DD4hep Status

HEP detector description supporting the full experiment life cycle

M. Frank, F. Gaede, M. Petric, A. Sailer
• Motivation and Goals

=> Introduction / Reminders

• Simulation

• Conditions support

• Alignments support

• Miscellaneous

• Summary
Motivation and Goal

- **Develop a detector description**
  - For the full experiment life cycle
    - detector concept development, optimization
    - detector construction and operation
    - “Anticipate the unforeseen”
  - Consistent description, with single source, which supports
    - simulation, reconstruction, analysis
  - Full description, including
    - Geometry, readout, alignment, calibration etc.
What is Detector Description?

- **Description of a tree-like hierarchy of “detector elements”**
  - Subdetectors or parts of subdetectors

- **Detector Element describes**
  - Geometry
  - Environmental conditions
  - Properties required to process event data
  - Optionally: experiment, sub-detector or activity specific data
Note:
DD4hep population is plugin based
=> Only one, not the exclusive way.
Saga in 5 Episodes: Sub-packages

- DD4hep – basics/core (1)
- DDG4 – Simulation using Geant4 (1)
- DDRec – Reconstruction supp. (2)
- DDAlign – Alignment support (3)
- DDCond – Detector conditions (3)

(1) Bug-fixes and maintenance
(2) See presentation of F. Gaede (WP3, Task 3.6)
(3) Work since start of AIDA 2020
● Motivation and Goals

  => Introduction / Reminders

● Simulation

● Conditions support

● Alignments support

● Miscellaneous

● Summary
DD4hep Core: Multiple Segmentations

Multiple Hit Collections

- Extension component using existing interfaces
- From the wish-list of FCC
- Collection selection according to 'key' and 'key value' or 'key range'

```xml
<readouts>
  <readout name="TestCalHits">
    <segmentation type="MultiSegmentation" key="layer"/>
    <segmentation name="Layer1grid" type="CartesianGridXY" key Min="0x1" key Max="4" grid Size X="0.1"/>
    <segmentation name="Layer2grid" type="CartesianGridXY" key Value="5" grid Size X="0.2"/>
    <segmentation name="Layer3grid" type="CartesianGridXY" key Min="0x6" key Max="0xFF" grid Size X="0.3"/>
  </segmentation>
  <hits_collections>
    <hits_collection name="TestCallInnerLayerHits" key="layer" key Value="0x1"/>
    <hits_collection name="TestCallMiddleLayerHits" key="layer" key Min="2" key Max="5"/>
    <hits_collection name="TestCallOuterLayerHits" key="layer" key Min="0x6" key Max="0xFF"/>
  </hits_collections>
  <id system="8", barrel="3", layer="8", slice="8", x:32=-16, y:16"/>
</readout>
</readouts>
```
Simulation: DDG4

- Simulation = Geometry + Detector response + Physics
- Mature status
  - Eventual bug fixes, smaller improvements
- Improvements
  - Support for multiple primary vertices from a single input source
    - Multiple input sources were already supported
- Full framework used by the Linear Collider community
- Individual components used by the FCC community
DDG4: Optimization

vtunes output from 20 events $e^+ e^- \rightarrow t\bar{t}$

- Nice example how a couple of stupid container look-ups can screw your day
- Now DDG4 framework overhead < 10% including: Input, hit handling in sensitive detectors, MC truth handling, output
- Motivation and Goals
- Simulation
- Conditions support
  => DDCond
- Alignment Support
- Miscellaneous
- Summary
DDCond: Conditions Data

• Time dependent data necessary to process the detector response [of particle collisions]

• Conditions data support means to Provide access to a consistent set of values according to a given time
  - Fuzzy definition of a “consistent set” typically referred to as “interval of validity”: time interval, run number, named period, ...
  - Configurable and extensible

• Data typically stored in a database
Conditions Data: Consistent Dataset

Production version:
- VDET: v3 for t<3, v2 for t3≤t <5, v3 for t5≤t <9, v1 for t ≥9
- HCAL: v1 for t<2, v2 for t2≤t <4, v1 for t ≥4
- RICH: v1 everywhere
- ECAL: v1 everywhere
DDCond: What do we want?

- We want to provide access to consistent set of accompanying data for processing event data
  - See previous slide
- We want to be “fastest”
  - Need reasonable users
- We want to support multi-threading at it’s best
  - Not wait for flushed event pipelines before updates
    - Fully transparent processing, minimal barriers
  - If we can do this, we can also expect some support from the experiment framework
- Reasonable use of resources
  - Cache where necessary but no more
DDCond: What can we assume?

(when used by reasonable users)

- **Conditions data are slowly changing**
  - e.g. every run O(1h), lumi section O(10min), etc.

- **Conditions data change in batches**
  - Interval of validity is same for a group (subdetectors)
  - Not every SD defines it himself (I know, needs discipline)

- **Conditions also are the result of computation(s)**
  - Conditions data may also be the combination of other conditions data applied to a functional object
  - Example: Alignment transformations from Delta-values
  - So-called “derived conditions” are mandatory
Yesterday and Today
Change of Paradigm

• Historically
  – C++ data processing frameworks were a novelty
  – Emphasis on flexibility, “discovery” of the data space
  – Only load what users ask for (Load-on-demand)
  – Multi-threading was no issue

• Today
  [no free lunch in life]
  – Load barriers and accessed conditions set is well specified
    [See for example ongoing discussions around Gaudi]
  – No late loading, no load on demand: minimize mutex-hell
  – Maybe a bit of overhead, but you gain by multi-threading
DDCond: The Data Cache

- ConditionsManager
- IOV type "fill"
- Door keeper
- Manage different data types
- IOV type "run"
- IOV type "lumi-section"
- IOV type "YEAR"

Data Stores:
- Organized by IOV
- Provide and manage data
- Cache for multiple configurations
- Relatively static
- Structure hidden

=> Only data cache
=> Once loaded data stay unless explicitly dropped

Hence may be replaced with alternative implementation
DDCond: User Data Access

1) creates ConditionsSlice
2) inserts Dependency Rules
3) inserts Load Addresses
4) requests load / computation (slice, run=127896)
5) Access consistent conditions

Condition pool managing each one unique IOV
-- ALL conditions in one pool have the identical IOV
-- IOV Type manage all pools with the same type:
  run, fill, epoch, year, ...

Run 127895
  Cond 1
  Cond 2
  Cond N

Run 127896
  Cond 1
  Cond 2
  Cond N

IOV type "run"

ConditionsManager

Conditions shared between multiple slices / user-pools

Slice management is up to client:
- Share slices between threads
- Populate new slices independently
DDCond: Flexibility where necessary

Plugin based concrete implementations:
- Experiment with different implementations
- Choose best implementation for the concrete use-case

1) creates ConditionsSlice
2) inserts Dependence
3) inserts Load Addresses
5) Access consistent conditions

4) requests load / computation (slice, run=127896)

ConditionsPool

UserPool

ConditionsManager

Slice management is up to client:
- Share slices between threads
- Populate new slices independently
DDCond: Framework Mode

Assigns proper slice to processing context according to event IOV

Experiment Framework
"Gaudi"

Thread 1
Read-only access. No locking
Slice 1
IOV Run=123 ... 128

Thread 2

Thread 3

Thread 4

... Thread N

Slice 2
IOV Run=129 ... 130

Slice M
IOV Run=KKKK

Create, populate and manage slices
As many non compatible slices as necessary
Drop unused slices early. Populate quickly from cache....
DDCond: Derived Conditions

- Data derived from conditions data are also conditions
  - Example: refractive index derived from atm. Pressure
  - Example: alignment transformations derived from $\Delta s$
  - Source may be one or multiple conditions
  - IOV is intersection of source IOVs

- Derived conditions depend on
  - Source condition(s)
  - Callback functor to perform the data transformation

- Derived condition dependencies must be registered to the projection slice
  - Computation is part of the “slice preparation”
/// Initialize the conditions manager and set plugins (here: defaults)
ConditionsManager condMgr = ConditionsManager::from(lcdd);
condMgr["PoolType"] = "DD4hep_ConditionsLinearPool";
condMgr["UserPoolType"] = "DD4hep_ConditionsMapUserPool";
condMgr["UpdatePoolType"] = "DD4hep_ConditionsLinearUpdatePool";
condMgr.initialize();

/// Register IOV type used to define IOV structures
const IOVType* iov_type_run = condMgr.registerIOVType(0,"run").second;

/// Create the conditions slice
ConditionsSlice* slice = new ConditionsSlice(condMgr);

/// Define slice content ........ (see next slide)

/// Now compute the conditions according to one IOV
IOV req_iov(iov_type_run,<specific value>);
/// Attach the proper set of conditions to the user pool
ConditionsManager::Result r = condMgr.prepare(req_iov,*slice);
/// Use the created conditions slice
ConditionsSlice* slice = ... 

/// Register required DATA condition using key:
ConditionKey key("Some-global-identifier");
slice->insert(key,LoadInfo("Persistent-location-where-to-find-data")); 

/// Register derived condition recipe:
/// – Depends on data from condition identified by "key": May be many!
/// – Uses "MYConditionUpdateCall" for the data transformation 
ConditionsUpdateCall* call = new MYConditionUpdateCall();
ConditionKey target_key("Some-other-global-identifier");
DependencyBuilder builder(target_key, call);
builder.add(key);  /// Derived condition depends on "key"
slice->insert(builder.release());
Conditions Access from the DetElement

- So far we defined the mechanism to manage conditions
- But we also need a friendly user interface for clients
  - This is all DD4hep is about
  - Hide details in the framework, expose simplicity to users
  - Framework may also mean “experiment framework”
    Expect a bit of support as long as real users are not affected

- Conditions are accessed by key from the detector elements in the hierarchy
  - Keys are encrypted from a user defined path (e.g. address)
  - Or an alias name such as “Alignment”, “Pressure” etc.
- Let’s move on to the code examples
Conditions Data: Dynamic Binding

• Any data may be bound to a condition object
  – If size < 64 bytes data aggregated in condition object
  – Otherwise from heap
  – May use boost::spirit grammar definitions

• Data access for both cases:

```cpp
/// Creator case: Create conditions object and bind the conditions data
Condition cond(name, type);
double& pressure = cond.bind<double>();
pressure = 981 * hPa;

/// Client case: access the conditions data using a projected slice
ConditionsSlice* slice = ...
ConditionsKey key(name);
Condition cond = slice->pool->get(key);
double& pressure = cond.get<double>();
```
// We need access to a projected slice
ConditionsSlice* slice = ...
DetElement detElement = ...

if ( detElement.hasConditions() ) {
    // Use specialized DetElement view (facade) to access conditions
    DetConditions dc(detElement);
    ConditionKey pressure("/world/TPC/EndA/Sector1#pressure");
    // Could also map "Pressure" to "/world/TPC/EndA/Sector1#pressure"
    // as alias to local DetElement namespace!

    // Access the condition by key from the container
    Conditions::Container container = dc.conditions();
    Condition cond = container.get(pressure.hash,*slice->pool);

    // Access conditions data using dynamic type binding
    const vector<int>& table = cond.get<vector<int> >();
    ...
}
Pros and Cons

- **Multiple slices: No global barriers on “change-run”**
  - ++ multi-threading, ++ advanced slice preparation
- **IOV-pools read-only after load + compute**
  - ++ no locking hell for event processors, only for the loader
- **No dependencies between IOV types (derived conditions)**
  - ++speed, ++simplicity --flexibility (use cases ?)
- **Many parallel IOV types are difficult to handle**
  - User problem: should limit yourself to 1,2 or 3
- **IOV pools must be reasonably populated**
  - -- 1 condition per pool would be bad. Many is efficient...
    - (→ need reasonable users)
Benchmarks and Timing (1)

- **CLICSiD example: ~ factor 5 beyond LHCb**
  - Standard CERN desktop 2 years old, Ubuntu 16.04 32 bit
    
    | Task Description                                                                 | Time (s)  |
    |----------------------------------------------------------------------------------|-----------|
    | Create 175 k conditions + registration to IOV type                               | ~ 0.22 s  |
    | Create and select slice for 175 conditions + 105 k computations                  | ~ 0.3 s   |
    | Subsequent select 280 k equivalent to run-change with already loaded conditions  | ~ 0.13 s  |
    | Slices for (175+105) for 20 runs (total of 5.8 Mcond)                            | ~ 0.22 s/run ~ 0.35 s/run |

- Looks quite scalable and quite fast
  - No database access nor XML parsing, but this was not part of this exercise
Benchmarks and Timing (2)

- **LHCb example**
  - Standard CERN desktop 2 years old, Ubuntu 16.04 32 bit
  - Statistics over 20 runs

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load slice with 9353 multi-conditions from XML snapshot + registration to IOV type [Mostly XML parsing]</td>
<td>~1.09 s</td>
</tr>
<tr>
<td>Compute 2493 alignments from conditions</td>
<td>~0.015 s</td>
</tr>
<tr>
<td>Fill slice from cache</td>
<td>~0.08 s</td>
</tr>
</tbody>
</table>

- Subsequent accesses nearly for free, since caches are active
- Influence of disk cache of XML files on timing?
DDCond: Status

- Described functionality is implemented
  - Tested with xml-input
  - Interfaced to LHCb conditions database for performance tests
- Prerequisite for the development of the handling of (mis-)alignments
- Documentation to be written
- No persistency implementation envisaged besides simple xml
  - Flux in the LHC community: COOL to be retired
  - If required adapt to coming database plugins

\(^{(1)}\) See also presentation from H.Grasland (AIDA\(^{2020}\), WP3, Task 3.4)
• Motivation and Goals
• Simulation
• Conditions support
• Alignment Support
  => DDAlign
• Miscellaneous
• Summary
DDAlign: Detector Alignment

- **Fundamental functionality to interpret event data**
  - Model mis-placement by construction
    - Non-ideal mounts of detector components
  - Must handle imperfections
    - Geometry => (Mis)Alignment
  - Anomalous conditions
    - Pressures, temperatures
    - Contractions, expansions
DDAlign: Standard Disclaimer

DDAlign does not provide algorithms\(^{(1)}\) to determine alignment constants and never will. DDAlign supports hosting the results of such algorithms and applies the resulting imperfections.

\(^{(1)}\) Algorithms are provided by WP3, Task 3.3 (C. Parkes et al.)
DD4hep (WP3, Task 3.2) collaborates with Task 3.3, but does not intend to interfere.
Milestone: MS 40 (31/01/2017)
DDAlign: Global and Local Alignments

• **Global alignment corrections**
  - Physically alters geometry
  - Intrinsic support by ROOT
  - By construction not multi-threaded
  - Possibility to simulate misaligned geometries

• **Local alignment corrections**
  - Geometry stays intact (either ideal or globally aligned)
  - Multi-threading supported, multiple versions
  - Local alignment corrections are conditions
  - Provide matrices from ideal geometry to world e.g. to adjust hit positions

• **Support both, emphasis on local alignment**
DDAlign: Global Alignments

• Interface implemented using TGeo: class TGeoPhysicalNode

• DD4hep interface needs revisiting
  – Implementation looks OK
  – Interface to load $\Delta$ – parameters from xml needs some adjustments

• Usage for iterative alignment purposes questionable
  – It was never foreseen in TGeo to reset an existing alignment and load new $\Delta$ – parameters \(^{(1)}\)

• Was put on hold to support multi-threading
  – Requires “Local Alignments”

\(^{(1)}\) private communication, A. Gheata, co-author of the ROOT Geometry Toolkit
Local Alignments and Conditions

- Local alignments data are conditions
  - Valid only for a certain time interval (IOV)
- Management is identical
  - Managed in pools
  - Access by slices
- Alignment transformations are derived conditions
  - Condition: $\Delta$ – parameters (corrections)
  - Derived: transformation matrices (to world or to hosting DetElement)
DDAlign: Alignment Corrections
(Δ - Parameters)

- Transformation matrix between two volumes is
  - Rotation
  - Or a rotation around pivot point
  - Followed by a translation
  - Combination

```cpp
class Delta {
public:
    typedef Translation3D Pivot;
    Position translation;
    Pivot pivot;
    RotationZYX rotation;
    unsigned int flags = 0;

    ...
};
```

```cpp
/// Initializing constructor
Delta(const Position& tr)
    : translation(tr), flags(HAVE_TRANSLATION) {}  
/// Initializing constructor
Delta(const RotationZYX& rot)
    : translation(), rotation(rot), flags(HAVE_ROTATION) {}  
```

- Use hints for faster computation (flags)
DDAlign: Apply $\Delta$ - Parameters

- Trickle-up the hierarchy and compute the matrices the most effective way
- Re-use intermediate results
Alignment handling: Code example

(see examples/AlignDet/src/*.cpp for the detailed usage of this code-fragments)

```cpp
lcdd.fromXML(input); // First we load the geometry

ConditionsManager condMgr = ConditionsManager::from(lcdd);
AlignmentsManager alignMgr = AlignmentsManager::from(lcdd);

// Load delta parameters: Use here simple plugin
char* deltas[] = {"Delta-Params.xml"};
lcdd.apply("DD4hep_ConditionsXMLRepositoryParser",1,deltas);

// Project required conditions into conditions slice
IOV iov iov_type_run,1500); // Project conditions for run 1500
ConditionsSlice* slice = createSlice(condMgr, *iov_typ);
ConditionsManager::Result cres = condMgr.prepare(iov, *slice);

// Register callbacks to transform Delta to matrices
...

// Compute the tranformation matrices
AlignmentsManager::Result ares = alignMgr.compute(*slice);
```
Support for Alignment Calibrations

- Common activity with WP3 Task 3.3 (C.Burr et al.)
- Development of facade object to simplify
  - the access,
  - the modification and
  - the management of alignment corrections for calibration processes
- Functionality
  - Bulk buffering and application of $\Delta$-parameters followed by re-computation of the transformation matrices
Alignment Calibrations: Code Example

(see examples/AlignDet/src/AlignmentExample_align_telescope.cpp for details)

```cpp
/// Use the created (and projected) conditions slice
ConditionsSlice* slice = ...
/// Create calibration object.
AlignmentsCalib calib(lcdn,*slice);
/// Update call may be specialized. Hence, no default
calib.derivationCall = new DDAlignUpdateCall();
/// Attach to DetElement placements to be re-aligned
Alignment::key_type key_tel = calib.use("/world/Telescope");
Alignment::key_type key_ml = calib.use("/world/Telescope/module_1");
calib.start();  // Necessary to enable dependency computations!

/// Let's "change" (re-align) some placements:
Delta delta(Position(333.0,0,0));
calib.setDelta(key_tel,Delta(Position(55.0,0,0)));
calib.setDelta(key_ml,Delta(Position(333.0,0,0),Rotation(pi/2,0,0)));

/// Push delta-parameters to the conditions objects
calib.commit(AlignmentsCalib::DIRECT);

/// Now all alignment conditions have the updated delta parameters.
/// All marices of the derived conditions are updated!
```
Alignment: Results

- DD4hep and alignment tools now used by Bach
- Please see presentation of C. Burr et al.
- MS40 (report)

The Bach alignment toolkit - Developments

- Developments:
  - Changed build system to use CMake
  - Rewritten to use DD4hep for geometry instead of ROOT
DDAlign: Status

• Implemented Global and Local (mis-)alignment
  – xml parser for Global (mis-)alignment constants needs re-visiting

• Started to integrate Local misalignments with the alignment procedures developed within WP3, Task 3.3
  – MS40: Running prototype for alignment Toolkit
  – To be tested in “real world” during test-beam at Desy (S. Borghi, C. Burr, C. Parkes)

• Documentation to be written
● Motivation and Goals
● Simulation
● Conditions support
● Alignment Support
● Miscellaneous
● Summary
Miscellaneous

- Main weak point is documentation
  - Need to revisit DDAlign design document
  - DDCond and DDAlign user manuals
- Need to build a test suite
  - Mainly for global alignment procedures
  - For local alignment procedures this should come for ‘free’ from Task 3.3
**Toolkit Users**

**Good news: We start to see contributions from users outside base community (ILC, CLICdp)**

- FCC, SiD
  - ILC  F. Gaede et al.
  - CLICdp  A. Sailer et al.
  - SiD  W. Armstrong
  - FCC-eh  P. Kostka et al.
  - FCC-hh  A. Salzburger et al.
  - CALICE  Calorimeter R&D, 280 persons: Started
  - FCC-ee  Some interest

<table>
<thead>
<tr>
<th></th>
<th>DD4hep</th>
<th>DDG4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILC</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CLICdp</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SiD</td>
<td>X</td>
<td>?</td>
</tr>
<tr>
<td>FCC-eh</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>FCC-hh</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CALICE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- Motivation and Goals
- Simulation
- Conditions support
- Alignment Support
- Miscellaneous
- Summary
Summary

- The DD4hep core and DDG4 simulation extension were consolidated and are to a large degree on maintenance level
  - Deployed by various customers
- Support for conditions handling is implemented
- Support for alignment handling is being used by collaborators from WP3
- Documentation for DDCond and DDAlign is weak and must be improved
- We are approaching the “polishing phase”
Questions and Answers
Backup
Implementation: Geometry

Subdetector Hierarchy (Tree)

- Detectors
  - DetectorElement
    - PlacedVolume
      - [TGeoNode]
        - PlacedVolume
          - [TGeoMatrix]
    - LogicalVolume
      - Envelope
        - [TGeoShape]
    - Material

- Alignment
- Conditions
- Readout
- Visualization
- Segmentation

Geometry

- GDML content
  - [TGeoBox]
  - [TGeoCone]
  - [TGeoTube]

Subdetector status (conditions)

Volume: 1

Placement: 0...1

Material

Visattr: 0...1

Transform: 1

Children: 1..n

Detector: 1
Views & Extensions: Users Customize Functionality

DD4hep is based on handles to data
- Clients only use the handles
- Possibility of many views based on the same DE data
  - Associate different behavior to the same data
  - Views consistent by construction
  - User data according to needs
- Be prudent: blessing or curse
  - User data: common knowledge
  - No one fits it all solution
  - Freedom is also to not do everything what somehow looks possible
Example of a DDG4 Action Sequence: Event Overlay with Features

- Combine simple and reusable modules
- Input module
- Any data format
- Primary vertex smearing
- Primary vertex boost
- Common: initialization, final merge
Multiple Input Sources

Compact description *xml*

Detector constructors *c++*

python

Conditions DB

DDDB converter *c++*

CAD drawing

CAD converter *c++*

Generic Detector Description Model
Based on ROOT TGeo *c++*
LHCb Detector