SOFTWARE DESIGN FOR GEOMETRY TOOLKIT

Mato, Pere (CERN)

02 August 2012

The research leading to these results has received funding from the European Commission under the FP7 Research Infrastructures project AIDA, grant agreement no. 262025.

This work is part of AIDA Work Package 2: Common software tools.

The electronic version of this AIDA Publication is available via the AIDA web site <http://cern.ch/aida> or on the CERN Document Server at the following URL: <http://cdsweb.cern.ch/search?p=AIDA-D2.3>
Abstract:
The initial design of the Geometry Toolkit is described. The design addresses the main use cases that are foreseen to be covered by the set of detector description software tools developed in Task 2.2. Main design choices have been initially tested for functionality and usability in a prototype called DD4Hep (Detector Description for HEP) that has been developed in the last few months. This document mainly presents the overall design of the main components of the toolkit as well as some detail design of key aspects of the implementations that are being developed.
Copyright notice:

Copyright © AIDA Consortium, yyyy.
For more information on AIDA, its partners and contributors please see www.cern.ch/AIDA

The Advanced European Infrastructures for Detectors at Accelerators (AIDA) is a project co-funded by the European Commission under FP7 Research Infrastructures, grant agreement no 262025. AIDA began in February 2011 and will run for 4 years.

The information herein only reflects the views of its authors and not those of the European Commission and no warranty expressed or implied is made with regard to such information or its use.

Delivery Slip

<table>
<thead>
<tr>
<th>Name</th>
<th>Partner</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authored by</td>
<td>P. Mato</td>
<td>CERN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>04/06/12</td>
</tr>
<tr>
<td>Edited by</td>
<td>P. Mato and G. Cosmo</td>
<td>CERN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dd/mm/yy</td>
</tr>
<tr>
<td>Reviewed by</td>
<td>G. Cosmo [Task coordinator]</td>
<td>CERN</td>
</tr>
<tr>
<td></td>
<td>F. Gaede [WP coordinator]</td>
<td>DESY</td>
</tr>
<tr>
<td></td>
<td>L. Serin [Scientific coordinator]</td>
<td>CNRS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dd/mm/yy</td>
</tr>
<tr>
<td>Approved by</td>
<td>Steering Committee</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>dd/mm/yy</td>
</tr>
</tbody>
</table>
EXECUTIVE SUMMARY

The geometry, and in general, the detector description is an essential component for the development of the data processing applications in the high energy physics experiments.

We describe in this document the guiding requirements and the architectural design for a generic detector description toolkit, as well as the main implementation choices. The toolkit will be built reusing already existing components of widely used packages and provide the missing functional elements and interfaces to offer a complete and coherent detector description solution to experiments. In addition, a new unified library of geometrical shapes reusing elements from existing geometry packages in ROOT and Geant4 has also been designed and is being implemented. The main motivation is to facilitate long-term maintenance and the same time improving the performance by selecting the best implementation.

The design is strongly driven by easy use of the toolkit. Developers of detector descriptions and applications using them should provide minimal information and minimal specific code to achieve the desired result.

The design described in this document is currently being prototyped and usable code is already available in the AIDA WP2 repository. Several client applications are under development to validate the design and provide feedback in functionality and usability. This input is essential for the development of the final implementation.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Introduction</td>
<td>5</td>
</tr>
<tr>
<td>1.1.</td>
<td>Scope and Requirements</td>
<td>5</td>
</tr>
<tr>
<td>1.2.</td>
<td>Existing Software components</td>
<td>7</td>
</tr>
<tr>
<td>1.2.1.</td>
<td>ROOT geometry package (TGeom)</td>
<td>7</td>
</tr>
<tr>
<td>1.2.2.</td>
<td>Geant4 geometry package</td>
<td>7</td>
</tr>
<tr>
<td>1.2.3.</td>
<td>GDML interchange format</td>
<td>7</td>
</tr>
<tr>
<td>1.2.4.</td>
<td>Python Scripting</td>
<td>7</td>
</tr>
<tr>
<td>1.2.5.</td>
<td>XML Parser</td>
<td>7</td>
</tr>
<tr>
<td>2.</td>
<td>Toolkit Design</td>
<td>8</td>
</tr>
<tr>
<td>2.1.</td>
<td>Compact Detector Description</td>
<td>8</td>
</tr>
<tr>
<td>2.2.</td>
<td>Detector Constructors</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1.</td>
<td>Detector Models Library</td>
<td>11</td>
</tr>
<tr>
<td>2.3.</td>
<td>Generic Detector Description Model</td>
<td>11</td>
</tr>
<tr>
<td>2.3.1.</td>
<td>Implementation Choices</td>
<td>12</td>
</tr>
<tr>
<td>2.3.2.</td>
<td>Detector Element tree versus the Geometry hierarchy</td>
<td>13</td>
</tr>
<tr>
<td>2.4.</td>
<td>Reconstruction Extensions</td>
<td>14</td>
</tr>
<tr>
<td>2.4.1.</td>
<td>Implementation choices</td>
<td>14</td>
</tr>
<tr>
<td>2.4.2.</td>
<td>Accessing the detector description</td>
<td>15</td>
</tr>
<tr>
<td>2.5.</td>
<td>Geant4 Converters and User Code Plugins</td>
<td>15</td>
</tr>
<tr>
<td>2.6.</td>
<td>Detector Alignment Support</td>
<td>16</td>
</tr>
<tr>
<td>2.7.</td>
<td>Interchange Formats</td>
<td>16</td>
</tr>
<tr>
<td>2.8.</td>
<td>Detector Display</td>
<td>17</td>
</tr>
<tr>
<td>3.</td>
<td>Unified library for geometrical primitives</td>
<td>17</td>
</tr>
<tr>
<td>3.1.</td>
<td>Introduction</td>
<td>17</td>
</tr>
<tr>
<td>3.2.</td>
<td>Design and Methodology of work</td>
<td>18</td>
</tr>
<tr>
<td>3.3.</td>
<td>Testing</td>
<td>20</td>
</tr>
<tr>
<td>3.3.1.</td>
<td>Solid Batch Test (SBT)</td>
<td>20</td>
</tr>
<tr>
<td>3.3.2.</td>
<td>Data Analysis and Performance (DAP)</td>
<td>20</td>
</tr>
<tr>
<td>3.3.3.</td>
<td>Optical Escape</td>
<td>21</td>
</tr>
<tr>
<td>4.</td>
<td>Conclusion</td>
<td>22</td>
</tr>
<tr>
<td>5.</td>
<td>References</td>
<td>23</td>
</tr>
</tbody>
</table>
1. **INTRODUCTION**

The goal of the Task 2.2 within AIDA is to develop a coherent set of software tools for the description of HEP detectors from a single source of information and to provide appropriate and consistent views to the simulation, reconstruction and analysis of data applications. This detector description includes the geometry and material information, the detection techniques, the alignment and calibration, the readout structures, the conditions data, etc. The aim is to reuse existing software components (e.g. Geant4 and ROOT geometry packages) and to complete the toolkit with the missing elements in order to achieve an overall consistent set of tools.

Very early in the project execution we identified one self-consistent and fairly independent sub-task that we wanted to pursue. This is to develop a common library of 3D geometrical primitives taking the best implementation for each shape from existing geometry packages in ROOT and Geant4, together with enhancing the set of primitives with missing ones and more optimal implementations. The rational of this new library is to simplify the long-term maintenance of both Geant4 and ROOT packages by modifying them to use this common library. This unified library of geometrical primitives fits nicely in the global design of the set of software tools as it will be described later in this document.

The design of the Geometry Toolkit described in this document is shaped on the experience of detector description systems that have been implemented for the LHC experiments in particular the LHCb experiment [1] as well as the lessons learnt from the various implementations of geometry description tools developed for the Linear Collider community [2,3]. Designing a coherent set of tools, with most of the basic components already existing in one form or another, is a great opportunity for getting the best of all existing solutions.

We will start in section 1.1 by stating the scope and the high-level requirements for the toolkit. This is basically the high level vision of what we think the toolkit should be providing to the experimental communities. Then, in section 1.2 we will review the existing software components on which we want to base our toolkit. As it has been already mentioned in the goals of task 2.2 the aim is to re-use a much as we can existing software components and not re-inventing everything from scratch for which we do not have the available effort. Section 2 will introduce the high-level or architectural design of the toolkit, and we will develop in the subsequent subsections some of the design aspects of the various functionality components and their interfaces. Section 3 will be devoted to design and methodology of work for the unified library of geometrical primitives.

### 1.1. **SCOPE AND REQUIREMENTS**

What we actually intend to develop is a ‘Detector Description’ toolkit, which is a generalization of a purely geometry toolkit. Experience from the LHC experiments shows that this generalization is very beneficial in order to get a more coherent set of tools. This is particularly important later in the experiment life cycle, in which the detector data will be stored in the so-called ‘conditions database’ that will be able to deliver a valid set of data for each moment in the detector lifetime. The detector data fully describe and qualify the detecting apparatus and are used to interpret the event or collision data. They consist typically of the description of the detector in terms of its geometry, the calibration and alignment constants, the environmental parameters such as temperature and pressure, etc. The toolkit
should provide functionality to manage in a coherent way all the detector data and to ease the implementation of the specifics of the sub-detectors in each experiment.

Besides the basic tools and interfaces that the toolkit will consists of, we also intent to deliver a number of specific software modules that describe concrete detector types that can easily be configured and are ready to be used by experimental communities that need to develop quickly new detector designs to be evaluated. This library or collection of modules can also be used as examples for other users to facilitate the development of their specific detector descriptions.

The following is the list of main requirements to drive the design of the software toolkit:

- **Full Detector Description.** The toolkit should be able to manage the data describing the detector geometry, detector materials, visualization attributes, detector readout information, alignment and calibration parameters, environmental parameters, etc. Special emphasis will be put to support alignment from the early versions of the toolkit.

- **Full Experiment life cycle.** The toolkit should be able to support all phases of the life-cycle of an experiment, from the detector concept development, to the detector optimization, construction and later operation of it. The transition from one phase to the next should be as easy as possible without having to make new developments. The initial phases are characterized by very ‘ideal’ detector descriptions, that is, with only very few parameters physicists should be able to describe new detector designs. In the latest phases the reality is very different than the ideal detector, and each detector element or module will have to have its own specific parameters and conditions.

- **Consistent Description.** The toolkit should envisage a single source of all detector information for all typical data processing applications such as simulation, reconstruction, online trigger, data analysis, etc. It is now generally accepted that the only way to guarantee a consistent description is by having a truly single source of information. Any attempt in the past to keep in synch several detector descriptions has been a problem in the long run. The logical consequence of this is that the information in the detector description database should be the union of the information needed by all applications; the level of detail of the views of this should be selectable by the client application.

- **Ease of Use.** This requirement should be driving the design and implementation. We should minimize the places in which the physicists should enter information. The developer using the toolkit should only concentrate with the specifics of his/her detector and leave the commonality to be provided by the toolkit itself.

We have not produced a more detailed list of user requirements at this current phase because we would like to apply a more ‘agile’ software development process. The manifesto for agile software development [5] welcomes changing requirements, even late in the development. Therefore, the listed requirements should be sufficient to provide an initial vision of what the toolkit should be. Then, developing initial prototypes that we will make available to potential users, would be the best way to obtain more concrete use cases that will need to be supported and developed. All this will be done in short development cycles to obtain feedback as soon as possible in the development process.
1.2. EXISTING SOFTWARE COMPONENTS

We would like to re-use existing and well-established software components already in use in the HEP community. This section describes briefly the functionality of each one and how it could fit in the overall toolkit design.

1.2.1. ROOT geometry package (TGeom)

The ROOT geometry package is a tool for building, browsing, navigating and visualizing detector geometries. It is part of the ROOT [7] project and delivered together with it. The code works standalone with respect to any tracking Monte-Carlo engine; therefore, it does not contain any constraints related to physics. However, the navigation features provided by the package are designed to optimize particle transport through complex geometries, working in correlation with simulation packages such as GEANT3, GEANT4 and FLUKA. In addition, the ROOT geometry package comes with very nice and sophisticated 3D visualization functionality, which will be ideal for building detector and event displays.

1.2.2. Geant4 geometry package

The Geant4 Geometry package [6] is a key component of the Geant4 simulation toolkit [5]. It has been designed to exploit at the best the features provided by Geant4, allowing the description of the geometrical structure of complex detectors in a natural way, ranging from a few up to hundreds of thousands volumes of the LHC experiments. As such, it is not meant to be used as a standalone package. It provides a great variety of geometrical shapes, advanced techniques for optimizing tracking in the geometrical model, handling logical, and “physical” volumes separately in a tree organization.

1.2.3. GDML interchange format

The Geometry Description Markup Language (GDML) [5] is an application-independent geometry description format based on XML. It can be used as the primary geometry implementation language as well as to provide a geometry data exchange format for the existing applications.

1.2.4. Python Scripting

Python [10] is a very flexible scripting programming language that has become very popular in the HEP community and complements the C++ language used when real performance is required. People can learn to use Python and see almost immediate gains in productivity and lower maintenance costs. C++ and Python can be easily interfaced using the PyROOT extension module that allows the user to interact with any C++ class from the Python interpreter. This is done generically using the ROOT dictionary; therefore there is no need to generate any Python wrapper code to include new C++ classes.

1.2.5. XML Parser

Xerces-C++ is used as XML parser. It is a validating XML parser written in a portable subset of C++. It makes it easy to give your application the ability to read and write XML data.
2. TOOLKIT DESIGN

The architecture overview of the toolkit is shown in Figure 1. It shows the main components of the toolkit and their interfaces to the end-user applications such as the simulation, reconstruction, alignment and visualization. Each of the components shown in the figure will be described in more details in subsequent sections of this document. The central element of the toolkit is the so-called generic detector description model. This is an in-memory model, i.e., a set of C++ objects holding the data describing the geometry and other information of the detector. The rest of the toolkit is just a set of tools and interfaces to input or output the information from this generic detector model.

![Figure 1 Main Components of the Detector Description Toolkit](image)

2.1. COMPACT DETECTOR DESCRIPTION

Inspired from the work of the ILC detector simulation, we are defining a compact detector description, which is ideal for the conceptual design phases of an experiment. This is probably not going to be adequate later in the detector life cycle when a more realistic detector with deviations from the ideal would need to be entered into the detector description toolkit. The description is in XML format with high-level elements and attributes. The main idea is to be minimalistic in terms of user-provided parameters that fully describe an ideal detector.

We have chosen the XML format for the compact description for the flexibility it offers to introduce new elements and attributes. This is in particular very relevant when users will need to describe custom detectors for which we will not provide generic descriptions. For this...
reason, we have also decided to configure the XML parser as non-validating against an XML fix schema. This allows us to provide this flexibility without burdening the end-user too much. Figure 2 shows a partial example of how to describe a very ideal silicon-based vertex detector with regular disposition of ladders in 5 layers in total.

```xml
<detector name="VXD" type="ILDExVXD" vis="VXDVvis" id="1">
  <layer id="1" vis="VXDLayerVis">
    <support thickness="0.03mm" material="Carbon" vis="VXDSupportVis"/>
    <ladder zhalf="65+mm" radius="16+mm" offset="-2+mm" thickness="0.1+mm" material="Silicon" number="10"/>
  </layer>
  <layer id="2" vis="VXDLayerVis">
    <support thickness="0.03mm" material="Carbon" vis="VXDSupportVis"/>
    <ladder zhalf="65+mm" radius="18+mm" offset="-2+mm" thickness="0.1+mm" material="Silicon" number="10"/>
  </layer>
  <layer id="3" vis="VXDLayerVis">
    <support thickness="0.03mm" material="Carbon" vis="VXDSupportVis"/>
    <ladder zhalf="125+mm" radius="38+mm" offset="-2+mm" thickness="0.1+mm" material="Silicon" number="11"/>
  </layer>
  <layer id="4" vis="VXDLayerVis">
    <support thickness="0.03mm" material="Carbon" vis="VXDSupportVis"/>
    <ladder zhalf="125+mm" radius="40+mm" offset="-2+mm" thickness="0.1+mm" material="Silicon" number="11"/>
  </layer>
  <layer id="5" vis="VXDLayerVis">
    <support thickness="0.03mm" material="Carbon" vis="VXDSupportVis"/>
    <ladder zhalf="125+mm" radius="60+mm" offset="-2+mm" thickness="0.1+mm" material="Silicon" number="17"/>
  </layer>
</detector>
```

Figure 2 Example of compact detector description for silicon based vertex detector showing the basic physical dimensions, organization in layers, materials composition, visualization attributes, etc.

### 2.2. DETECTOR CONSTRUCTORS

The work of processing the high level description from a compact detector XML file and converting it into a expanded hierarchy of volumes of given shapes and dimensions placed in space is done in the so-called ‘detector constructors’. These are relatively small code fragments that get as input one XML element that represents a single ‘detector’ instance and build its geometry model in memory making use of the elements from the ‘generic detector description model’ (see later). These code fragments are responsible of interpreting the elements and attributes of the compact detector description. The toolkit will take care of managing and collecting all these code fragments. They will be called during the initialization phase of the application by providing as argument a single XML element with their attributes and sub-elements, which will be sufficient for constructing a single detector with the help of some functionality also provided by the toolkit. Users will only need to focus on a single detector type at the time to be able to construct a complex and large detector setup.

Two implementations are currently supported at this moment. One is based on C++ code fragments using the Xerces-C++ XML parser, and the other is based on Python fragments using the native XML parser provided by Python. We plan to exercise both implementations and after gaining some experience standardize on one of them. The C++ implementation has clearly better execution time performance and is able to detect errors at compiler time, but the code is slightly larger and quite technical. The Python implementation is very readable and compact but errors are only detected at execution time.

A naming convention does the link between a ‘detector’ in the compact description and its ‘constructor’. The attribute ‘type’ in the ‘detector’ element is used to locate the adequate detector constructor to call. For example in the Python implementation the function named...
‘detector_<type>()’ will be called when a detector of a given type is supposed to be constructed.

The next two figures show an example of a detector constructor. In this case, it is the detector constructor for the silicon vertex detector described in Figure 2 in its C++ and Python forms. The purpose of this is to visualize what is needed in terms of code for interpreting the high level description of a detector. Obviously during the implementation we may change the actual functionality of the detector model and therefore the constructors will need to be adapted accordingly.

```
#include "DD4hep/DD4hepFactoryHelper.h"
#include "ILDExVDX.h"
using namespace std;
using namespace DD4hep;
using namespace DD4hep::Geometry;

namespace DD4hep { namespace Geometry {

    template <> Ref_t DetElementFactory<ILDExVDX>::create(LCDDG lcdd, const xml_h & e, SensitiveDetectors) {
        xml_det_t x_det = e;
        string name = x_det.nameStr();
        ILDExVDX vdx (lcdd.name, x_det.typeStr(), x_det.id());
        Volume mother = lcdd.pickMotherVolume(vdx);
        for(xml_coll_t c(e,xml_iter); c;++c) {
            xml_comp_t x_layer (c);
            xml_comp_t x_support (x_layer.child({x["support"]}));
            xml_comp_t x_ladder (x_layer.child({x["ladder"]}));
            int layer_id = x_layer.id();
            int nladders = x_ladder.number();
            string layername = name_ + _to_string(layer_id, "_layer"d);
            double dphi = 2. * PI / double(nladders);
            double zhalf = x_ladder.zhalf();
            double offset = x_ladder.offset();
            double sens_radius = x_ladder.radius();
            double supp_thickness = x_support.thickness();
            double thickness = 2 * atan(dphi / 2.); // sphere radius - sens_r_thick/2.
            Material sens_mat = lcdd.material(x_ladder.materialStr());
            Material supp_mat = lcdd.material(x_support.materialStr());
            Box ladderbox (lcdd.layername("ladder").solid, sens_thick + supp_thickness / 2., width / 2., zhalf);
            Box sensbox (lcdd.layername("sens").solid, sens_thick / 2., width / 2., zhalf);
            Volume sensvol (lcdd.layername("sens").volume, sensbox, sens_mat);
            Box suppbox (lcdd.layername("supp").volume, supbox, supp_mat);
            Position senspos (-sens_thick + supp_thickness / 2., -sens_thick / 2., 0);
            Position supppos (-sens_thick + supp_thickness / 2., -sens_thick / 2., 0);
            sensvol.setVisAttributes(lcdd.visAttributes(x_layer.visStr()));
            laddervol.placeVolume(sensvol, senspos);
            laddervol.placeVolume(suppvols, supppos);
            for(int j = 0; j < nladders; ++j) {
                string laddername = layername + _to_string(j, "_ladder");
                double radius = sens_radius - ((sens_thick + supp_thickness) / 2. - sens_thick / 2.);
                double radius = sin(wphi) - offset * cos(wphi),
                Position pos(radius = cos(wphi)),
                mother.placeVolume(laddervol, pos, Rotation(0, 0, j * dphi));
            }
            vdx.setVisAttributes(lcdd, x_det.visStr(), laddervol);
            return vdx;
        }
    }

    DECLARE_NAMED_DETELEMENT_FACTORY(DD4hep, ILDExVDX);

Figure 3 An example of a detector constructor in C++
```
2.2.1. Detector Models Library

In addition to developing the different parts of the toolkit that should facilitate the creation of new detector models by end-users, we will also provide a library of configurable detector models (detector constructors) for various common detection technologies. This will range from silicon based tracking detectors to calorimeter detectors. The idea would be that for quick detector studies a rich set of generic models would be available and ready to be used.

The end-user will just need to modify the compact description using these detector types to start building a new detector concept by specifying physical dimensions, materials and a number of basic parameters. At the same time these detector models in the library can be used as working examples for specific and more customized detector types that a user may want to develop.

2.3. GENERIC DETECTOR DESCRIPTION MODEL

This is the heart of the detector description toolkit. Its purpose is to build in memory a model of the detector including its geometrical aspects, as well as structural and functional aspects. We base the current design on re-using the elements already existing in the ROOT geometry package and extend them when the required functional elements are not available.

Any detector is modelled as a tree of so called ‘detector elements’ with a parent-child relationship. The Detector Element is central to this design since it offers the natural entry point to any detector of the experiment from the applications. It may represent a complete sub-detector (e.g. ECAL), a part of a sub-detector (e.g. ECAL Barrel), a detector module, or any other convenient detector device. Its main purpose is to give access to the data associated with the detector.
to the detector device. For example, if the user writes some TPC reconstruction code, accessing the TPC detector element from this code will provide access the all TPC geometrical dimensions, the alignment and calibration constants and other slow varying conditions such as the gas pressure, end-plate temperatures and so on. In the design, the Detector Element class acts as a data concentrator.

We plan to use the ROOT geometry classes directly. There is no intention to create unnecessary interfaces to isolate the end-user from the actual implementation based on ROOT. Figure 5, shows a class diagram with the main players and their relationships of the design. We indicate in brackets the ROOT class name that we will use for the implementation. Not shown in the diagram are a couple of singletons: the TGeoManager and the LCDD, which provide management, bookkeeping and ownership to the model instances.

![Class diagram with the main classes and their relations for the Generic Detector Description Model](image)

2.3.1. Implementation Choices

In an attempt to mitigate the problems related to object ownership and garbage collection inherently of the C++ programming language, we have designed all classes of the detector description model separating explicitly the ‘data’ from its ‘behaviour’. Each class consists of a number of member methods or functions and a single data member, which is a pointer to a data object, actually a ‘plain old data’ (POD) object. This way, we can make as many copies of the object as we want, pass it as arguments to functions, crossing the boundary between C++ and Python, without having to worry about properly cloning them, garbage collect the unreferenced ones and so on. With this design it is also very easy to recast the data object part...
to a completely different behaviour, which is adequate to the application we are interested in (e.g. simulation, reconstruction) without having to make any complicated C++ construct and avoiding all compile-time and run-time dependencies.

2.3.2. Detector Element tree versus the Geometry hierarchy

The geometry part of the detector description is delegated to the ROOT classes. Logical Volumes are the basic objects used in building the geometrical hierarchy. They represent unpositioned objects but store all information about the placement of the other volumes they may contain. Therefore the same volume can be replicated several times in the geometry. In order to create a volume, the user has to put together a shape with its dimensions and a material. In addition, a Logical Volume also represents a system of reference in which the user places all the daughter volumes. Reusing Logical Volume instances in different placements is done to optimize memory consumption. This allows utilizing a reasonable amount of memory for describing the detailed geometry of today’s complex detector setups that may consist of millions of volumes. The difficulty is in identifying a given positioned volume in space (touchable volume). Identification will be needed for example when applying some misalignment of one these volumes. The solution is to use the full path from mother to daughter starting from the World volume.

The tree structure of Detector Elements is a parallel structure to the geometrical hierarchy. This structure will probably not go as deep as the geometrical one since there would not need to associate detector information at very fine-grain level. In other words, it is very unlikely we would need to associate detector information (alignment, conditions, etc.) to a little metallic screw in a support structure, while this screw and many other replicas of it may be important for the geometry description in order to account for its material contribution in the simulation application. Therefore, we have decided that the tree structure of Detector Elements will be fully degenerate. Each detector element object will be placed only once in the detector element tree. This facilitates enormously the access to the information. To illustrate how the two structures are related we have created an object diagram of a hypothetical TPC detector as shown in Figure 6. The important point here is that the relationship between the Detector Element and the placements (TGeoNode) does not only require a single pointer to the TGeoNode object but also the full path from the top of the detector geometry model in order to resolve the ambiguity due to the reuse of Logical Volumes.
2.4. RECONSTRUCTION EXTENSIONS

As it is depicted in Figure 1 the reconstruction application will get access to the detector description by means of a set of classes that will be extending the basic functionality of the common detector element. These will be specializations for the reconstruction program. Typically these class extensions will be in charge of providing specific answers to the questions formulated by the reconstruction algorithms (pattern recognition, tracking, vertexing, particle identification, jet finding, etc.) One example could be to transform an ECAL calorimeter cell ID into a 3D space position in the global coordinate system. Obviously, a generic detector description toolkit would be unable to answer this concrete question, however it can provide a convenient framework in which the developer can slot-in some code that would convert the cell ID into a local coordinate using some parameters stored in the XML compact description and then use the generic geometry description to convert the local coordinates into global ones. Another example could be to get the amount of material (radiation lengths) between two consecutive layers in the tracking detector. Again here, to answer this question, the functionality can be implemented by a collaboration between some specific code that understands the concrete tracking detector and its generic geometry description including the details of the materials used in the detector.

2.4.1. Implementation choices

At this moment we have not decided on a way to implement these reconstruction extensions. The naïve implementation using class inheritance in C++, meaning classes that inherit the

Grant Agreement 262025 PUBLIC 14 / 23
**detector element** class does not work well with the implementation choice presented in section 2.3.1. This is because extending a class will add new methods (or overwrite existing ones) but also will add new data members and therefore the object will not have the form of a single pointer to the data part of the object. We are currently prototyping several options for implementing these extensions.

### 2.4.2. Accessing the detector description

Making the access to detector description information as simply as possible has driven the design of the toolkit. After having build the detector model in memory from the compact geometry description, the entry point for the user is always a Detector Element that is accessed by its name. The generic Detector Element can then be ‘extended’ or customized to a specialized class that would provide the specific answers to specific questions as it is shown example code fragment in Figure 7. The current design allows us to customize a given generic detector element with different extensions that can be made specifically for each type of concrete application.

```cpp
LCDD ldc = LCDD::getInstance();
lccd.fromCompact(argv[1]);

GearTPC tpc(ldc.detector("TPC"));
cout << "Gear TPC functionality:" << endl;
cout << "-----> Inner Radius:" << tpc.getInnerRadius() << endl;
cout << "-----> Outer Radius:" << tpc.getOuterRadius() << endl;
cout << "-----> DriftLength:" << tpc.getMaxDriftLength() << endl;
```

*Figure 7* Fragment of code to illustrate how an end-user can access to geometrical information about a given detector, in this case a TPC

### 2.5. GEANT4 CONVERTERS AND USER CODE PLUGINS

The simulation program based on the Geant4 simulation toolkit will either require to have the detector geometry described in terms of Geant4 geometry objects or to re-implement the Geant4 navigator in terms of the navigator that is available in the generic detector description and to interface it to the simulation toolkit. We have opted for the first solution, since is the one that will be easy to validate since it is a one time geometry conversion can be easily validated. In this way Geant4 will be running natively with no added overhead due the additional interfaces. The only drawback is that during the initialization of the simulation, two copies of the geometry will co-exist in memory. The second mentioned option is the one chosen by the Virtual Monte Carlo developed by the Alice collaboration [12].

Converting the geometry and associated data shouldn’t be a big problem since the underlying geometry models for ROOT and Geant4 are very similar. In addition, the foreseen use of the unified solid library will facilitate enormously this conversion. For the prototype we have been using the VGM package to implement this conversion, but probably later we will be using something simpler and probably self-contained in the Geometry toolkit.

To complete the simulation application the developer needs to provide the so-called ‘user actions’ or ‘sensitive detectors’ with custom code and plug them into the Geant4 simulation engine. The toolkit should facilitate this operation. Typically this custom code is modelled as a user class implementing a given Geant4 interface. An instance of this class needs to be associated to a detector element in the geometry model in memory. The idea here is to supply
the necessary information in the persistent detector description (i.e. compact description) such that the toolkit is able to instantiate generically the user class and connect it to the geometry model. A plugin mechanism should allow us to extend and customize the functionality of the toolkit at run-time in a generic manner.

The implementation of a plugin mechanism can be build on top of the Python bindings to C++ (e.g. PyROOT). The mechanism to locate what library contains the user custom code, to load dynamically the library, and to construct generically by its type name an instance of the user class is already available in the Python C++ bindings.

2.6. DETECTOR ALIGNMENT SUPPORT

An important requirement for the toolkit has been to support alignment operations to the geometry description. This support is basically missing in all the currently used geometry description systems in the linear collider community. We need to have the possibility to input into the detector description alignment ‘deltas’ (differences with respect the ideal or measured position) or read them from an external source. A typical alignment application would consist of calculating a new set of ‘deltas’ from a given starting point. This new set of ‘deltas’ could then be loaded and applied again in order to validate the alignment by recalculating some alignment residuals.

For the implementation we will be using the already existing functionality of the ROOT geometry package. The class \texttt{TGeoPhysicalNode} is able to apply an [mis]-alignment to the actual ‘touchable’ objects in the geometry. These ‘touchable’ objects are identified by a path of positioned volumes starting with the top node (e.g. path=/TOP/A_1/B_4/C_3). With this functionally already available in the ROOT package, we need just to connect the \texttt{Detector Element} to these ‘deltas’ in order to facilitate the work of the end user. So, in this way the user needs to specify the path relative to the \texttt{Detector Element} and not from the top node or \texttt{World} volume.

The toolkit will also provide the necessary functionality to input the alignment deltas from various sources. The initial implementation will be based on simple XML files, which is already in use for the input of the compact geometry description. However we plan to provide interfaces to other sources such as the detector conditions database.

2.7. INTERCHANGE FORMATS

Figure 1 show the possibility to output and input the geometrical information in some interchange format between applications. We will be using the Geometry Description Markup Language (GDML), which is an application-independent geometry description format based on XML [5]. In principle, it can also be used as the primary persistent geometry implementation language but its verbosity would be a natural impediment during the early phases of the detector development.

Probably we would need to extent the existing GDML standard with the additional information we have in the generic geometry model, for example the detector elements, alignment constants, sensitive detectors, etc. This is done typically by adding new tags and new elements, which is naturally foreseen in XML. In this way we will be able to fully save a detector model and be able to restore it completely from a GDML file.
Having decided to use the ROOT geometry classes allow us also to write and read the geometry part in ROOT file format. This is not a directly editable format but has the advantage of being very efficient in disk space and processing time.

2.8. DETECTOR DISPLAY
As already mentioned, the decision of using the ROOT geometry package to implement the detector model allows us to benefit directly from its graphics capabilities. It is extreme simple to produce an OpenGL model of the detector geometry; a simple call to a ROOT function does the work. The user can interactively modify the viewpoint and many other attributes of the display.

Visualization attributes can be associated to the compact detector description to facilitate the conversion of the detector model to a 3D visualization representation.

![Detector Display Example](image)

*Figure 8 An example of a display of a simple TPC detector with Vertex detector described with an XML compact description*

3. UNIFIED LIBRARY FOR GEOMETRICAL PRIMITIVES

3.1. INTRODUCTION
Geometrical primitives are used in both simulation and reconstruction programs for composing the geometrical setups of detectors and experimental setups. The Geant4 Simulation Toolkit provides a well-tested implementation of such primitives; an alternative implementation is also provided within the ROOT framework. It is estimated that in both cases, a good 70-80% of the effort spent for code maintenance in the geometry modeller is due to the implementation of the geometrical primitives; the code, written in C++, includes all algorithms required for efficiently tracking particles with a high level of precision in an infinite variety of geometrical setups.
In order to reduce the effort required for support and maintenance, and converge on a unique solution based on high quality code, we aim to build a new software library for geometrical primitives. The new library can then be used for solid modelling in Monte Carlo detector simulations, and will unify the existing implementations in Geant4 and ROOT, eventually improving where possible both reliability and CPU performance of the implemented algorithms. The library will include ~25 primitives, corresponding to the complete set of solids which is part of the GDML (Geometry Description Mark-up Language) schema. In the medium term, we plan to replace the code now existing in the Geant4 and Root software packages with this new Unified Solids Library.

3.2. DESIGN AND METHODOLOGY OF WORK

One fundamental aspect of the approach to be adopted for the realization of the new library is to put in place a comprehensive testing suite, which can allow to systematically monitor correctness and performance of the new algorithms in comparison with the corresponding implementations in Geant4 and ROOT.

In order to achieve this, it is required to define and implement a basic infrastructure, which allows transparently exercising the new code in Geant4 and ROOT and easily comparing the functionality.

Figure 9 Bridge pattern classes to facilitate transition and parallel comparisons
In Figure 9, the design for the bridge pattern classes is shown. Based on such bridge classes, each geometrical shape implemented in the new Unified Solid Library can be used as native type in either Geant4 or ROOT, allowing for direct comparison of their functionality. Figure 10 shows the final class diagram for usage of the new geometrical primitives types in Geant4 and Root, once the bridge classes are removed.

Figure 10 Final configuration in Geant4 and ROOT for the use of the new geometrical primitives

The approach envisioned for the realization of the new library can be summarized as follows:

1. Define the interface to be adopted, by combining the APIs provided in Geant4 and ROOT, and implement this in the bridge classes.
2. Review all algorithms of the existing solids in Geant4 and ROOT; when possible adapt them, keeping the overall code quality, correctness and performance high.
3. For the cases where new shapes are introduced, develop the new algorithms and code according to the specifications required by the defined interfaces.
4. The implementation in the new Unified Solids Library should consist of optimized code and should offer better or at least comparable performance as offered by the existing implementations in Geant4 or ROOT.
5. The implementation of each geometrical primitive must be done in parallel with comprehensive testing for the validity of results and performance. The code produced must be maintainable, efficient, readable, compact and portable.
6. Make use wherever possible of tools and data structures provided by the Standard C++ library.
7. The new library should not depend on any external package, but be realized as a standalone package.
8. The implementation should take into account possibilities of vectorizing the code where possible.
3.3. TESTING
Tests and verification of the new code will be performed through a comprehensive testing suite, which will be executed in parallel with the development of each algorithm and function for each shape. The testing suite will be used for the validation of the code, to guarantee correctness under any possible condition and the best optimization for CPU performance.

Tests include also the development of dedicated performance and numerical value comparison tools to help with the analysis of the various classes and methods for the identification of hot spots and areas of improvement.

Specialized tests for verifying scalability of the implemented algorithms in function of the complexity of the geometrical setup for selected solids will also be implemented and adopted in the testing suite.

Some of the tests are described in the following sections.

3.3.1. Solid Batch Test (SBT)
The Solid Batch Test (SBT) application is an extendible framework for performing batch tests to solids in Geant4; the test is used as part of the existing testing suite for the Geant4 geometry modeller. It contains several geometry tests applicable to each geometrical primitive, based on the generation of random points and voxelization technique. The use of the bridge classes allows for the integration of the code in the new library to make use of the SBT suite; similarly, cross-comparisons are possible with Geant4 and ROOT implementations.

3.3.2. Data Analysis and Performance (DAP)
Extensions to the SBT suite are foreseen, in order to realize a comprehensive Data Analysis and Performance (DAP) platform for testing the results of each algorithm at numerical level. DAP will be centred on testing each element in the Unified Solids Library (USolids) in direct comparison with Geant4 and ROOT implementations. The final testing suite will allow for automatic verification of correctness and performance results and will represent the core part of the testing suite for the USolids Library. The approach used for the testing is divided in two phases: one simulation phase using pre-calculated, randomly generated sets of points and direction vectors, and one analysis phase creating 3D plots allowing analysis of all or parts of the data sets produced in the simulation phase, including the visualization of the shape under investigation, see e.g. Figure 11.
3.3.3. Optical Escape

An additional test, derived from the Geant4 geometry modeller testing-suite is the Optical Escape test. The test allows applying real Monte Carlo simulation of optical photons using as detector a specific geometrical shape to be considered for testing. The adoption of the bridge classes again allows for performing cross-comparisons of the various implementations. Optical photons are generated and reflected on the inner surface of the solid, with the aim to easily identify error conditions of tracks exiting the solid. We plan to use Optical Escape (as eventually other tests employed in Geant4 and ROOT and not mentioned in this document) as a supplemental test complementing the DAP suite. See Error! Reference source not found. for a visual output from the Optical Escape application.
4. CONCLUSION

The design of the Detector Description Toolkit and the Unified Solids Library described in the previous sections has already been prototyped for some of the main key aspects. This prototype has served the purpose of testing some of the design ideas and implementation choices. The code is available the AIDA WP2 SVN repository. Initial documentation and links to the code repositories can be found in dedicated web pages in for the Geometry Toolkit [14] and for the Unified Solids Library [15]. The current working prototype will evolve to a complete developed solution as part of the WP2 task 2.2 and will constitute the final deliverable.

The software development will follow an iterative process with periodic releases and offering incremental functionality in each iteration. It is important for the success of the project as well as for the quality of the final product to identify very early in the development phase possible clients willing to use the new software packages in order to provide very valuable feedback during the development phase. We plan to achieve this by using the toolkit in the other task and subtasks of WP2 as well as in other work packages of the AIDA project.
5. REFERENCES


