AIDA 2020

Advanced European Infrastructures for Detectors at Accelerators

DDAlign

Alignment Support for the DD4hep Geometry Description Toolkit

M. Frank

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

CERN, 1211 Geneva 23, Switzerland

Abstract

Experimental setups in High Energy Physics are highly complex assemblies consisting of various detector devices typically called *subdetectors*. Contrary to the ideal world, where all these components are of perfect shape and at exact positions, existing devices have imperfections both in their shape and their relative and absolute positions. These are described by the alignment parameters.
To still measure the detector response from particle collisions with the highest possible precision, these imperfections are taken into account when converting measured signals to space-points in the measurement devices. This procedure is called *detector alignment*. `DD4hep` does not want to solve the exact problem of the detector alignment itself, but rather support firstly algorithms determining the alignment parameters and secondly support the application which apply the measured alignment parameters and apply them to the ideal geometry for further event data processing.
We will present the tools to support the detector alignment procedures using the `DD4hep` detector description toolkit. The `DDAlign` toolkit implements a modular and flexible approach to introduce and access the alignment parameters.
The design is strongly driven by easy of use; developers of detector descriptions and applications using them should provide minimal information and minimal specific code to achieve the desired result.

Document History

Document version	Date	Author
1.0	01/04/2014	Markus Frank CERN/LHCb
1.1	30/04/2014	Markus Frank CERN/LHCb
1.2	28/02/2017	Markus Frank CERN/LHCb

Contents

1 Introduction

This manual should introduce to the `DDAlign` framework. One goal of `DDAlign` is to easily model geometrical imperfections applied to the ideal geometry of detection devices as they are typically used in high energy physics experiments.
To avoid confusion within this document, a few terms need to be defined with respect to detector alignment:

- The *ideal geometry* describes the detector as it was designed. Such a detector is an utopia, which can never be realized in terms of the placement of the individual components as such.
- The *actual geometry* describes - as a first approximation to the real world - the real detector at a given time valid for a rather significant amount of time e.g. for a year of data taking. The *actual geometry* typically includes corrections deduced e.g. from optical surveys etc. The *actual geometry* does not change during the life-time of an analysis or calibration process. In the following this is called *Global Alignment*. The transformation of the ideal geometry to the actual geometry is steered by alignment parameters aka *Alignment Deltas*. Such *deltas* may be applied at any level of the geometrical hierarchy. In short, the *actual geometry* results from the *ideal geometry* after applying the global *Alignment Deltas* and is then the *geometry in memory*. The ROOT geometry toolkit is the only one, which allows for global alignment procedures [1].
- *Realignment* then defines the procedures to correct data collected in particle collisions. These data are taken with the real, a priori unknown geometry, which on top of the actual geometry suffers from small shifts e.g. due to temperature or pressure changes. These shifts normally are frequently computed by specialized applications with respect to the to the actual geometry and typically are valid for relatively short time periods O(1 hour). These shifts, called *Alignment Deltas*, are used to re-align the detector response for physics analysis. This process in the following is called *Local Alignment*. The handling of the *Alignment Deltas* for local alignments in fact is very similar to the handling of detector conditions implemented in the package `DDCond` [7]. In section 4 this issue is further elaborated.

Technically the *Alignment Deltas* used for the global alignment and the *Alignment Deltas* used for the local alignment are identical. Though it should be stressed that the use is entirely different: Whereas the first actually alter the geometry, the latter are only used to properly interpret the data collected.
`DDAlign` formalizes both the access and the application of alignment parameters to the ideal geometry. The possibility to properly describe actual geometries with respect to ideal geometries is essential to understand the detector response to particle collisions and to connect response of geometrical independent areas of the experiment e.g. to one single track.
In this manual we will shortly describe the model used to describe an experiments detector description and then in more detail document the support for alignment with its programming interfaces.

1.1 Generic Detector Description Model

This is the heart of the DD4hep 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. The design reuses the elements from the ROOT geometry package and extends them in case required functionality is not available. Figure 1 illustrates the main players and their relationships [1]. Any detector is modeled as a tree of *Detector Elements*, the entity central to this design, which is represented in the implementation by the *DetElement* class [2]. It offers all applications a natural entry point to any detector part of the experiment and represents a complete sub-detector (e.g. TPC), a part of a sub-detector (e.g. TPC-Endcap), a detector module or any other convenient detector device. The main purpose is to give access to the data associated 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

[1] A conversion of this geometry e.g. to Geant4 (using the functionality provided by `DDG4` allow to simulate distorted geometries with the Geant4 toolkit.

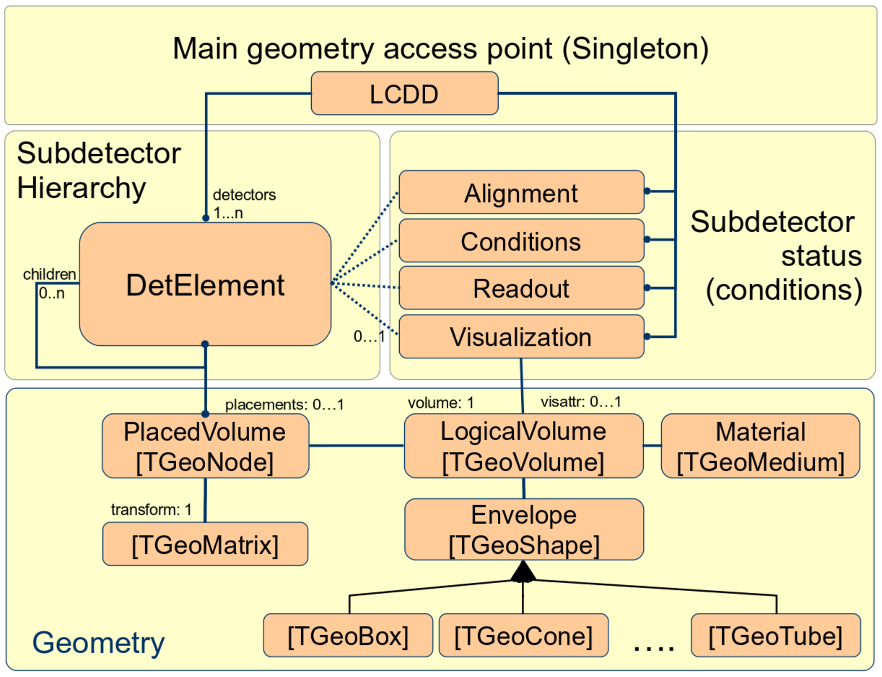

Figure 1: Class diagram with the main classes and their relations for the Generic Detector Description Model. The implementing ROOT classes are shown in brackets.

conditions such as the gas pressure, end-plate temperatures etc. The *Detector Element* acts as a data concentrator. Applications may access the full experiment geometry and all connected data through a singleton object of type *Detector*, which provides management, bookkeeping and ownership to the model instances.
The geometry is implemented using the ROOT geometry classes, which are used directly without unnecessary interfaces to isolate the end-user from the actual ROOT based implementation. `DDAlign` allows client to access, manage and apply alignment parameters or smallish changes to the ideal geometry. The mechanism to achieve this is described in the following.

1.2 Detector Element Tree and 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. A *Logical Volume* is a shape with its dimensions and consist of a given material. They represent unpositioned objects which store all information about the placement of possibly embedded volumes. The same volume can be replicated several times in the geometry. The *Logical Volume* also represents a system of reference with respect to its containing volumes. The reuse of instances of *Logical Volumes* for different placements optimizes the memory consumption and detailed geometries for complex setups consisting of millions of volumes may be realized with reasonable amount of memory. The difficulty is to identify a given positioned volume in space and e.g. apply alignment parameters to one of these volumes. The relationship between the Detector Element and the placements is not defined by a single reference to the placement, but the full path from the top of the detector geometry model to resolve existing ambiguities due to the reuse of *Logical Volumes*. Hence, individual volumes must be identified by their full path from mother to daughter starting from the top-level volume.
The tree structure of *Detector Elements* is a parallel structure to the geometrical hierarchy. This structure will probably not be as deep as the geometrical one since there would not need to associate

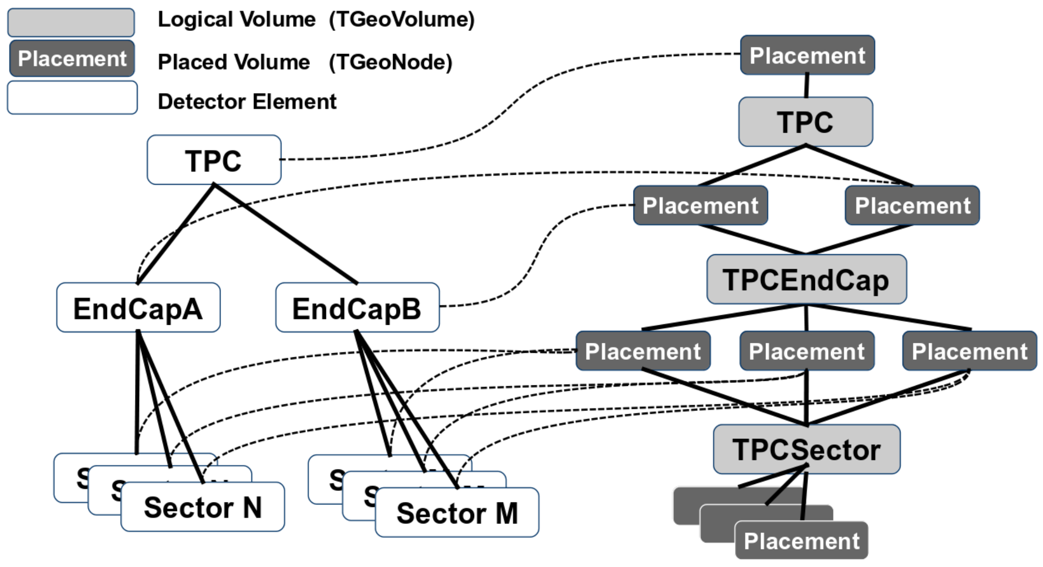

Figure 2: The object diagram of a hypothetical TPC detector showing in parallel the *Detector Element* and the *Geometry* hierarchy and the relationships between the objects.

detector information at very fine-grain level - it is unlikely that every little metallic screw needs associated detector information such as alignment, conditions, etc. Though this screw and many other replicas must be described in the geometry description since it may be important e.g. for its material contribution in the simulation application. Thus, the tree of Detector Elements is fully degenerate and each detector element object will be placed only once in the detector element tree as illustrated for a hypothetical Time Projection Chamber (TPC) detector in Figure 2 with an ideal geometry, where no positioning corrections are applied to neither child. It is essential to realize that the geometry tree in an ideal geometry is degenerate contrary to the tree of detector elements.
It should be noted, that alignment parameters may be applied to any volume of the ideal geometry. The alignment only affects the actual position of a volume it is e.g. irrelevant if the volume is sensitive or not.

2 Global Alignment

2.1 Global Alignment of Detector Components

In this section the backgrounds of the *Global Alignment* is described. Alignment parameters never apply in the same way to *all* placements of the same volume in this hierarchy. Hence, to (re-)align a volume in the hierarchy means to logically lift a full branch of placements from the top volume down to the element to be (re-)aligned out of this shared hierarchy and apply a correction matrix to the last node. This procedure is illustrated in Figure 5. Re-alignment of volumes may occur at any level. In the above example of a TPC this results in the following effects:

- A realignment of the entire subdetector, i.e. the TPC as a whole, would affect consequently move all contained children with respect to the top level coordinate system. An example is shown in Figure 5 (a). A movement of the subdetector would affect all transformation between local coordinates of any part of the subdetector to the top level coordinate system. Such effects would be visible at all stages of the data processing e.g. when translating signals from particles into global coordinates.
- A realignment of parts of a subdetector affects only the partial subdetector itself and child volumes at lower levels. As in the example, where the entire subdetector is moved, here only the sectors

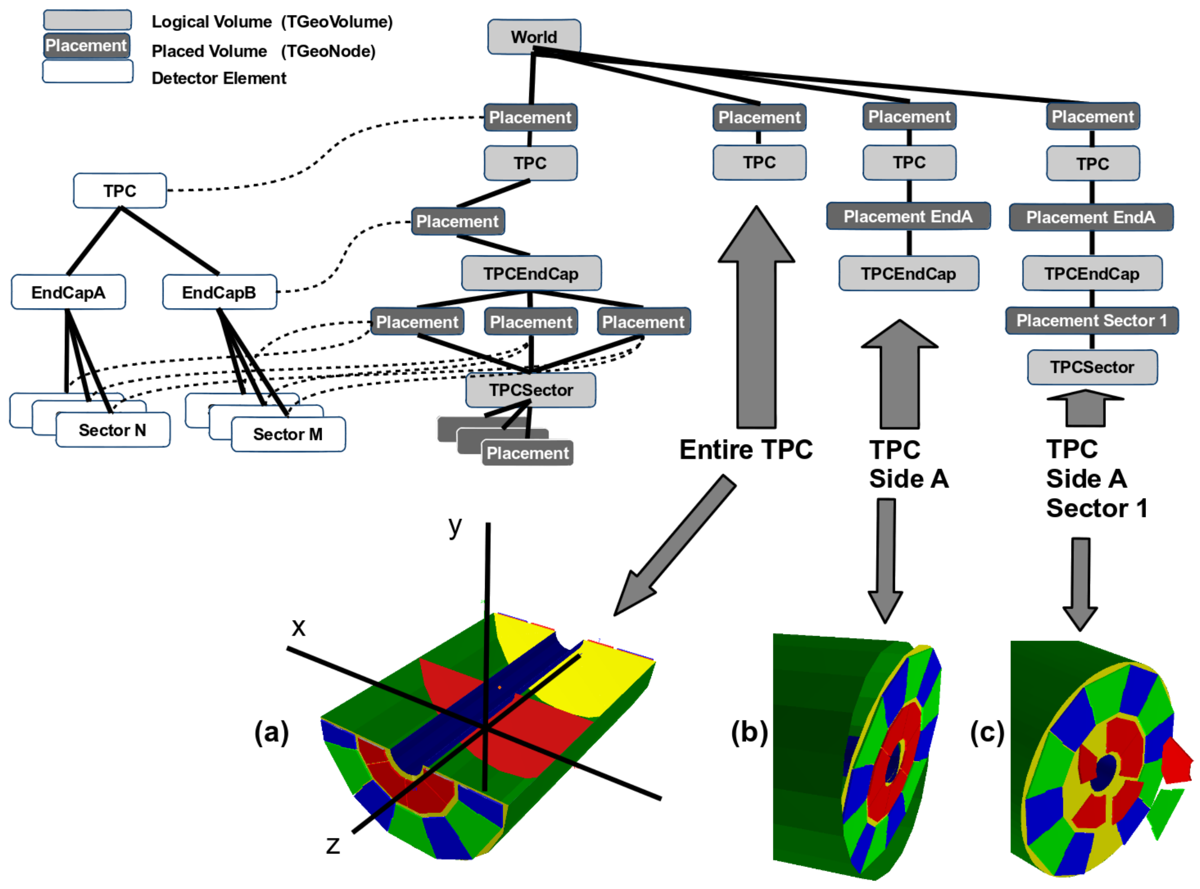

Figure 3: The object diagram of a hypothetical TPC detector showing in parallel the *Detector Element* and the *Geometry* hierarchy and examples of mispositioned detector parts: (a) mispositioned entire subdetector (translation), (b) mispositioned end-cap (tilt) and (c) mispositioned individual sectors within one endcap.

on one side of the TPC would be affected as shown in Figure 5 (b).

- In Figure 5 (c) within one end-cap of the TPC individual sectors may not be positioned at the ideal location (Figure 5 (c) exaggerates: "flying sectors" are a rather rare case in reality). Finally also the sectors itself could be fragmented and be assemblies of other shapes, which are not ideally placed and may need correction.

The origin of the volume misplacements may be many-fold:

- Elements may be weak and assembled parts move due to weak support structures. This is a common problem e.g. for tracking detectors, where heavy and solid structures dramatically influence the measurement result. Misplaced sectors could e.g. be the consequence of a deforming end-cap frame due to the weight of the sectors.
- Environmental conditions such as the temperature may influence the position or the shape of a volume.
- Some of the measurement equipment may be moved from a parking position into a data taking position such as the two halves of the LHCb vertex detector. Whereas the position of the sensors on each half are known to a very high precision, the position of the absolute position of the two halves with respect to the full experiment may change after each movement.

Changes to the volume placement do not only affect sensitive material i.e. detector components with an active readout, but also passive material. The placement of any volume, passive or active, may be corrected using `DDAlign` . The determination of the alignment parameters of passive components however may be more difficult in the absence of located signals resulting e.g. from the traversal of a track.
All effects resulting from such causes obviously need to be corrected in order to fully explore the capabilities of the detection devices and to minimize measurement errors. In general any deviation from the ideal position of a volume can be described by two elementary transformations:

- a translation
- a rotation around a pivot point.

giving a full transformation matrix of the form:

$$T = L * P * R * P^{-1} \tag{1}$$

where

- T is the full transformation in 3D space containing the change to the exiting placement transformation. The existing placement is the placement transformation of the volume with respect to the mother volume.
- L is a translation specifying the position change with respect to the mother volume.
- $P * R * P^{-1}$ describes a rotation around a pivot point specified int he mother volume's coordinate system.
- P is the translation vector from the mother volumes origin to the pivot point. The concept of a pivot point does not introduce a new set of parameters. Pivot points only help to increase the numerical precision.

Most of the changes do not require the full set of parameters. Very often the changes only require the application of only a translation, only a rotation or both with a pivot point in the origin. These simplifications are supported in the user interface described in Section 5.

2.2 Iterative Application of Global Alignments

Technically it is possible to apply global alignment procedures iteratively. This however id **deprecated** and violates thread safety for the simple reason that the *geometry in memory* is altered. If applied, it is duty of the client framework to ensure that during the change of global alignment no processing of event data is ongoing. Hence, the procedure is described here only for completeness:

1. Create the ideal detector using an ideal geometry.
2. Apply a set of alignment parameters for a given time interval corresponding to the time a set of particle collisions were collected in the experiment.
3. Process the set of collected particle collisions.
4. Reset the misaligned detector to the ideal.
5. Choose new event data input corresponding to another time interval and restart at item 2.

Graphically this use case is illustrated in Figure 4. In Section 5 the implementation to realize this use case is described.

2.3 Procedures to Determine Global Alignment Parameters

Typically the determination of alignment parameters requires a starting point which is not necessarily identical to the ideal position of a volume [3]. These volume positions are the result of a survey measurement or the result of internal position measurements of a sub-volume within a sub-detector e.g. on a measurement bench. In the following we call these parameters *survey parameters*. *Survey*

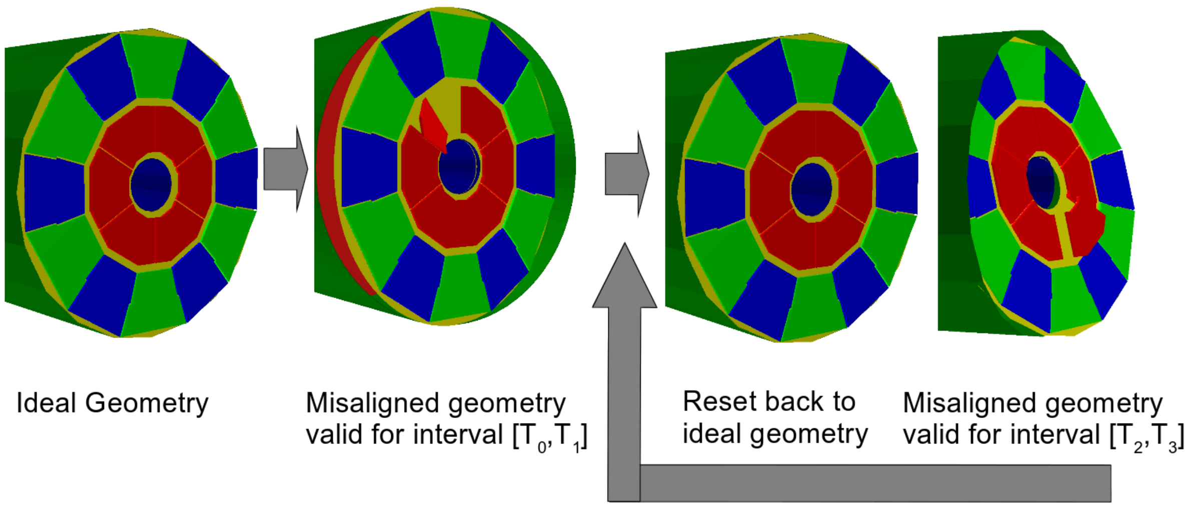
Ideal Geometry
Misaligned geometry valid for interval [T$_0$,T$_1$]
Reset back to ideal geometry
Misaligned geometry valid for interval [T$_2$,T$_3$]

Figure 4: The iterative application of alignment parameters as described in Section 2.2. For each interval of validity ($[T_0, T_1]$, $[T_2, T_3]$, $[T_4, T_5]$, ...) a seperate set of alignment constants is applied to the ideal geometry. The two steps to reset the misaligned geometry back to the ideal geometry and to re-apply a new set of alignment constants may be executed as often as necessary when processing data from particle collisions.

parameters default to the ideal volume position if not supplied, alternatively, if set, to the provided position. *Survey parameters* are, like the alignment parameters, provided in terms of *changes* with respect to the ideal position and hence may be treated in a similar way.
The survey parameters are accessible to users through the interface offered by the *DetElement* objects.

2.4 Simulation of Non-Ideal Detector Geometries

It is a standard procedure in high energy physics to at least verify the measured detector response of a given physics process in particle collisions with the expected simulated detector response. For most purposes the simulation of an ideal detector is certainly is sufficient - though not describing the full truth. Sometimes however, the detector geometry must be simulated with a geometry as close to the known geometry as possible.
The simulation of such a geometry with applied alignment parameters can rather easily be realized using using the `DD4hep` , `DDAlign` and the `DDG4` frameworks:

- The ideal geometry is constructed using the standard procedures of `DD4hep` [1].
- Then the alignment parameters are applied and finally
- the corrected geometry is translated to *Geant4* [6] using the `DDG4` [4] package. All particle collisions simulated with this translated geometry correspond to the modified geometry including the geometry modifications.

There is a caveat though: The application of alignment parameters can easily create volume overlaps, which are highly disliked by the *Geant4* runtime. If the above described procedure is applied, it is highly advised to check the resulting geometry for overlaps. Both, *ROOT* [5] and *Geant4* [6] offer tools to perform such tests.
To simulate distorted geometries clients should use the *Global Alignment* interface described in section 2.5.

2.5 The Global Alignment Interface

In this chapter will be documented how to use the *Global Alignment* interface of `DDAlign` . As already mentioned in section 1, this interface allows to alter the layout of the geometry in memory. Use cases are e.g. the simulation of non-ideal geometries.
Global alignment can be applied to detector elements using a specialized interface *GlobalDetectorAlignment* [2]. This interface provides the API to apply global changes to the geometry provided the presence of alignment parameters as shown in the following code snippet:

```
 1  /// First install the global alignment cache to build proper transactions
 2  Detector& detector = ...;
 3  GlobalAlignmentCache* cache = GlobalAlignmentCache::install(detector);
 4
 5  /// Now create the tranaction context. There may only be one context present
 6  GlobalAlignmentStack::create();
 7  GlobalAlignmentStack& stack = GlobalAlignmentStack::get();
 8
 9  /// Now we can push any number of global alignment entries to the stack:
10  DetElement         elt       = ...detector element containing the volume to be re-aligned ...;
11  string             placement = "/full/path/to/the/volume/to/be/realigned";
12  Alignments::Delta delta      = ...;
13  double             ovl       = allowed_overlap_in cm; // e.g. 0.001;
14
15  // Create the new stack entry and insert it to the stack
16  dd4hep_ptr<StackEntry> entry(new StackEntry(elt,placement,delta,ovl));
17  stack->insert(entry);
18
19  /// Finally we commit the stacked entries and release the stack.
20  cache->commit(stack);
21  GlobalAlignmentStack::get().release();
```

Explanation:

Line	
3	Install the *GlobalAlignmentCache*. Required to be done once. The object is registered to the *Detector* instance and kept there.
3-8	The fact that the classes *GlobalAlignmentCache* and *GlobalAlignmentStack* are singletons is not a fundamental issue. However, we want to call the XML parser (or other database sources) iteratively and currently cannot chain a context (stack).
16-21	The created stacked entries are automatically released once the transaction is committed.

Please note, that this interface normally is not directly invoked by users, but rather called by plugin mechanisms as the one described below capable of reading the global misalignments from XML.

2.5.1 Loading Global Geometrical Imperfections from XML

In this section we describe how to load global geometry imperfections and to apply them to an existing geometry. Loading the XML file is done automatically using the standard XML loader plugin provided by `DD4hep` . This mechanism is favoured and much simpler than programming the global misalignment directly. This plugin is interfaced to the `Detector` instance and invoked from code as follows:

```
  Detector& detector = ....;
  detector.fromXML("file:AlepTPC_alignment.xml");
```

To fully exploit the capabilities it is important to understand the interpreted structure of the XML file being processed. At the top level of the primary input file (i.e. the file given to the XML processor) the following structure is expected:

[2]See the header file *DDAlign/GlobalDetectorAlignment.h* for details.

```
<global_alignment>
  <!-- Open the alignment transaction  -->
  <open_transaction/>
  <subdetectors>         <!-- Container with the list of subdetectors to be processed. -->
    <detelement path="TPC" reset="true" reset_children="true">
      <!-- Move the entire TPC in the world volume                                     -->
      <position="" x="30"   y="30"  z="80"/>

      <!-- Now add daughter detector elements                                           -->

      <!-- Twist a bit the entire endcap by rotating it around the x and the y axis    -->
      <detelement path="/world/TPC/TPC_SideA" check_overlaps="false">
        <position x="0"   y="0"  z="0"/>
        <rotation x="-0.2" y="-0.2"  z="0"/>
        <!-- Apply corrections of type Translation*Rotation to a single sector
        <detelement path="TPC_SideA_sector02" check_overlaps="true">
          <position x="0"   y="0"   z="0"/>
          <rotation x="0.5" y="0.1" z="0.2"/>
        </detelement>
      </detelement>

      <!-- And the full shooting match of transformations for this sector              -->
      <detelement path="TPC_SideA/TPC_SideA_sector03" check_overlaps="true">
        <position x="0" y="0"    z="290.0*mm"/>
        <rotation x="0" y="pi/2" z="0"/>
        <pivot    x="0" y="0"    z="100"/>
      </detelement>

      ....

      <!-- Include alignment files to be processed in the context of the "TPC" DetElement
      <include ref="file-name"/>

    </detElement>
  </subdetectors>

  <!-- Include alignment files to be processed at the top level context               -->
  <include ref="file-name"/>

  <!-- Close the alignment transaction  -->
  <close_transaction/>
</global_alignment>
```

The structure of the alignment file explained quickly:

Line	
1	The `root` tag for the primary alignment file is `<alignment/>`. The primary tag name is mandatory and actually is used to invoke the correct interpreter.
2,41	Trigger the alignment transaction by specifying the transaction tags in the main XML file.
4	Defintion of the set of `subdetectors` to be processed. A valid alias for this directove is `detelements`.
5	The first subdetector: `TPC`. The subdetector tag is `detelement` Each `detelement` may recursively contain other `detelement` tags. as they were defined in the `DetElement` hierarchy. Internal `detelement` elements are processed in the context of the outer element i.e. pathes may be specified relative to the parent or as absolute pathes with respect to the world (starting with a '/').
7	Global movement of the TPC
12-20	Realignment entry for the TPC endcap A named `TPC_SideA`
16-19	Realignment entry for sector named `TPC_SideA_sector02` of the TPC endcap A. Here the sector is specified directly as a daughter of the endcap. The name of the `DetElement` is relative to the parent.
23-27	Realignment entry for sector named `TPC_SideA_sector03` of the TPC endcap A containing a full transformation: $Translation * Pivot * Rotation * Pivot^{-1}$
32	Optionally `detelement` elements may include other alignment files specifying lower volume levels. These files are interpreted in the context of the calling detector element.
38	Optionally the subdetector alignment constants may be fragmented into several files, which can be loaded using the `include` directive. Each file could for example describe one single detector.

The specification of any transformation element is optional:

- The absence of a translation implies the origin (0,0,0)
- The absence of a pivot point implies the origin (0,0,0)
- The absence of a rotation implies the identity rotation. Any supplied pivot point in this case is ignored.

The absence of a transformation element is absolutely legal and does not issue any warning or error.
All transformations describe the change of placement with respect to the coordinate system of the closest mother-volume in the volume hierarchy, i.e. translations, rotations and pivot points are local to the mother coordinate system.
Included files may directly start with the `root` tags `subdetectors`, `detelements` or `detelement` and may recursively include other files. Except for the top level these files are processed in the calling context. The result of this procedure is shown in Figure 5.

2.5.2 Export Geometrical Imperfections to XML

In this section we describe how to export geometry imperfections to an XML file. A small helper class `AlignmentWriter` achieves this task as shown in the snippet:

```
Detector&  detector = ....;
DetElement top = detector.world();
if ( top.isValid() )   {
  AlignmentWriter wr(detector);
  return wr.write(top,output,enable\_transactions);
}
```

This code will dump all alignment constants contained in the `DetElement` hierarchy of `top` to the output file `output`. The optional argument `enable_transactions` (default: true) will add the tags `<open_transaction/>` and `<close_transaction/>` to the output file. The output file conforms to the specifications described in Section **??** and may later be imported by another process.
FIXME: This chapter sort of still has to be written/completed!!!!

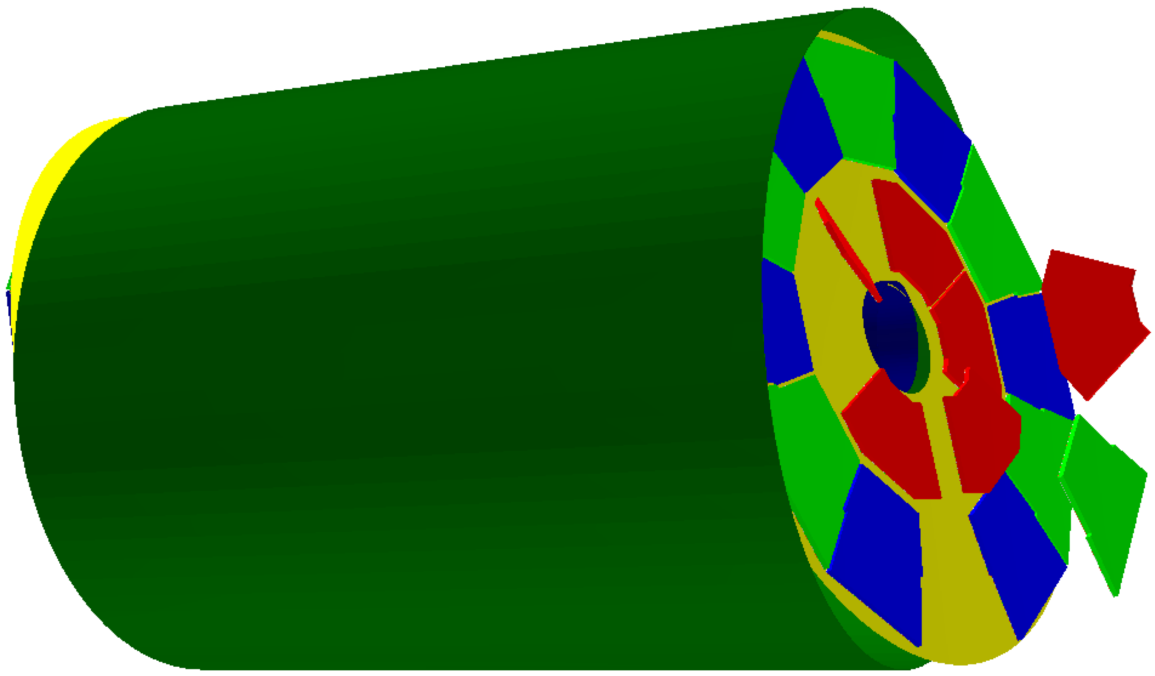

Figure 5: The ALEPH TPC after the import of the alignment file. Note, that the geometry in *memory* changed. The original geometry desciption is no longer present.

3 Up to here the manual should be pretty much correct. Everything below is at least questionable.

4 Local Alignment

bla bla bla
Generally such a behavior can be achieved in two ways. The usage strongly depends on the use-case required by the client:

1. either the ideal geometry in memory is changed directly to reflect the measured geometry. This approach has the disadvantage, that all measurement points on a daughter volume can only be transformed to the global coordinate system using one single transformation. Time-dependent changes of these transformations cannot be modeled. Hence, for multi-threaded systems this approach is of limited use. However, this is the perfect approach to simulate distorted geometries. This approach is naturally supported by the ROOT geometry toolkit.

2. The second possibility is to not modify the ideal geometry in memory, but to provide instead transformations to move measured coordinates to their correct place in space. This approach allows to keep several - also time-dependent - transformations in memory. Ideal to support multi-threaded data processing frameworks, which become more and more popular.

`DDAlign` supports both possibilities as will be described in the following sections.

5 The Local Alignment Interface

DDAlign implements a machinery to apply and access the alignment parameters describing the difference between an ideal detector given by an ideal geometry and the geometry of the actually built assembly in real life. To ease its usage for the clients and to shield clients from the internals when actually dealing with realigned geometries, a set of helper classes was designed. The access to the alignment parameters in read-only mode was separated from the import or export thereof.
As a basic concept within DD4hep any *sizable* detector component can be realigned. *Sizable* as a rule of thumb is anything, which is manufactured as an individual piece and which you may "hold in your hands". Such objects are also described by a *detector element* of type DetElement. An example is e.g. a single silicon wafer of a tracking device or the entire tracking detector itself. The access to the alignment parameters is possible from each DetElement instance as described in Section 5.1. The interface assumes "planar" alignment parameters i.e. the shape of a given volume does not change [3].
As mentioned earlier, in the local alignment DDAlign allowed to retrieve time dependent alignment parameters and transformations. This time dependency was relatively easy achieved by re-using the conditions mechanism from DDCond . In this spirit Alignment transformations are practically no different from conditions like temperatures, pressures etc. To access the *alignment conditions* clearly not only some identifier must be provided, but also a *interval of validity*, which defines from which point in the past to which point in the future the required alignment constants may be applied.
Please be aware that the extensive use of misalignments is highly memory consuming.

5.1 Access to Alignment Parameters from the Detector Element

The *DetAlign* class as shown in Figure 1 gives the user access to the alignment structure of type *Alignment* as illustrated in the following example:

```
ConditionsSlice slice = ...  // Prepared slice containing all condiitons
DetElement wafer_det  = ...  // Valid handle to a detector element
DetAlign   wafer = wafer_det;
Alignment  wafer_alignment = wafer.get();
if ( wafer_alignment.isValid() )  {
    // This wafer's placement differs from the ideal geometry when
    // alignment parameters are present.

    // Access the misalignment transformation with respect to the parent volume:
    Transform3D tr = wafer_alignment.toMotherDelta();
}
```

The access to details of an invalid alignment object results in a runtime exception. The following calls allow clients to access alignment information from the *DetElement* structure:

```
  /// Access to the actual alignment information
  Alignment alignment() const;

  /// Access to the survey alignment information
  Alignment surveyAlignment() const;
```

The call to *alignment*() return the parameters *applied* to the the existing ideal geometry. The call *surveyAlignment*() returns optional constants used to perform numerical calculations as described in section 2.3.
All functionality of the DetElement, which depends on applied alignment parameters are automatically updated in the event of changes. These are typically the geometry transformations with respect to the mother- and the world volume:

[3]This is a restriction to the possibilities provided by the ROOT implemetation [5] based on experience [3]. If at a later time the need arises the provided alignment interface may be extended to support shape changes.

```
    /// Create cached matrix to transform to world coordinates
    const TGeoHMatrix& worldTransformation() const;

    /// Create cached matrix to transform to parent coordinates
    const TGeoHMatrix& parentTransformation() const;

    /// Transformation from local coordinates of the placed volume to the world system
    bool localToWorld(const Position& local, Position& global) const;

    /// Transformation from local coordinates of the placed volume to the parent system
    bool localToParent(const Position& local, Position& parent) const;

    /// Transformation from world coordinates of the local placed volume coordinates
    bool worldToLocal(const Position& global, Position& local) const;

    /// Transformation from world coordinates of the local placed volume coordinates
    bool parentToLocal(const Position& parent, Position& local) const;
```

it is worth noting that the update of cached information is performed by the *DetElement* objects, other user defined cached information is **not** updated. To update user caches it is mandatory to provide a user defined update callback to the *DetElement*:

```
  template <typename Q, typename T>
  void callAtUpdate(unsigned int type, Q* pointer,
                    void (T::*pmf)(unsigned long typ, DetElement& det, void* opt_par)) const;
```

The interface of the *Alignment* structure to access detector alignment parameters is as follows (see also the corresponding header file DD4hep/Alignment.h):

```
    /// Number of nodes in this branch (=depth of the placement hierarchy from the top level volume)
    int numNodes() const;

    /// Access the placement of this node
    PlacedVolume placement()   const;

    /// Access the placement of the mother of this node
    PlacedVolume motherPlacement(int level_up = 1)   const;

    /// Access the placement of a node in the chain of placements for this branch
    PlacedVolume nodePlacement(int level=-1)   const;

    /// Access the currently applied alignment/placement matrix with respect to the world
    Transform3D toGlobal(int level=-1) const;

    /// Transform a point from local coordinates of a given level to global coordinates
    Position toGlobal(const Position& localPoint, int level=-1) const;

    /// Transform a point from global coordinates to local coordinates of a given level
    Position globalToLocal(const Position& globalPoint, int level=-1) const;

    /// Access the currently applied alignment/placement matrix with respect to mother volume
    Transform3D toMother(int level=-1) const;

    /// Access the currently applied alignment/placement matrix (mother to daughter)
    Transform3D nominal() const;

    /// Access the currently applied correction matrix (delta) (mother to daughter)
    Transform3D delta() const;
```

```
30
31      /// Access the inverse of the currently applied correction matrix (delta) (mother to daughter)
32      Transform3D invDelta() const;
```

- The calls in line 3-8 allow access to the relative position of the *nth.* element in the alignment stack with respect to its next level parent. Element $numNodes() - 1$ denotes the lowest level and element 0 is the world volume. The default argument (-1) addresses the lowest placement in the hierarchy.
- Calls in line 9-12 allow to access/execute transformations from a given level in the placement hierarchy to coordinates in the top level volume (world).
- The call in line 14 allows to transform a global coordinate to the local coordinate system in a given level of the hierarchy.
- The call $toMother$ in line 16 returns the local transformation of the node at a given level to the mother's coordinate system.
- The calls in line 17-20 give access to the nominal placement matrix of the realigned node with respect to the parent volume and the changes thereof.

Besides these convenience calls the full interface to the class `TGeoPhysicalNode`, which implements in the ROOT geometry package alignment changes, is accessible from the *Alignment* handle using the overloaded $operator->()$. Further documentation is available directly from the ROOT site .

5.2 Manipulation of Alignment Parameters

There are multiple possibilities to apply alignment parameters:

- The pedestrian way "by hand" using C++ as described in Subsection 5.2.1
- Loading a whole set of misalignment constants from XML, the "poor man's" database. This mechanism is described in Subsection **??**
- Loading a whole set of misalignment constants from a database. This possibility depends heavily on the database and its schema used. A typical use case is to load misalignment constants depending on the experiment conditions at the time the event data were collected. `DDAlign` does not provide an implementation. This possibility here is only mentioned for completeness and will be subject to further developments to support conditions in `DD4hep` .

5.2.1 Manipulation of Alignment Parameters for Pedestrians using C++

In this section we describe how to apply geometry imperfections to an existing detector geometry in memory using `C++`. To apply misalignment to an existing geometry two classes are collaborating, the `AlignmentCache` attached to the geometry container `Detector` and a temporary structure the `AlignmentStack`. The `AlignmentCache` allows to access all existing alignment entries based on their subdetector. The `AlignmentStack` may exist in exactly one instance and is used to insert a consistent set of alignment entries. Consistency is important because changes may occur at any hierarchical level and internal transformation caches of the ROOT geometry package must be revalidated for all branches containing a higher level node. **For this reason it is highly advisable to apply realignment constants for a complete subdetector.** Note that this restriction is not imposed, in principle a consistent set of misalignments may be applied at any level of the geometry hierarchy.
Though the application of alignment is much simpler using XML files, the following description should give an insight on the mechanisms used behind the scene and to understand the concept.
Any manipulations are transaction based must be embraced by the following two calls opening and closing a transaction:

```
// Required include file(s)
#include "DDAlign/AlignmentCache.h"
```

```

  Detector& detector = ....;
  AlignmentCache* cache = detector.extension<Geometry::AlignmentCache>();

  // First things first: open the transaction.
  cache->openTransaction();

  // Prepare the entry containing the alignment data
  AlignmentStack::StackEntry* entry =  .....;
  //.... and add the element to the AlignmentStack .....
  AlignmentStack::insert(entry);

  // Finally close the transaction. At this moment the changes are applied.
  cache->closeTransaction();
```

In the following we describe the mechanism to create and prepare the `StackEntry` instances of the above code snippet. The calls to open and close the alignment transaction do not have to be in the same code fragment where also the alignment entries are prepared. However, all changes are only applied when the transaction is closed. The alignment entries do not necessarily have to be prepared in the sequence of the hierarchy they should be applied, internally the entries are re-ordered and follow the geometry hierarchy top to bottom i.e. mother volumes are always re-aligned **before** the daughters are re-aligned.
The `StackEntry` instances carry all information to apply the re-alignment of a given volume. This information contains:

- The transformation matrix describing the positional change of the volume with respect to its mother volume.
- The placement path of the volume to be realigned.
- A flag to reset the volume to its ideal position **before** the change is applied.
- A flag to also reset all daughter volumes to their ideal position **before** the change is applied.
- A flag to check for overlaps after the application of the change and
- the actual precision used to perform this check.

The `ROOT::Math` library provides several ways to construct the required 3D transformation as described in Section 2.1:

```
// Required include file(s)
#include "DD4hep/Objects.h"

  Position      trans(x_translation, y_translation, z_translation);
  RotationZYX   rot  (z_angle, y_angle, x_angle);
  Translation3D pivot(x_pivot, y_pivot, z_pivot);

  Transform3D trafo;
  /// Construct a 3D transformation for a translation and a rotation around a pivot point:
  trafo = Transform3D(Translation3D(trans)*pivot*rot*(pivot.Inverse()));

  /// Construct a 3D transformation for a translation and a rotation around the origin
  trafo = Transform3D(rot,pos);

  /// Construct a 3D transformation for a rotation around a pivot point
  trafo = Transform3D(piv*rot*(piv.Inverse()));

  /// Construct a 3D transformation for a rotation around the origin
  trafo = Transform3D(rot);

  /// Construct a 3D transformation for simple translation
```

```
    trafo = Transform3D(pos);

```

The following code snippet shows how to extract this information from the `DetElement` and prepare such a `StackEntry` instance:

```
// Required include file(s)
#include "DDAlign/AlignmentStack.h"

    // Prepare the entry containing the alignment data
    typedef AlignmentStack::StackEntry Entry;
    /// Detector element to be realigned
    DetElement element = ...;
    /// The transformation describing the relative change with respect to the mother volume
    Transform3D trafo = ...;
    /// Instantiate a new alignment entry
    Entry* entry = new Entry(element);
    entry->setTransformation(trafo)                         // Apply the transformation matrix
        .applyReset(/* argument default: true */)           // Set the reset flag
        .applyResetChildren(/* argument default: true */)   // Set the daughter reset flag
        .checkOverlaps(/* argument default: true */)        // Set flag to check overlaps
        .overlapPrecision(0.001/mm);                        // With this precision in mm

    /// Now add the entry to the alignment stack:
    AlignmentStack::insert(entry);
```

The constructor will automatically determine the volumes placement path from the `DetElement`. Then the transformation is applied and the flags to reset the volume, its children and to trigger the overlap checks with the given precision.
When passing the entry to the `AlignmentStack` the `AlignmentStack` takes ownership and subsequently the entry is deleted after being applied to the geometry. For further shortcuts in the calling sequence please consult the `AlignmentStack` header file.

References

[1] M. Frank et al, "DD4hep: A Detector Description Toolkit for High Energy Physics Experiments", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2013), Amsterdam, Netherlands, 2013, proceedings.

[2] S. Ponce et al., "Detector Description Framework in LHCb", International Conference on Computing in High Energy and Nuclear Physics (CHEP 2003), La Jolla, CA, 2003, proceedings.

[3] C. Parkes, private communications.

[4] M.Frank, "DDG4 - A Simulation Toolkit for High Energy Physics Experiments using Geant4 and the DD4hep Geometry Description".

[5] R.Brun, A.Gheata, M.Gheata, "The ROOT geometry package", Nuclear Instruments and Methods **A** 502 (2003) 676-680.

[6] S. Agostinelli et al., "Geant4 - A Simulation Toolkit", Nuclear Instruments and Methods **A** 506 (2003) 250-303.

[7] M.Frank, "DDCond – Conditions Support for the DD4hep Geometry Description Toolkit".