EPJ Web of Conferences **214**, 02037 (2019) https://doi.org/10.1051/epjconf/201921402037
CHEP 2018

New Developments in DD4hep

Marko Petrič[1,*], *Markus* Frank[1], *Frank* Gaede[2], and *André* Sailer[1]

[1]CERN, CH-1211 Geneva 23, Switzerland
[2]DESY, Notkestraße 85, D-22607 Hamburg, Germany

Abstract. For a successful experiment, it is of utmost importance to provide a consistent detector description. This is also the main motivation behind DD4hep, which addresses detector description in a broad sense including the geometry and the materials used in the device, and additional parameters describing, e.g., the detection techniques, constants required for alignment and calibration, description of the readout structures and conditions data. An integral part of DD4hep is DDG4 which is a powerful tool that converts arbitrary DD4hep detector geometries to Geant4 and provides access to all Geant4 action stages. It is equipped with a comprehensive plugins suite that includes handling of different IO formats; Monte Carlo truth linking and a large set of segmentation and sensitive detector classes, allowing the simulation of a wide variety of detector technologies. In the following, recent developments in DD4hep/DDG4 like the addition of a ROOT based persistency mechanism for the detector description and the development of framework support for DDG4 are highlighted. Through this mechanism an experiment's data processing framework can interface its essential tools to all DDG4 actions. This allows for simple integration of DD4hep into existing experiment frameworks.

1 Introduction

The DD4hep package [1] is a software toolkit that aims to unify aspects of detector description at different stages of an experiment and increases the efficiency and the ease of use. The goal is to provide a consistent description from a single source in simulation, reconstruction and analysis phases. The advantages of such a consistent, comprehensive description are clearly established from experience in high energy physics experiments.

The DD4hep toolkit was developed using experience from the LHCb experiment [2] and the efforts of the Linear Collider Community [3]. The aim is to cover on the one side the geometrical and material properties, and on the other side also providing parameters such as alignment constants and calibration, description of the readout structures and conditions, and those directly associated with detection techniques. The main idea behind DD4hep is to build on widely used pre-existing software to achieve a comprehensive generic detector description. The construction and visualisation of the detector geometry are implemented through the ROOT geometry package [4], while the detector simulation is performed via an interface with the Geant4 simulation toolkit [5].

[*]e-mail: marko.petric@cern.ch

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

2 Main objectives

The requirements which DD4hep aims to satisfy are:

- **Complete detector description:** A full geometry description including all structures and the corresponding materials, the attributes necessary for visualization, the parameters needed for alignment, calibration, and environment description, as well as detector readout information.
- **Coverage of the full life cycle of an experiment:** Offering functionality for all stages in the life-cycle of the experiment, from concept development, through construction to operation, providing a smooth transition between various stages.
- **Single source of information:** Containing all of the necessary information for interpretation of data in all stages, from simulation to reconstruction and analysis.
- **Ease of use:** Provides the final user with a simple and intuitive interface, which requires minimal external dependencies.

3 The DD4hep toolkit

DD4hep is built around a core based on the ROOT TGeo package, named Generic Detector Description Model (GDDM) . It is an in-memory model, that consists of a set of objects containing geometry and auxiliary information about the detector. Figure 1 shows a schematic depiction of the various DD4hep components, the interactions between them and the end-user applications, such as alignment, visualisation and reconstruction.

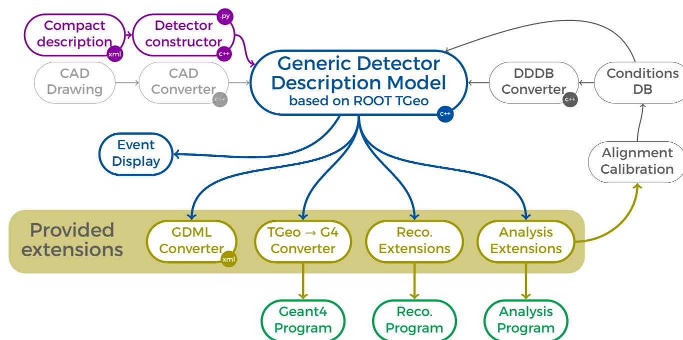

Figure 1. The components of the DD4hep detector geometry toolkit (light gray components represent planned future developments).

The working principles of DD4hep [6] and details about simulation [7] have been summarized at previous conferences. The aim of these proceedings is to summarize the most relevant changes that occurred since release DD4hep v00-17 (now at v01-09). Developments related to conditions and alignment are discussed separately in [8].

3.1 Namespace reorganization

To facilitate the ease of use and to streamline future developments, the DD4hep namespaces have been reorganized to be all lower case and shortened:

```
namespace DD4hep -> dd4hep;
namespace DD4hep::DDRec -> dd4hep::rec;
namespace DD4hep::Simulation -> dd4hep::sim;
namespace XML -> xml;
namespace JSON -> json;
```

All other namespaces have also been renamed according to the following pattern: the namespace `DD4hep::Geometry::` was incorporated into `dd4hep::`. All utilities are moved to `dd4hep::detail`. LCDD (name deriving from Linear Collider Detector Description) was renamed to `Detector` to more clearly denote the generic nature of the object.

3.2 Changes to the VolumeManager interface

Due to the ambiguous meaning of the function used to lookup a placement from the volume manager, discrepancies of how clients use this function have been observed: it may be considered as the placement of the closest detector element – a functionality used by various tests – or to be the placement of the sensitive volume itself. Since there can be sensitive volumes which are directly connected to more than one `DetElement` structure, this function was split to resolve this ambiguity. Since each sensitive volume in the DD4hep tests is represented by a `DetElement` structure, so far both approaches referenced the same placed volume.

```
/// Lookup a physical (placed) volume identified by its 64 bit hit ID
PlacedVolume lookupVolumePlacement(VolumeID volume_id) const;
/// Lookup a physical (placed) volume of the detector element
/// containing a volume identified by its 64 bit hit ID
PlacedVolume lookupDetElementPlacement(VolumeID volume_id) const;
```

3.3 The ROOT persistency mechanism

A new functionality of saving and restoring the full detector description from ROOT files has been added. Object extensions can also be stored, this requires a dictionary for the extension itself and a dictionary for the class holding the extension. These are:

```
// for DetElement extensions.
dd4hep::DetElement::DetElementExtension<IFACE,CONCRETE>
// for simple extension managed by the user framework (user calls explicitly
↪  destructor).
dd4hep::SimpleExtension<IFACE,CONCRETE>
// for simple extension managed by dd4hep (dd4hep calls automatically
↪  destructor on hosting object destruction).
dd4hep::DeleteExtension<IFACE,CONCRETE>
// As above, but these extensions support calling the copy constructor of
↪  the embedded object and hence allow to copy also the hosting objects.
dd4hep::CopyDeleteExtension<IFACE,CONCRETE>
```

The functionality has some limitations, concerning the usage of void pointers and the impossibility to save callback objects, which store in-memory pointers to member functions. These depend on the loading of libraries at run-time and hence may differ from execution to execution. To persist these objects it is necessary to no longer use the type-info of the objects as an identifier, but rather a 64-bit-hash of the raw type-info-name. This relies on the name being identical across platforms. This is typically true for linux, but not enforced by any standard.

3.4 Cell ID encoding

Information about the location of a sensitive cell that was hit in the simulation is stored in DD4hep via help of the `BitField64` class, which is not thread safe. A new thread-safe class `BitFieldCoder` was introduced as a replacement for `BitField64`, while `BitField64` was reimplemented using `BitFieldCoder` and exhibits thread safety if used locally. It can be instantiated from `const BitFieldCoder*`. Heap allocation of `BitFieldElements` was removed and `move` constructors for an efficient filling of vectors were added.

3.5 Support for importing and exporting GDML data

DD4hep now supports importing and exporting partial geometry trees with GDML. This however requires ROOT version 6.12, since some changes were necessary in the ROOT GDML handlers. The plugin implementation can be reviewed in `DDCore/src/gdml/GdmlPlugin.cpp`.

3.6 External framework support in DDG4

The integration of DDG4 into event processing frameworks has been simplified. The `Geant4Context` is the main thread specific accessor to DDG4 and the user framework.

Access to the DD4hep objects is via the `Geant4Context::detectorDescription()` call, to DDG4 as a whole is supported via `Geant4Context::kernel()` and to the user framework using a specialized implementation of `template <typename T> T& userFramework() const`. User defined implementations must be specialized somewhere in a compilation unit of the user framework, not in an header file. The framework object could host e.g. references for histogramming, logging and data access. In this way any experiment related data processing framework can exhibit its essential tools as DDG4 actions. An example specialised implementation is as follows:

```
struct Gaudi  {
  IMessageSvc*   msg;
  IHistogramSvc* histos;
  // ....
};
template<> Gaudi& Geant4Context::userFramework<Gaudi>()  const  {
 UserFramework& fw = m_kernel->userFramework();
 if ( fw.first && &typeid(T) == fw.second ) return *(T*)fw.first;
 throw std::runtime_error("No user specified framework context present!");
}
```

To access the user framework, after initialization the following call is then used:

```
Gaudi * fw = context->userFramework<Gaudi>();
GaudiKernel& kernel = // initialize kernel ;
kernel.setUserFramework(fw);
```

The `G4RunManager` is now instantiated via a plugin. To allow for user defined run managers in DDG4, the run manager is encapsulated in a `Geant4Action`. An example implementation can be found `DDG4/plugins/Geant4RunManagers.cpp`. Currently there are two factories implemented; `G4RunManager` is a factory that invokes the single threaded `G4RunManager` and `G4MTRunManager` is a factory that invokes the multi threaded `G4MTRunManager`. The factory names here are identical to the names of the native Geant4 classes.

3.7 String parsing

The Detector class was expanded to support parsing of XML from string. The existing view `dd4hep::DetectorLoad` was enhanced to allow this functionality:

```
Detector detector = ....;
/// We parse the raw XML string
DetectorLoad loader(detector);
loader.processXMLString(buffer,0);
```

A detailed example can be reviewed under `ClientTests/src/XML_InMemory.cpp`.

3.8 Geometry building state

The Detector object has a new state function `Detector::state()` which returns three values:

```
/// The detector description states
enum State   {
  /// The detector description object is freshly created. No geometry
  ↪  nothing.
  NOT_READY = 1<<0,
  /// The geometry is being created and loaded. (parsing ongoing)
  LOADING   = 1<<1,
  /// The geometry is loaded.
  READY     = 1<<2
};
```

It starts with `NOT_READY`, moves to `LOADING` once the geometry is opened and goes to `READY` once the geometry is closed. The initial object is invalid and gets created only once the geometry is opened. As a corollary, the object may not be accessed before. Geometry parsers must take this behavior into account.

3.9 SensitiveDetector types behaviour

The sensitive detector type defined in the detector constructors is no longer changed intransparently in the users' background. This does not work with Geant4, where a mapping of these types must be applied to supported sensitive detectors. Now the mapping of a `SensitiveDetector` type (e.g. "tracker") is strictly in the python setup. The default factory to create any sensitive detector instance in Geant4 is a property of the `Geant4Kernel` instance and defaults to:

```
declareProperty("DefaultSensitiveType", m_dfltSensitiveDetectorType =
 ↪  "Geant4SensDet");
```

Since the actual behavior is defined in the sequencer instantiation this default should be sufficient for most cases, otherwise the factory named `Geant4SensDet` may be overloaded.

3.10 Parallel geometries

It is possible to define volumes in a parallel world, which might be used for track or event reconstruction or for distinguishing different behaviors in the association of Monte Carlo truth information or any other means. In principle any volume hierarchy may be attached to the parallel world. None of these volumes participate in the tracking as long as the “connected” attribute is set to false. The hierarchy of parallel world volumes can be accessed from the main detector object using:

```
dd4hep::Volume parallel = dd4hep::Description::parallelWorldVolume()
```

This parallel world volume is created when the geometry is opened together with the world volume itself. If the name of the volume is “`tracking_volume`” within the compact notation it is declared as the Detector’s trackingVolume entity and is accessible as well:

```
dd4hep::Volume trackers = dd4hep::Description::trackingVolume()
```

Although the concept is available in the DD4hep core, its configuration from XML is only implemented for the compact notation. If the volume should be connected to the world one has to set `connected = true`. This is useful for debugging because the volume can be visualized; otherwise set `connected = false` to make the volume only part of the parallel-world. The volume is always connected to the top level. The anchor detector element defines the base transformation to place the volume within the (parallel) world.

```
<parallelworld_volume name="tracking_volume" anchor="/world" material="Air"
 ↪ connected="true" vis="VisibleBlue">
  <shape type="BooleanShape" operation="Subtraction">
    <shape type="BooleanShape" operation="Subtraction">
      <shape type="BooleanShape" operation="Subtraction"  >
        <shape type="Tube" rmin="0*cm" rmax="100*cm" dz="100*cm"/>
        <shape type="Cone" rmin2="0*cm" rmax2="60*cm" rmin1="0*cm"
         ↪ rmax1="30*cm" z="40*cm"/>
        <position x="0*cm" y="0*cm" z="65*cm"/>
      </shape>
      <shape type="Cone" rmin1="0*cm" rmax1="60*cm" rmin2="0*cm"
       ↪ rmax2="30*cm" z="40*cm"/>
      <position x="0" y="0" z="-65*cm"/>
    </shape>
    <shape type="Cone" rmin2="0*cm" rmax2="55*cm" rmin1="0*cm" rmax1="55*cm"
     ↪ z="30*cm"/>
    <position x="0" y="0" z="0*cm"/>
  </shape>
  <position x="0*cm"   y="50*cm" z="0*cm"/>
  <rotation x="pi/2.0" y="0"     z="0"/>
</parallelworld_volume>
```

The above example shows how the subtraction of shapes including translations can be defined with the help of `BooleanShape`. With such an approach an arbitrary parallel world volume can be created.

3.11 Enhancement of assemblies, regions and production cuts

Particle specific production cuts may now be specified in the compact notation. These production cuts are specified as `cut` entities in the `limitset`. The hierarchy of cuts to be applied

is: if present particle specific production cuts for a region exist, they are applied – otherwise the `cut` attribute of the compact region specification is used. If none of the above is specified, the global Geant4 cut is automatically applied by Geant4. An example is shown below:

```
<limits>
  <limitset name="VXD_RegionLimitSet">
    <!--
         These are particle specific limits applied to the region
         ending in Geant4 as a G4UserLimits instance
    -->
    <limit name="step_length_max"  particles="*" value="5.0" unit="mm" />
    <limit name="track_length_max" particles="*" value="5.0" unit="mm" />
    <limit name="time_max"         particles="*" value="5.0" unit="ns" />
    <limit name="ekin_min"         particles="*" value="0.01" unit="MeV" />
    <limit name="range_min"        particles="*" value="5.0" unit="mm" />
    <!--
         These are particle specific production cuts applied to the region
         ending in Geant4 as a G4ProductionCuts instance
    -->
    <cut   particles="e+"          value="2.0"   unit="mm" />
    <cut   particles="e-"          value="2.0"   unit="mm" />
    <cut   particles="gamma"       value="5.0"   unit="mm" />
  </limitset>
</limits>
```

4 Conclusion

The DD4hep Detector Description Toolkit provides an extended toolset for detector description, event simulation and reconstruction. The user community of DD4hep is growing as is the list of features developed to help clients and users of the toolkit. The toolkit is under investigation for future software implementation in the LHCb and CMS collaborations. These proceedings mostly cover features developed for clients of the toolkit to make their adoption easier.

Acknowledgements

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

References

[1] M. Frank, F. Gaede, M. Petric, A. Sailer, *AIDASoft/DD4hep* (2018), webpage: http://dd4hep.cern.ch/, `https://doi.org/10.5281/zenodo.592244`
[2] S. Ponce, P. Mato Vila, A. Valassi, I. Belyaev, eConf **C0303241**, THJT007 (2003), `physics/0306089`
[3] LCC, *The Linear Collider Collaboration*, `https://www.linearcollider.org/`
[4] R. Brun, A. Gheata, M. Gheata, Nucl. Instrum. Meth. **A502**, 676 (2003)
[5] S. Agostinelli et al. (GEANT4), Nucl. Instrum. Meth. **A506**, 250 (2003)
[6] M. Frank, F. Gaede, C. Grefe, P. Mato, J. Phys. Conf. Ser. **513**, 022010 (2014)
[7] M. Petrič, M. Frank, F. Gaede, S. Lu, N. Nikiforou, A. Sailer, Journal of Physics: Conference Series **898**, 042015 (2017)
[8] M. Frank, F. Gaede, M. Petrič, A. Sailer, Same proceedings (2018)