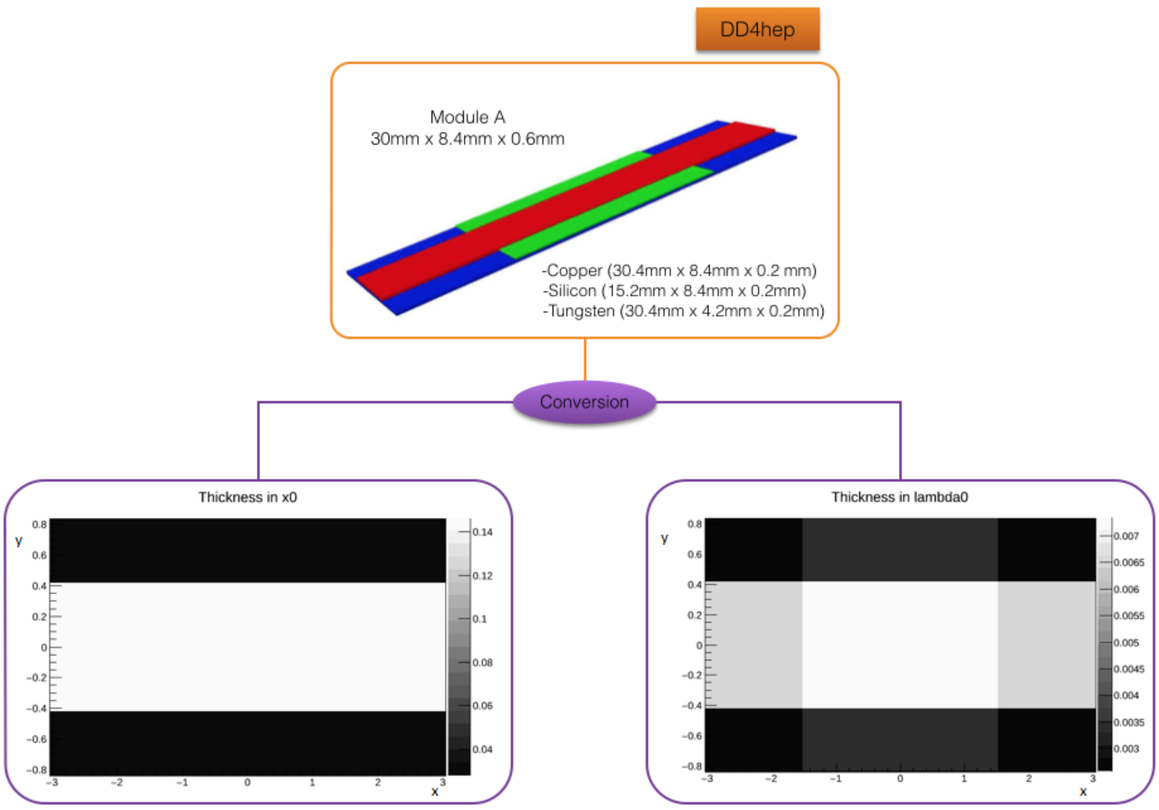

DD4hep
Module A
30mm x 8.4mm x 0.6mm
-Copper (30.4mm x 8.4mm x 0.2 mm)
-Silicon (15.2mm x 8.4mm x 0.2mm)
-Tungsten (30.4mm x 4.2mm x 0.2mm)
Conversion
Thickness in x0
Thickness in lambda0
x
y

Figure 4.4: Example for the material approximation of a module consisting of Tungsten, Silicon and Copper, on a 2D grid on a `Surface` of the reconstruction geometry.

this reason they are sorted and binned in r and φ for `DiscLayers` and in φ and z for `Cylinderlayers`. It is possible to put more than one surface at one bin – layered in z for disc layers and layered in r for cylinder layers. A `BinnedArray` of a vector of surfaces is handed over to the belonging layer. After the creation of the layers, the functions `binCylinderLayers()` and `binDiscLayers()` order the layers, and generate navigation layers (class `NavigationLayer`) in every empty space of the volume. Once the full layer set of a volume is built, the pointers to the previous and next layers are set. In this way, navigation is enabled between the layers within a volume. A distinction between cylinder and disc layers is done, as the layers in the barrel are binned in the r and the layers in the endcaps are binned in z.
The ordered layers, containing the navigation layers are then handed over to the belonging volume. The aim is to get a tree structure held by the world volume and a navigation through the entire tracker. A relation between the volumes has to be created. Therefore the volumes are grouped together in `ContainerVolumes`. Beginning from the lowest hierarchy, the volumes are grouped alternating in r and in z and the pointers to the next/previous volumes are set during this phase (see Fig 4.5). Finally, the pointers to

the beam tube are set. For enabling this translation a few assumptions had to be met. For example one convention for this translation type is, that every sub-volume consists of a barrel and two endcap volumes. All volumes must be attached to each other with no empty space in-between and the barrel of the next hierarchies surrounds the whole sub-detector of the last. The list of conventions to use the current implementation of

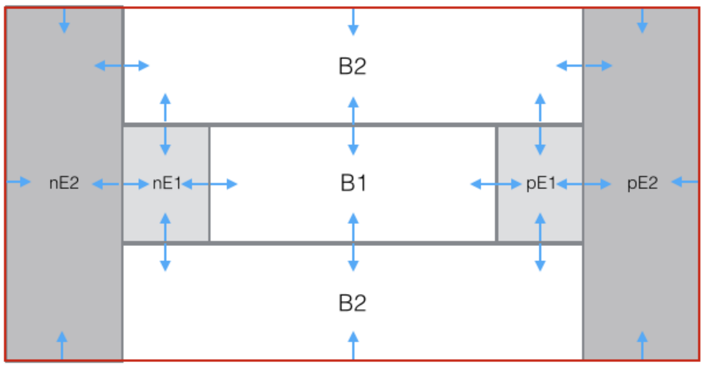


Figure 4.5: The embedding of volumes, enabling navigation between the volumes by setting the pointers of the boundary surfaces to the next/previous volumes.

the `ClassicalRecoGeoSvc` is given below (at the moment this translation was made for a simple prototype and will be extended for more complex detector types):

- every hierarchy of a sub-detector is composed of one barrel and two endcaps
- the detector has to be symmetric around the origin (barrel is placed at zero, one endcap at positive, the other at the negative side)
- all volumes have to be exactly attached to each other, with no empty space in-between (whereas the volumes can be empty)
- one must also create a beam pipe
- every endcap volume is attached to the beam pipe
- every detector has to have the tree like structure, composed of volumes, layers, modules and components, shown in Fig 4.1
- volumes have to surround the layers, the layers have to surround the modules and the modules have to surround their components
- In the detector constructor every `DetElement` has to be extended by its proper extension (see Fig 4.2)

Currently only box and trapezoid shapes for modules and components and cylinders for volumes are implemented in the reconstruction geometry and the translation mechanism. The translation also offers the possibility to place as many hierarchies, layers, modules and components as wanted. The components can be rotated against each other and it is also possible to stagger modules over each other in one layer.

4.2 A Test Tracker

To test the first setup of the framework and the translation between the different geometries a test tracker was build. Since the development of the FCC detector designs are still in progress, a general tracker design, has been chosen. This model can later be replaced by a specific tracker design.
The current structure of the first tracker model is built from a barrel and two endcaps.

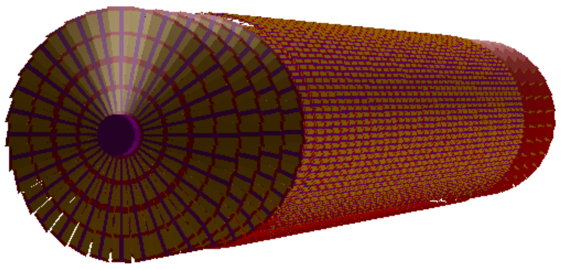


Figure 4.6: A first simple test tracker composed of two hierarchies of barrel with its corresponding endcaps an a beam tube. The detector modules are composed of three components: Silicon (red), Carbon(yellow) and Aluminium (violet).

This structure is embedded into another barrel-endcap setup. An arbitrary number of barrels with endcaps can be nested into each other, with the implemented translation mechanism.
Each barrel is centered at the origin, hence, the endcaps are placed accordingly on the negative/positive side of the barrel. A beam tube, placed innermost inside all volumes, is also part of the model.
Five cylindrical layers are placed concentrically inside each barrel. The endcaps consist of four disc layers, placed along the z-axis. The number of layers, inside the volumes is arbitrary.
The layers are filled with modules, which are composed of an arbitrary number of components. Currently, the modules in the cylindrical layers are chosen to be simple box volumes, composed of three box-shaped components and the modules in the disc layers are trapezoid volumes, with trapezoidal components.

4.3 Geometry Build & Validation

In order to check the geometrical structure, a simple test algorithm was written. Beginning from the center of the detector, straight line trajectories in an arbitrary direction are calculated through the detector, using the internal navigation of the reconstruction geometry to intersect layers and sensitive surfaces, using the guiding technology of the boundary surfaces to move from one volume to the next. Whenever a geometrical node of the reconstruction geometry is intersected, the material is retrieved and recorded. Since this resemble to tracking the particle through the detector of the full simulation by Geant4, a comparison of the geometry used for full and fast simulation can be done. At first, a comparison of the traversed path through the detector components can be seen in Fig 4.7.

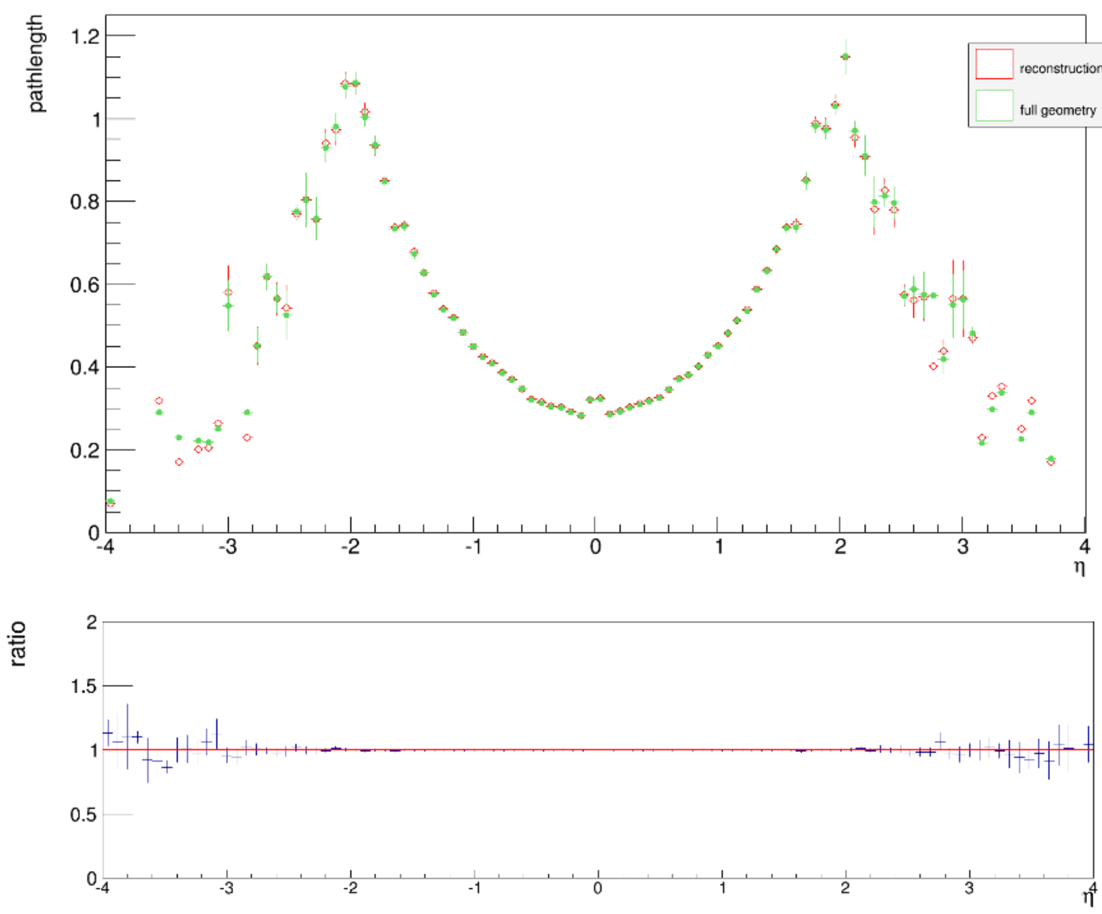


Figure 4.7: Comparison of the path length through the detector as seen by the full and the fast simulation.

In Fig 4.8 the comparison of the DD4hep geometry, displayed with ROOT, with the geometries retrieved for the fast simulation (reconstruction geometry) and the Geant4 geometry is shown. These are hit plots as well. Everytime a module was hit the position was written out in a file. Afterwards this points where displayed with ROOT and compared with the DD4hep geometry.

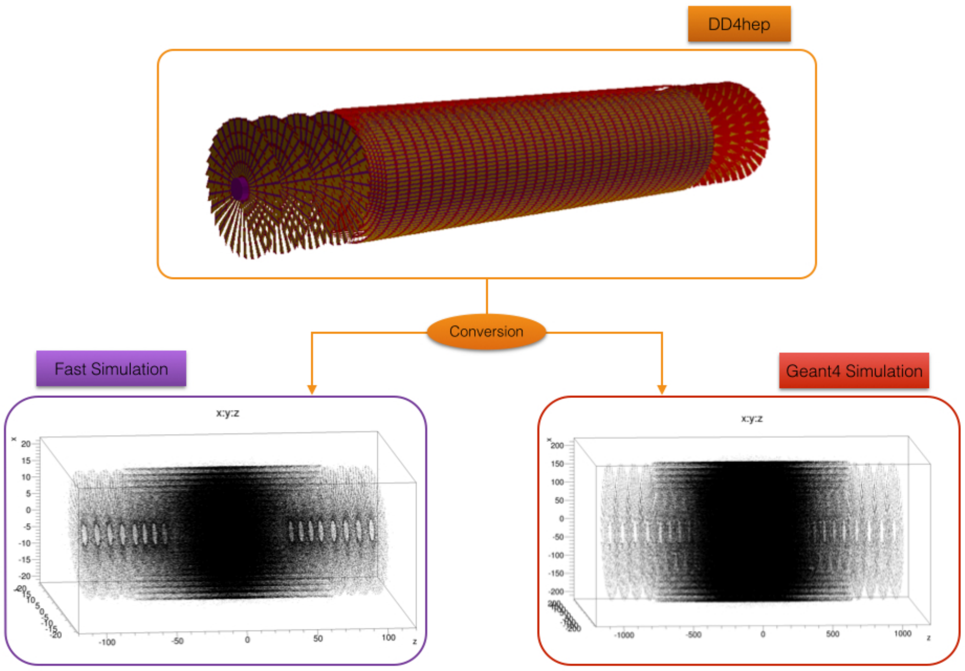


Figure 4.8: Comparison of the DD4hep geometry, with hit plots of the geometries used for fast and full simulation.

Figure 4.9 shows the structure of the detector in the reconstruction geometry, by printing out the position whenever a module, a layer or a boundary surface was hit.

4.3.1 Comparison of Material Budget

A further test compares material seen in the full simulation and in the fast simulation. A coherent description of the material in the reconstruction geometry is not only essential for controlling the occuring disturbance/noise terms in reconstruction, but is inevitable for using the reconstruction geometry as the basis of fast simulation. A Gaudi algorithm retrieves the DD4hep geometry and the Geant4 geometry over the `GeoSvc` and invokes detector building in the reconstruction geometry with the `ClassicalRecoGeoSvc`. Afterwards it invokes a small Geant4 simulation and a simulation through the reconstruction geometry. Both simulations are stepping through their detector and writing out the

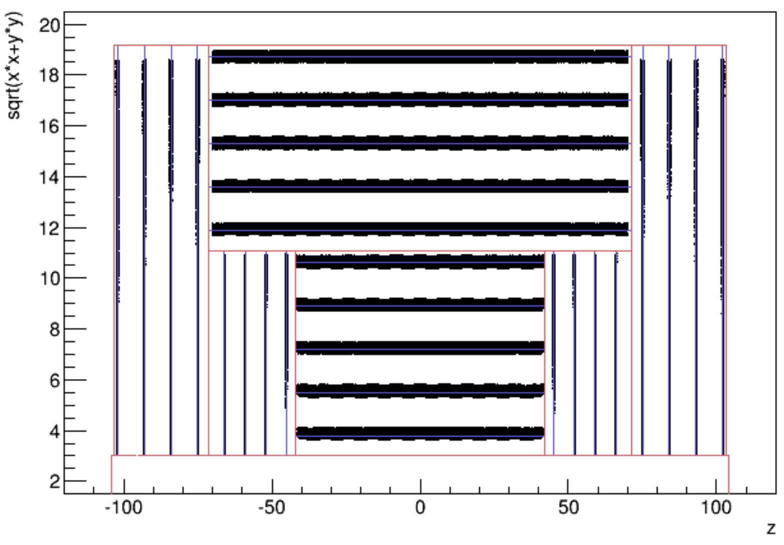


Figure 4.9: Hit plot of the reconstruction geometry, done by straight line intersection with the modules (black), the layers (blue) and the boundary surfaces (pink), displayed in r over z.

thickness in units of the radiation length over η. The results are displayed in ROOT and compared (see Fig 4.10).

The slight discrepancy between the thickness in terms of radiation length passed in the full simulation and passed in the fast simulation occurs due to the use of different internal material parameter definitions. The fast simulation is build from the TGeo geometry of ROOT and uses the material parameters defined in ROOT, while the full simulation uses the Geant4 geometry and the definitions defined in the Geant4 environment. The difference occurs beginning from the third decimal place which allows a rather good agreement for low eta, however the more modules are passed the greater the error gets and therefore grows with η. A possibility to avoid this effect would be to turn off the automatic search of material given by name in the geometry conversion from DD4hep to Geant4 (`Geant4Converter`) and to create completely new materials with the given ROOT parameters.

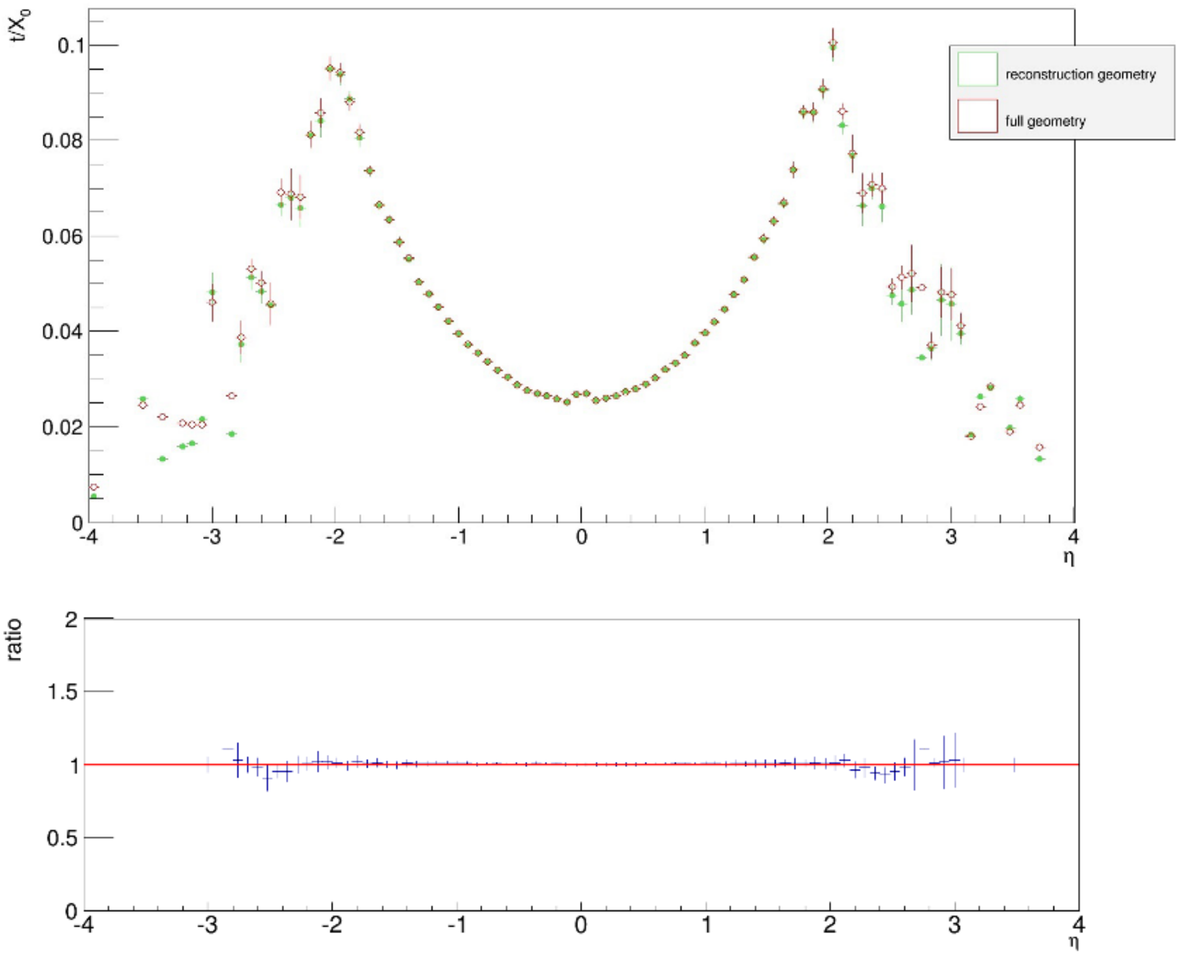


Figure 4.10: Comparison of the thickness in terms of radiation length seen from the full and the fast simulation. The slight dicrepancy between the thickness in terms of radiation length passed in the full simulation and passed in the fast simulation occurs due to the use of different internal material parameter definitions in the different simulation environments.

Chapter 5

Towards a Fast Track Simulation

As outlined in Section 1.2, full simulation is often too CPU and work intensive for testing reasons or if large statistics samples are needed for certain studies. Therefore fast simulation approaches are essential. One possibility to speed up the simulation, is to simplify the geometry.
The reconstruction geometry uses a simplified geometry, describing the detector modules as infinitesimal thin surfaces and the material approximately on this surface. The reconstruction geometry is directly translated from the full DD4hep geometry and is thus changed accordingly to the geometry used for full simulation. A navigation to follow the particle through the detector and enable material effects in the reconstruction has been implemented. Given these advantages, it is only natural to implement a fast simulation based on the reconstruction geometry. Furthermore, this fast simulation can easily create data formats that are directly usable for the reconstruction, making an easy integration in the simulation chain (see Fig 1.2) possible.
In this Chapter the basic concepts of the fast simulation based on the reconstruction geometry are explained and first results are shown.

5.1 Navigation of Geometry

To run a simulation, a navigation to emulate particles traversing through the detector is essential. The purpose of the navigation is to create relations between the different volumes, layers and surfaces so that first a continuous movement through the detector is possible. Second, every geometrical object can be accessed at every point allowing the simulation to start at any given point in the detector.
The navigation through the detector is enabled by two basic concepts:

1. Volumes are bordered by `BoundarySurface`s (see Section 3.5). These boundary surfaces point to their next and previous volumes.

2. Volumes are fully packed with layers – either filled with detector surfaces or empty, only implemented for navigation reasons. These layers point to their next and previous layer and enable navigation within one volume.

To access a surface at every given point in the detector, the surfaces are ordered with a `BinnedArray` (Section 3.2) in a layer. The `BinnedArray` makes a grid on the layer and stores the surfaces according to their position, to access the correct surfaces at every point on the layer. With the same principle the layers are stored in the volumes and the volumes itself are stored in container volumes. In the end one can access the whole geometry through one top `ContainerVolume`. The container volumes, the volumes and the layers have methods implemented to access their stored elements at every point.

5.2 Fast Simulation Principle

Using the navigation between volumes, layers and detector surfaces, the simulation mimics the straight particle trajectory through the detector and the intersections with the sensitive material can be used as hit creation.
After the world detector volume is accessed over the `IRecoGeoSvc` interface, the volume at a specific start point (point of interaction) can be accessed. Afterwards all layers of this volume, which contain detector modules can be accessed beginning from a certain point in a given direction. By intersecting in a straight line with the layers, the point where the layer will be hit can be extracted. This is possible due to the fact, that the `Layer` classes are implemented as `Surface`s as well. This point is then used to retrieve the surfaces which can possible be hit by this track – the surface at this specific point, plus the surrounding surfaces. Once this surfaces are retrieved, a straight line intersection can be applied and hits can be created.
If the end of the current volume is reached, the next volume is reached by making straight line intersections with the boundary surfaces of the volume and asking the retrieved `BoundarySurface` for the next `Volume`. In Fig 5.1 the number of Hits created by the fast simulation, compared with the full simulation can be seen. When the straight line intersection is, at later stage, replaced with a proper numerical field integration and material interactions are simulated according to the traversal amount of material, a rather realistic hit simulation is obtained.

5.3 Monte Carlo Based Material Effects

When a particle traverses through matter it interacts with the atoms of the material. Relying on the material, the interaction causes deflection of the particle, energy loss or, hadronic processes can lead to the destruction of the particle if the interaction is nuclear. Depending on the particles mass, charge and energy it interacts with the electron shell and can interact with the atomic nuclei. This can lead to excitations, ionization of the material, secondary particle production and radiation losses. The particle loses energy caused by these processes and will be scattered according to the atomic structure of the material [11].

5.3.1 Multiple Scattering

When the particle passes through the material it will be scattered many times by just small angles, depending on the energy of the particle, it can sum up to a rather big displacement. The main contribution is from coulomb scattering between the particle and the nuclei of the material.
In approximation for very thin surfaces and for a first simple test, only the multiple scattering was taken into account of all material interactions. For a test setup the function `materialInteraction()` was applied to the `Surface` class. This function changes the direction according to the scattering angle of the particle. The Code was taken from ATLAS and adapted. The underlying process is described by the Highland scattering formula [26].
The projected angular distribution is described by:

$$\theta_0 = \frac{13.6MeV}{\beta c p} z \sqrt{\frac{t}{X_0}}[1 + 0.038\ln(\frac{t}{X_0})] \tag{5.1}$$

where $\beta c, p$ and z describe the velocity, the momentum and the charge of the incident particle and $\frac{t}{X_0}$ is the thickness in terms of radiation length.
One can compare the final location of a deflected particle on a `Surface`, with a not deflected particle. The resulting deviations of a statistic sample is described by a gaussian distribution. In Fig 5.2 some of these are plotted in one graph for different transverse momenta. The higher the initial energies, the less the particle is deflected. The corresponding standard deviations are plotted in Fig 5.3.

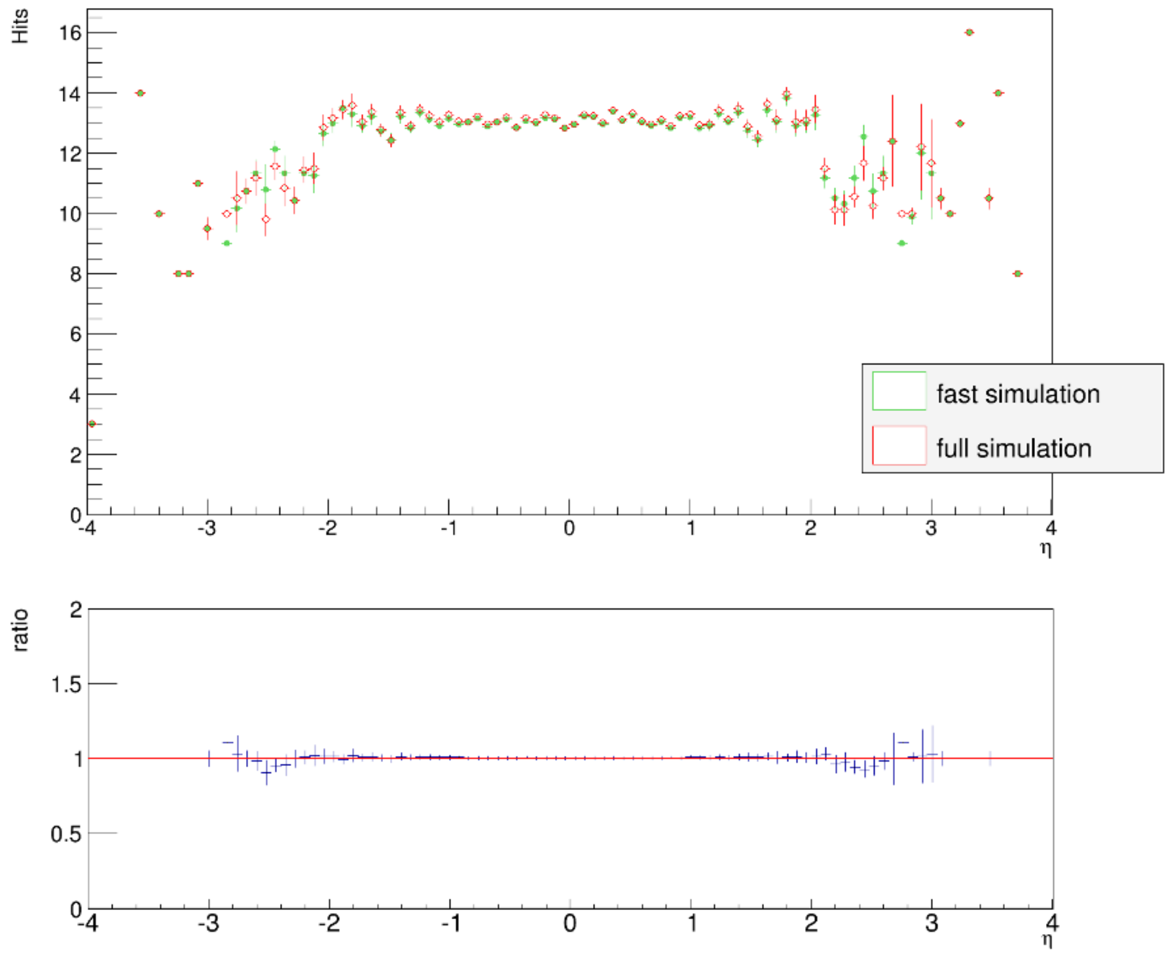


Figure 5.1: Comparison of the number of hits seen from the full and the fast simulation. Full simulation is done using Geant4, while fast simulation is based on the reconstruction geometry and uses the embedded navigation to intersect the sensitive detector modules.

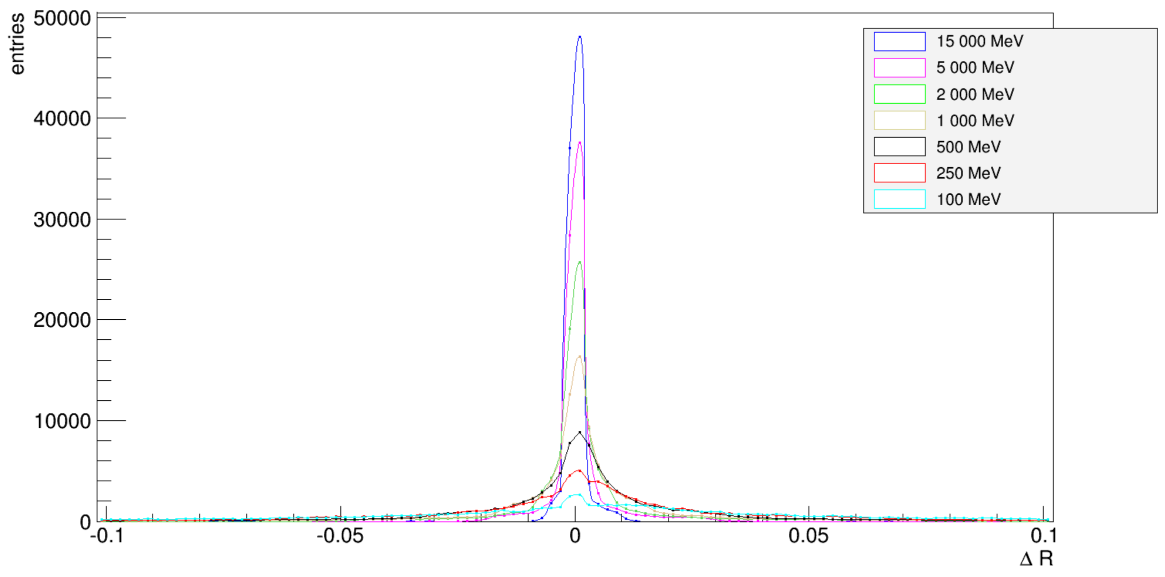


Figure 5.2: Residual distributions of 100 000 events for muons with different energies.

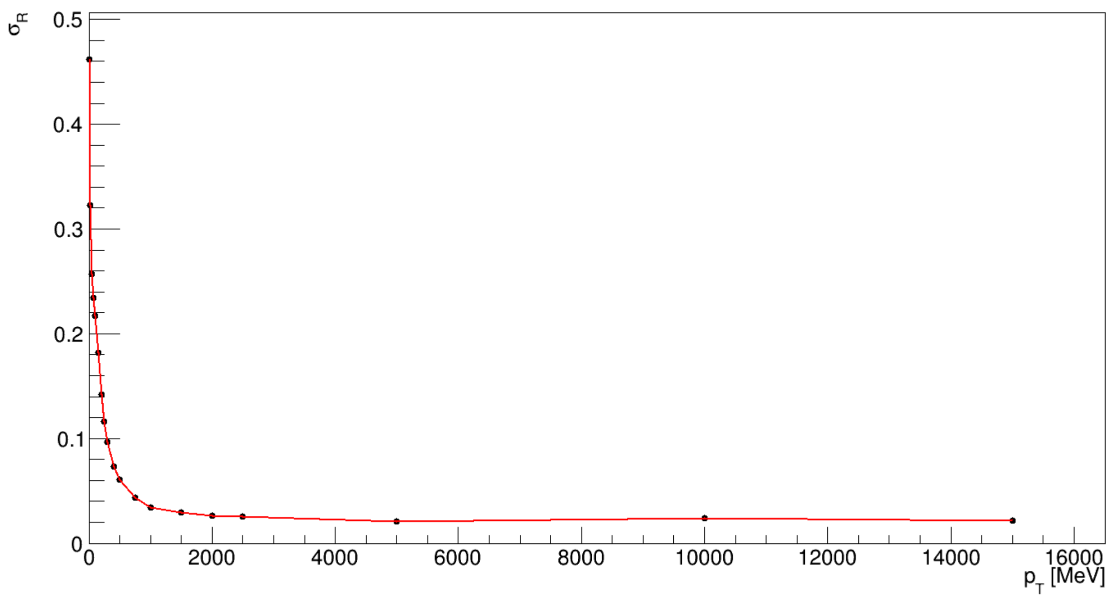


Figure 5.3: Standard deviations of the residual distributions of 100 000 events for muons with different energies.

Chapter 6

Summary and Outlook

A coherent framework which allows to run full, fast and parametric simulation from a single source of detector description and one configuration setup was established and successfully tested. Gaudi is used as event processing framework and the framework can be steered for all different simulations from a job configuration file. DD4hep is used for the geometry description and is provided in the different geometries needed by the simulations and the reconstruction. A simplified geometry for reconstruction and fast simulation purposes was implemented. The approximation of the material and the translation into this geometry from the full DD4hep geometry is done by an automated transcript. The reconstruction geometry uses the same readout segmentation as the DD4hep geometry to guarantee consistency.
Currently a more flexible translation from the DD4hep geometry into the reconstruction geometry is in development. It allows to adapt support tubes, which give structural support to the tracker and are needed to build a realistic tracker. Furthermore the tracker can be created more flexible – not every hierarchy needs to have the same structure. A new feature in DD4hep which allows to assign detector elements together to a sub hierarchy (so-called *nested detectors*), allow this more flexible realization. Along the way a more realistic tracker design will be introduced, following the first suggestions for the dimensions of a FCC tracker.
The next steps for the fast simulations will be to apply a magnetic field. This will be done by the creation of a magnetic field service, which returns the magnetic field at the requested position. The magnetic field map will currently be read in from a file. In this way charged particles will form curved tracks and it is possible to calculate their momentum. A transportation tool will also be applied from the LHC experiments, taking multiple scattering, energy loss and magnetic field into account.
An adaption of the reconstruction algorithms for the use in the FCC software suite is foreseen as a further step to find particle tracks and in the end identify the particle.
A further future prospect would be an expansion of the fast simulation for a use in calorimeters.

Bibliography

[1] LHC webpage. http://home.web.cern.ch/topics/large-hadron-collider.

[2] ATLAS webpage. http://atlas.ch.

[3] CMS webpage. http://cms.web.cern.ch.

[4] LHCb webpage. http://lhcb.web.cern.ch/lhcb.

[5] LEP webpage. http://home.web.cern.ch/about/accelerators/large-electron-positron-collider.

[6] FCC webpage. http://cern.ch/fcc.

[7] F. Zimmermann M. Benedikt. Future Circular Collider (FCC) Study, 2015. https://indico.cern.ch/event/352487/material/slides/1.pdf.

[8] F. Gianotti. FCC-hh Workshop: 26-28 May, Introduction, May 2014. http://indico.cern.ch/event/304759/contribution/0/attachments/577941/795902 /FCChh-WS.pdf.

[9] Torbjörn Sjöstrand. PYTHIA 8 webpage, 2014. http://home.thep.lu.se/ torbjorn/pythia81html/Welcome.html.

[10] HEWRIG webpage, 2014. https://herwig.hepforge.org.

[11] H. Leeb H. Abele. Atom-, Kern- und Teilchenphysik 2. Vorlesungsskript, 2014.

[12] The ATLAS Collaboration. Charged-particle multiplicities in pp interactions measured with the ATLAS detector at the LHC , December 2010. https://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/PAPERS/STDM-2010-06.

[13] The ATLAS Collaboartion. The simulation principle and performance of the ATLAS fast calorimeter simulation FastCaloSim, October 2010. ATL-PHYS-PUB-2010-013.pdf.

[14] W. Lukas, editor. *Fast Simulation for ATLAS: Atlfast-II and ISF*. The ATLAS Collaboration, IOP Science, 2012. ATL-SOFT-PROC-2012-065.

[15] The ATLAS Collaboration. Data m/c comparison for calorimeter shower shapes of high et electrons, October 2011. http://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/EGAMMA/PublicPlots /20111005/ ATL-COM-PHYS-2011-1299/index.html.

[16] E. Ritsch. *ATLAS Detector Simulation in the Integrated Simulation Framework applied to the W Boson Mass Measurement.* PhD thesis.

[17] P. Maley M. Cattaneo. Gaudi Users Guide, 2001. http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/Gaudi/Gaudi$_v$9/*GUG/GUG.pdf*.

[18] DD4hep webpage. http://aidasoft.web.cern.ch/DD4hep.

[19] DD4hep manual. http://svnsrv.desy.de/viewvc/aidasoft/DD4hep/trunk/doc /DD4hepManual.pdf.

[20] DD4hep DDcore. https://svnsrv.desy.de/public/aidasoft/DD4hep/trunk/DDCore.

[21] ROOT webpage. https://root.cern.ch.

[22] Geant4 webpage. https://geant4.web.cern.ch/geant4.

[23] Geant4 User's Guide: For Application Developers. https://geant4.web.cern.ch/geant4.

[24] Twiki page for FCC EDM. https://twiki.cern.ch/twiki/bin/viewauth/FCC /Fcc-SoftwareEDM.

[25] M. Wolter A. Salzburger, S. Todorova, editor. *The ATLAS Tracking Geometry Description*. The ATLAS Collaboration, CERN, 2007. ATL-SOFT-PUB-2007-004.

[26] K.A.Olive, editor. *Passage of particles through matter 1*. PDG - Particle Data Group, 2014. http://pdg.lbl.gov/2014/reviews/rpp2014-rev-passage-particles-matter.pdf.

[27] J. Beringer, editor. *Kinematics*. PDG - Particle Data Group, 2012. http://pdg.lbl.gov/2013/reviews/rpp2012-rev-kinematics.pdf.