Detector Simulations with DD4hep

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 042015 doi:10.1088/1742-6596/898/4/042015

Detector Simulations with DD4hep

M Petrič[1], M Frank[1], F Gaede[2], S Lu[2], N Nikiforou[1,3], A Sailer[1]

[1] CERN, CH-1211 Geneva 23, Switzerland
[2] DESY, Notkestraße 85, 22607 Hamburg, Germany
[3] Now at University of Texas at Austin, Austin, TX 78712, USA

E-mail: marko.petric@cern.ch

Abstract. Detector description is a key component of detector design studies, test beam analyses, and most of particle physics experiments that require the simulation of more and more different detector geometries and event types. This paper describes DD4hep, which is an easy-to-use yet flexible and powerful detector description framework that can be used for detector simulation and also extended to specific needs for a particular working environment. Linear collider detector concepts ILD, SiD and CLICdp as well as detector development collaborations CALICE and FCal have chosen to adopt the DD4hep geometry framework and its DDG4 pathway to Geant4 as its core simulation and reconstruction tools. The DDG4 plugins suite includes a wide variety of input formats, provides access to the Geant4 particle gun or general particles source and allows for handling of Monte Carlo truth information, eg. by linking hits and the primary particle that caused them, which is indispensable for performance and efficiency studies. An extendable array of segmentations and sensitive detectors allows the simulation of a wide variety of detector technologies. This paper shows how DD4hep allows to perform complex Geant4 detector simulations without compiling a single line of additional code by providing a palette of sub-detector components that can be combined and configured via compact XML files. Simulation is controlled either completely via the command line or via simple Python steering files interpreted by a Python executable. It also discusses how additional plugins and extensions can be created to increase the functionality.

1. Introduction

For a long time, high energy physics experiments have been striving to develop software tools for the complete description of detector models based a single source of information. For the success of an experiment it is important to provide a consistent detector description to simulation, reconstruction and analysis applications from a single source. When referring to the detector description this includes, in addition to the geometry and the materials used in the device, also parameters describing e.g. the detection techniques, constants required for alignment and calibration, description of the readout structures and conditions data.

The main motivation behind the DD4hep package is to devise a toolkit that addresses all these issues for all the stages of an experiment [1, 2]. It was designed based on experiences from the LHCb experiment [3], as well as from developments in the Linear Collider community [4]. The tool relies on usage of pre-existing and widely used software, combining it into a consistent generic detector description. The main components are the ROOT geometry package [5], which is used for construction and visualization of geometry, and the Geant4 simulation toolkit [6], which can be interfaced via DD4hep to perform detector simulation in complex detector designs [7].

2. Project Aim

The intent is to provide a comprehensive detector description for the quantification of the detector and provide all the information necessary for interpretation of data form the experiment. From past experiences it can be established that a comprehensive view, not limited only to geometry, is extremely advantageous for the construction of a complete set of tools aimed at data interpretation, e.g. alignment, where detector positions vary over time and need to be matched with the geometry for reconstruction.

The conception of the toolkit was driven mainly by the following requirements:

- **Complete Detector Description**: provides full detector geometry, the materials used when building the structures, visualization attributes, detector readout information, alignment, calibration and environmental parameters.
- **Coverage of the full life cycle of the experiment:** supports all stages from detector concept development, detector optimization, construction, operation and at the same time enables easy transition from one stage to the next.
- **Single source of information:** provides a consistent detector description, for simulation, reconstruction, analysis.
- **Ease of Use:** delivers a simple and intuitive interface, with minimal external dependencies.

3. The DD4hep Toolkit

The core element of DD4hep is the the so-called Generic Detector Description Model (GDDM). Figure 1 depicts the interaction of the main components of DD4hep and their interfaces to the end-user applications, namely the simulation, reconstruction, alignment and visualization. The generic detector description is an in-memory model, that consists of a set of objects containing geometry and auxiliary information about the detector.

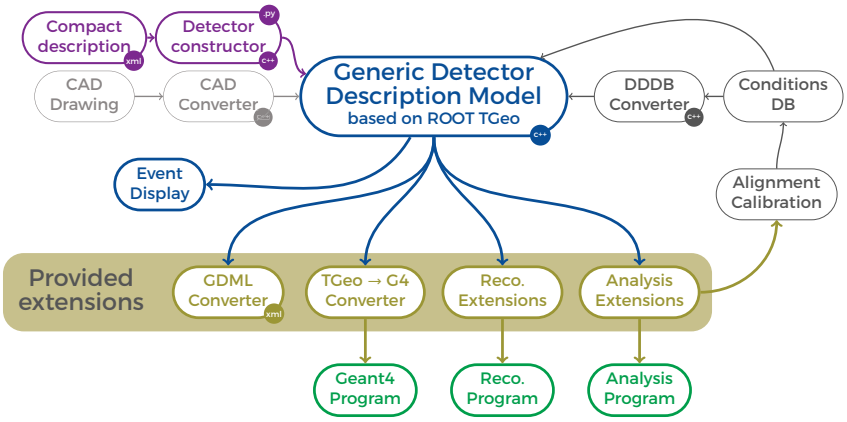

Figure 1. The components of the DD4hep detector geometry toolkit.

It is envisioned that the GDDM can be constructed through several means, but current development is focused on a mechanism that converts a compact detector description in `XML` format through specialized code fragments (called Detector constructors) into the GDDM. These code fragments instantiate the GDDM of the detector defined by a set of `C++` classes. From this model it is possible to transform in memory the GDDM to e.g. Geant4 geometry or a GDML file.

3.1. The Compact Description

The compact detector description is based on concepts of the linear collider detector simulation [8]. The compact description is provided in `XML` format, has a minimalistic form (see example in Figure 2) and was designed with the aim for conceptual design studies. Since the parsers of DD4hep do not validate the schema of the `XML` it facilitates simple addition of new elements and attributes into the detector description. The information in the `XML` is per-se not enough for a comprehensive detector description. To achieve this, specialized code fragments are needed, the so called Detector Constructors, which interpret the `XML`.

```
<detector id="15" name="HCal" type="GenericCalBarrel_o1_v01" readout="HCalCollection">
  <envelope vis="HCALVis">
    <shape type="PolyhedraRegular" numsides="HCal_sym" rmin="HCal_rmin" rmax="HCal_rmax" dz="HCal_dz" material="Air"/>
    <rotation x="0*deg" y="0*deg" z="90*deg-180*deg/HCal_symmetry"/>
  </envelope>
  <dimensions numsides="HCal_sym" rmin="HCal_rmin" z="HCal_dz*2"/>
  <layer repeat="(int) HCal_layers" vis="HCalLayerVis">
    <slice material="Steel235"    thickness="0.5*mm" vis="HCalAbsorberVis" radiator="yes"/>
    <slice material="Steel235"    thickness="19*mm"  vis="HCalAbsorberVis" radiator="yes"/>
    <slice material="Polystyrene" thickness="3*mm"   sensitive="yes"  limits="cal_limits"/>
    <slice material="Copper"      thickness="0.1*mm" vis="HCalCopperVis"/>
    <slice material="PCB"         thickness="0.7*mm" vis="HCalPCBVis"/>
    <slice material="Steel235"    thickness="0.5*mm" vis="HCalAbsorberVis" radiator="yes"/>
    <slice material="Air"         thickness="2.7*mm" vis="InvisibleNoDaughters"/>
  </layer>
</detector>
```

Figure 2. An example compact description for a calorimeter.

3.2. Detector Constructors

Detector Constructors are short code pieces that contextualize the `XML` input and create a detector instance. Each code segment constructs a single sub-detector at a time, but the toolkit enables to combine several detector instances together and thus constructs a complex and large apparatus. Detector construction can be written either via `C++` (example snippet shown in Figure 3) or Python. Whereas the `C++` variant enables better detection of errors at compilation time, the Python interfaces are easier to use, but errors are usually only detected at execution time. With the combination of compact description and the detector constructors one is able to simulate and visualize (Figure 4) the detector.

```
static Ref_t create_detector(LCDD& lcdd,
       xml_h e, SensitiveDetector sens) {

  xml_det_t x_det = e;
  Layering layering(x_det);
  xml_comp_t staves = x_det.staves();
  xml_dim_t dim = x_det.dimensions();
  DetElement sdet(det_name, x_det.id());
  Volume motherVol = lcdd.pickMotherVolume(sdet);

  PolyhedraRegular polyhedra(numSides, rmin, rmax, detZ);
  Volume envelopeVol(det_name, polyhedra, air);

  for (xml_coll_t c(x_det, _U(layer)); c; ++c) {
    xml_comp_t x_layer = c;
    int n_repeat = x_layer.repeat();
    const Layer* lay = layering.layer(layer_num - 1);
    for (int j = 0; j < n_repeat; j++) {
      string layer_name = _toString(layer_num, "layer%d");
      double layer_thickness = lay->thickness();
      DetElement layer(stave, layer_name, layer_num);
```

Figure 3. Code snippet to create a calorimeter.

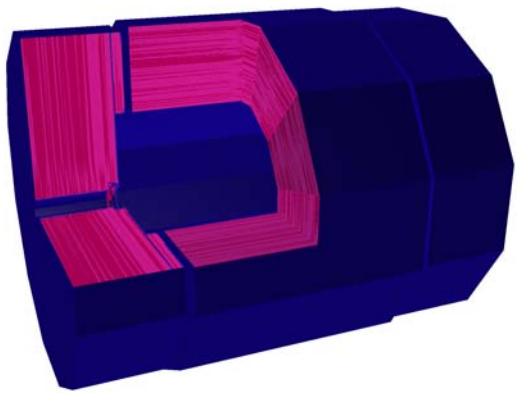

Figure 4. Visualization based on the compact description and a detector constructor.

3.3. Generic Detector Description Model

The aim of the GDDM is to build an in-memory model of the detector that can provide geometrical as well as functional facets. It is based on the TGeo geometry package from ROOT [5] with certain extensions. The TGeo geometry classes are used directly without isolation interfaces, with the exception of detector constructors, where users can write easy, compact and readable code for custom detector constructors. The in-memory object is structured as a tree of `DetectorElement` objects (Figure 5). These elements provide an easy entry point to any given detector part of the apparatus and represent a complete sub-detector (e.g. tracker or calorimeter), a part of a sub-detector (e.g. tracker-disk or tracker-barrel), a module or any other arbitrary sub-detector part.

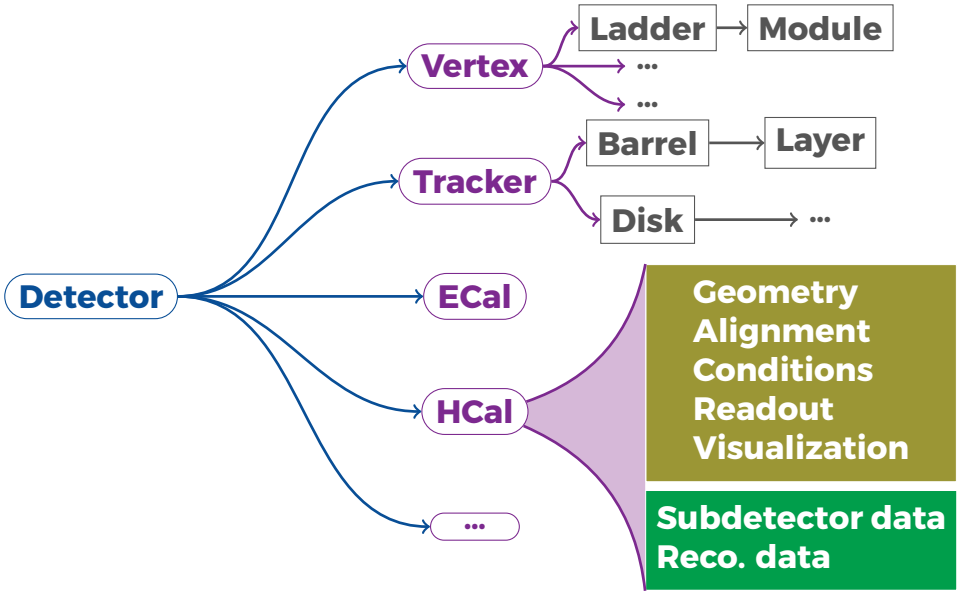

Figure 5. The tree structure of `DetectorElement` objects that constitute the GDDM.

The main purpose is to provide an easy access to a desired sub-detector information. If a tracking program wants access to the tracker `DetectorElement`, DD4hep will allow the program access to all the geometry properties, alignment and calibration constants and other slow varying conditions such as temperatures. The `DetectorElement` acts as a data container. DD4hep provides a singleton object called `LCDD`, through which access to the whole geometry is possible, and which provides at the same time ownership, bookkeeping and management properties to the instances of the detector model.

As shown in Figure 5, an application might require special functionality. This can be achieved with specialized classes that extend the `DetectorElement`. Such extensions are usually needed in addressing issues connected to reconstruction algorithms, such as pattern recognition, tracking, vertexing and particle identification. One example could be surfaces that aid in track fitting. A generic detector description toolkit cannot address all cases, there are far too many of them, but it can provide a flexible extension mechanism via code fragments to develop the necessary functionality which is needed for a given case. Depending on the case, the extensions can store additional specialized data, expose additional behavior or both. The user can easily add additional behavior by overloading the `DetectorElement` class and extending its internal data. Since the internal data is public and addressed by reference, an arbitrary number of extensions can be created with minimal overhead. In addition to this, more data can be incorporated at any

time by the user into this scheme by using a simple aggregation mechanism. The only constraint is that these data extensions have to be of different types. It is thus possible to optimize the attachment depending on the particular use case. DD4hep utilities this functionality in the reconstructions extension called DDRec. The content and its application to the Linear Collider case is discussed elsewhere [9].

4. Simulation support – DDG4

Since the DD4hep geometry is based on ROOT/TGeo, it has to be translated to the Geant4 specific geometry representation to enable simulation. Using the Geant4 user interface `G4VUserDetectorConstruction`, a generic mechanism has been implemented that converts on the fly all geometry entities like shapes, materials, volumes and volume placements to a Geant4 geometry. The mechanism is implemented to traverse the geometry tree from the top level element and recursively collect all entities. A special treatment was needed for certain objects, where the implementation paradigm differ between the Geant4 and the ROOT geometry, e.g. shape assemblies.

To simplify the usage and implementation of plugin functionality, all components use a common base class. The base class is designed to provide simple access to all objects needed to perform simulation tasks. It is implemented to handle all common functionalities, e.g. external configuration of component properties, data access via Geant4 or the detector description toolkit. Since all components inherit from the `Geant4Action`, a sequence of actions can be created e.g. to merge and simulate two different data types. The typical action sequence is:

Initialization step: properties related to the event and its context are prepared.

Acces to input data type 1: reading or generation of events of type 1 (e.g. simulated e^+e- events). The primary vertices from the input data and resulting tracks are then boosted and the primary vertices are smeared in subsequent modules according to the assumed beam profiles and beam conditions.

Acces to input data type 2: reading or generation of events of type 2 (e.g. beam induced background events). The primary vertices from the input data are treated in the same manner as in the previous step.

Merging: merge to one single record combining the primary vertices and outgoing primary tracks of both types of data (with possibility to add an arbitrary number of input data).

Geant4: the entire record containing all primary vertices and all primary tracks is given to Geant4, and the simulation can start.

4.1. Monte Carlo Truth Handling

A basic functionality that DDG4 offers, is the assignment of generated energy deposits to the tracks which created the deposit. There are several ways to address this issue, e.g:

Complete history record: the entire history is stored containing all energy deposits and the full record of created tracks. The drawback of this is that the record becomes quite big due to a large amount of secondary particles and the information associated to them. Such a record is usually only used for detailed studies and not for mass production.

Reduced history record: Only relevant energy deposits and tracks are kept. Low energetic tracks are reassigned to the parent tracks, whereas the energy deposits in calorimeters are handled differently.

DDG4 can handle the propagation of Monte Carlo truth information between hits and tracks for both cases. It utilizes a truth handler which uses an internal track object model, that can be tuned by the user at the output stage to the desired user track model. This gives the user a high

degree of flexibility to construct an arbitrary output record based on the complete simulation information. This reduces the user effort needed to evaluate performance and efficiency of detector models.

4.2. Components and handling

DD4hep/DDG4 provides a ready-to-use pallet of components, which include standard input and output format support such as LCIO [10], StdHEP [11] or HepMC [12] modules to manipulate primary interactions, e.g. smearing or boosting, and also generic components collecting the detector response of tracking detectors or calorimeters. All these features enable users interested in fast detector prototyping to perform all necessary simulation activities of detector designs. DDG4 comes with a default palette of modules that are generic enough to be applicable to a wide variety of designs. The users can also provide specific detailed modules via the plugin mechanism.

Since the Geant4 physics setup is contained in a set of factories, the subsequent instantiations to particle constructors, physics processes, physics constructors or predefined physics lists are implemented in DDG4 by means of factory bindings. The configuration of a DDG4 application is possible via the following ways: data in a `XML` file, a `C++` or a Python script. Due to the fact that `C++` is the native language of the framework, one can configure the simulation via the same language, however a change of configuration requires a recompilation of the software which might be cumbersome. The most flexible way to configure DDG4 is via Python. The ROOT interface PyROOT is the most flexible and still robust way to bootstrap a DDG4 simulation application. The Python interfaces are provided automatically at the time of the generation of ROOT dictionaries. The dictionary mechanism exports the same classes to Python that reside in `C++`, which leads to a fairly similar configuration. Furthermore this approach does not need any recompilation. An example showing how easy it is to configure the input file in Python can be seen in Figure 6, and the usage of a filter in Figure 7.

```
#...
gen = DDG4.GeneratorAction( kernel , "LCIOInputAction/LCIO1" )
gen.Input = "LCIOFileReader|" + inputFile
#...
```

Figure 6. Example showing how to configure a DDG4 input file in Python.

```
#...
part = DDG4.GeneratorAction(kernel, "Geant4ParticleHandler/ParticleHandler")
kernel.generatorAction().adopt(part)
part.SaveProcesses = ['Decay']
part.MinimalKineticEnergy = 1*MeV
part.KeepAllParticles = False
#...
user = DDG4.GeneratorAction(kernel,"Geant4TCUserParticleHandler/UserParticleHandler")
user.TrackingVolume_Zmax = DDG4.tracker_region_zmax
user.TrackingVolume_Rmax = DDG4.tracker_region_rmax
#...
```

Figure 7. Example for configuring actions, filters, sequences and cuts in Python.

5. Use cases

The DD4hep toolkit has been adopted by several of conceptual design studies for future high-energy colliders, including the CLICdp, ILD, SiD and FCC collaborations. The visualization of each detector concept that is actively developing simulation and reconstruction software based on DD4hep can be seen in Figure 8.

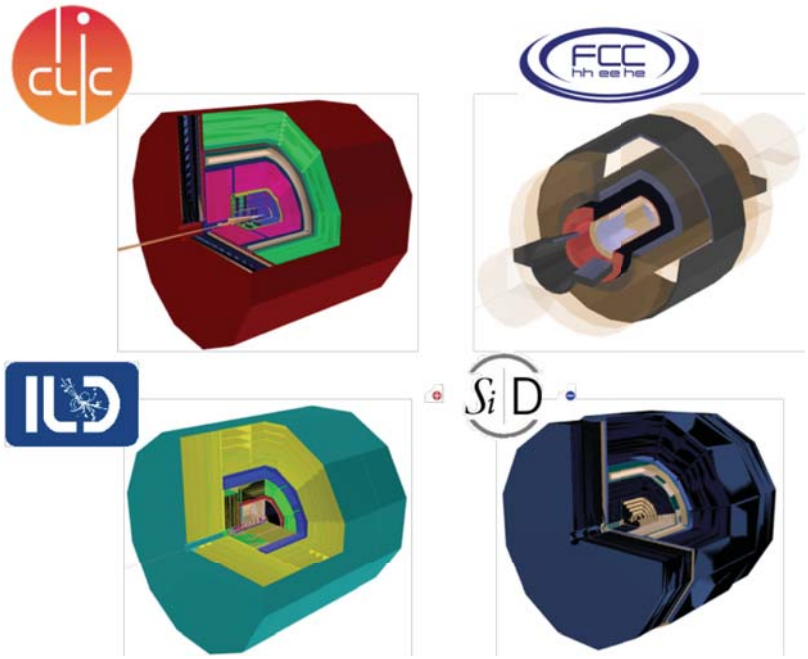

Figure 8. Visualization of detector models of known DD4hep user collaborations.

5.1. Linear Collider Simulation Extension

Furthermore, the Linear Collider community has developed an extension to DDG4 called DDSim. It adds an additional layer of abstraction and simplifies simulation, allowing even command line configuration of the simulation. An Python example of DDSim steering can be seen in Figure 9.

```
from DDSim.DD4hepSimulation import DD4hepSimulation
from SystemOfUnits import mm, GeV, MeV, keV
SIM = DD4hepSimulation()
SIM.compactFile = "CLIC_o3_v06.xml"
SIM.runType = "batch"
SIM.numberOfEvents = 2
SIM.inputFile = "electrons.HEPEvt"
SIM.part.minimalKineticEnergy = 1*MeV
SIM.filter.filters ['edep3kev'] =
dict (name="EnergyDepositMinimumCut/3keV" ,
      parameter={"Cut" : 3.0*keV} )
```

Figure 9. Example usage of the DDSim Python application.

6. Conclusion

The DD4hep Detector Description Toolkit provides an extended toolset for detector description, simulation and reconstruction. The user community of DD4hep is growing. In addition to the Linear Collider community as first users, it was also adopted for the FCC detector studies. The toolkit is under investigation for future software implementation in LHCb. The software is in stable condition and the Linear Collider community is planning to commence mass Monte Carlo production and reconstruction based on DD4hep in 2017. Topics such as alignment and the connection of condition data to detector elements are addressed in the design and are in the process of being implemented. These will be a subject of future work.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

References

[1] Frank M *et al.* 2017 Aidasoft/dd4hep: v00-20-pre01 URL `https://doi.org/10.5281/zenodo.242377`
[2] Frank M, Gaede F, Grefe C and Mato P 2014 *J. Phys. Conf. Ser.* **513** 022010
[3] Ponce S, Mato Vila P, Valassi A and Belyaev I 2003 *eConf* **C0303241** THJT007 (*Preprint* `physics/0306089`)
[4] LCC The Linear Collider Collaboration `https://www.linearcollider.org/`
[5] Brun R, Gheata A and Gheata M 2003 *Nucl. Instrum. Meth.* **A502** 676–680
[6] Agostinelli S *et al.* (GEANT4) 2003 *Nucl. Instrum. Meth.* **A506** 250–303
[7] Frank M, Gaede F, Nikiforou N, Petric M and Sailer A 2015 *J. Phys. Conf. Ser.* **664** 072017
[8] Gaede F, Graf N and Johnson T 2007 (International Conference on Computing in High Energy and Nuclear Physics, Victoria (Canada), 2 Sep 2007 - 7 Sep 2007) URL `http://bib-pubdb1.desy.de/record/81331`
[9] Sailer A, Frank M, Gaede F, Hynds D, Lu S, Nikiforou N, Petric M, Simoniello R and Voutsinas G 2016 *J. Phys. Conf. Ser.* **These Proceedings**
[10] Gaede F, Behnke T, Graf N and Johnson T 2003 *eConf* **C0303241** TUKT001 (*Preprint* `physics/0306114`)
[11] Alwall J *et al.* 2007 *Comput. Phys. Commun.* **176** 300–304 (*Preprint* `hep-ph/0609017`)
[12] Dobbs M and Hansen J B 2001 *Comput. Phys. Commun.* **134** 41–46