Table of Contents

THIS PAGE IS OBSOLETE, PLEASE GO TO CondDBHowTo

Getting started with CondDB (using Gaudi/LHCb)

Overview

In this TWiki page, I'll try to give the fundamental informations needed to start using the Conditions DataBase (CondDB).

Few basic concepts will be explained together with the necessary steps that allow a user to experiment with conditions and then prepare to use them in real life.

I'm currently preparing another the page CondDBHowTo to collect detailed instructions for the most common tasks.

Introductory stuff

I am not going to tell what a Condition Database (CondDB) is or what COOL stands for. You can find all the information on the Conditions Database Project web page.

I shall briefly explain the basic concepts needed for an LHCb user.

Condition objects

Of course a CondDB contains *conditions*. Whatever a *condition* is in your use-case (a temperature, a pressure, the number of people that were hanging around the counting room...) you need a way of using the information inside your piece of C++ code (an Algorithm, a DetectorElement, a Tool...).

LHCb provides a flexible class that can contain almost all types of information you may need: **Condition**. The main characteristics of this class are that it inherits from DataObject (which means that it can go into a transient data store), implements the IValidity interfaces (it has two fields which indicate the period in time for which the data contained are valid) and it is a ParamValidDataObject like DetectorElement. The class ParamValidDataObject provides the necessary infrastructure for an associative container of integers, doubles, std::strings and vectors of them. Each datum is identified by a string.

Example
A Condition called "Status" in "/dd/Conditions/ReadoutConf/MyDetector" can contain the following data:

- double "Temperature" : 27.8
- std::vector "BrokenChannels" : 1, 5, 7, 28
- std::vector "Thresholds" : 25.1, 20.6, 29.0

Persistency of Condition objects

Condition objects need to be read from a persistend medium, like a file or the CondDB, so we need converters.

Currently Condition objects can only be converted from an XML representation. This XML representation can be put in a file (as it is already done for DetectorElement objects) or can be put as a string inside the CondDB.

Since it can be annoying to write XML code by hand for lots of conditions, the method Condition::toXml() has been provided. So you can prepare your Condition object using its methods, then convert it to an XML string that can be put in a file or written to the CondDB.

Example
The Condition object of the previous example can be represented by a file (let's call it "ROstatus.xml") containing the following lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DDDB SYSTEM "conddb:/DTD/structure.dtd">
<DDDB>
  <condition name="Status">
    <param name="Temperature" type="double">
      27.8
    </param>
    <paramVector name="BrokenChannels" type="int">
      1 5 7 28
    </paramVector>
    <paramVector name="Thresholds" type="double">
      25.1 20.6 29.0
    </paramVector>
  </condition>
</DDDB>
```

For a detailed example you can look into the algorithm PopulateDB of the package Ex/DetCondExample.

Use Conditions

Now you know how to prepare the XML representation of a Condition object, but that is of no use if you are not able to access it from your C++ code.

Condition objects and the "Transient Store"

I just said that Condition objects can be registered to a transient store.

Since Condition objects are a bit special and are ideally connected to DetectorElement objects it seems natural to put them in the same transient store: the Detector Data Store (DDS).

One can access the DDS through the service DetDataSvc (the class GaudiAlgorithm provides useful methods to simplify access to the DDS).

In order to connect a path in the DDS to an object in an XML file, it is enough to add the appropriate tags in the root XML file. You need a nested "catalog" for each directory you want to see in the DDS, and a "conditionref" that tells the system where to get the description for your Condition object. (I know that it is not so clear, but I hope to do a better job with the example.)

Example
I assume you downloaded a copy of the XML Conditions package with a command like getpack Det/XmlConditions v1r4 The file DDDB/Conditions/MainCatalog.xml contains the lines:

(...)

```
  <catalog name="ReadoutConf">
    <catalogref href = "Velo/ReadoutConfCatalog.xml#Velo"/>
    <catalogref href = "TT/ReadoutConfCatalog.xml#TT"/>
    <catalogref href = "IT/ReadoutConfCatalog.xml#IT"/>
    <catalogref href = "Spd/ReadoutConfCatalog.xml#Spd"/>
    <catalogref href = "Prs/ReadoutConfCatalog.xml#Prs"/>
    <catalogref href = "Ecal/ReadoutConfCatalog.xml#Ecal"/>
    <catalogref href = "Hcal/ReadoutConfCatalog.xml#Hcal"/>
    <catalogref href = "Muon/ReadoutConfCatalog.xml#Muon"/>
    <catalogref href = "Rich1/ReadoutConfCatalog.xml#Rich1"/>
```

```
    <catalogref href = "Rich2/ReadoutConfCatalog.xml#Rich2"/>
  </catalog>
```

(...)

I want to add to the DDS the entry "/dd/Conditions/ReadoutConf/MyDetector/Status", where I want to find my Condition object, so I put the file ROstatus.xml into the directory DDDB and I modify DDDB/conditions.xml to have something like:

(...)

```
  <catalog name="ReadoutConf">
    <catalogref href = "Velo/ReadoutConfCatalog.xml#Velo"/>
    <catalogref href = "TT/ReadoutConfCatalog.xml#TT"/>
    <catalogref href = "IT/ReadoutConfCatalog.xml#IT"/>
    <catalogref href = "Spd/ReadoutConfCatalog.xml#Spd"/>
    <catalogref href = "Prs/ReadoutConfCatalog.xml#Prs"/>
    <catalogref href = "Ecal/ReadoutConfCatalog.xml#Ecal"/>
    <catalogref href = "Hcal/ReadoutConfCatalog.xml#Hcal"/>
    <catalogref href = "Muon/ReadoutConfCatalog.xml#Muon"/>
    <catalogref href = "Rich1/ReadoutConfCatalog.xml#Rich1"/>
    <catalogref href = "Rich2/ReadoutConfCatalog.xml#Rich2"/>
    <catalog name="MyDetector">
      <conditionref href="ROstatus.xml#Status"/>
    </catalog>
  </catalog>
```

(...)

Now, if your program is able to access the DDS, you should be able to get a pointer to the Condition object by adding to your GaudiAlgorithm the line

```
Condition* myCond = getDet<Condition>("Conditions/ReadoutConf/MyDetector/Status");
```

(of course you should have included the file "DetDesc/Condition.h" )

DetectorElement

DetectorElement objects are special cases.

If you have a DetectorElement object that needs a Condition object in order to operate properly, you can specify it in the DetectorElement object's XML representation using the special tag "conditioninfo" and giving a name to the link. Afterwards you can access the condition object by calling the method DetectorElement::condition() passing to it the name you assigned to the link in the XML representation.

Example
In the Ex/DetCondExample package you can find a file containing the lines:

```
    <detelem classID="6669999" name="Dummy">
      <param name="comment" type="string"> Dummy detector element used to test the update </param
      <conditioninfo name="ReadOut" condition="/dd/Conditions/ReadoutConf/DummyDE/Condition"/>
      <conditioninfo name="Temperature" condition="/dd/Conditions/Environment/DummyDE/Temperature
    </detelem>
```

An algorithm in the same example package is accessing those condition with instructions like:

```
int nROChannels = dummyDE->condition("ReadOut")->param<int>("NChannels");
```

Easy, isn't it?

The UpdateManagerSvc

When a Condition object is stored in the CondDB, it has an associated IOV (interval of validity), which means that when you are going to process an event which time is outside the IOV of a Condition object, somebody must get the appropriate Condition object from the CondDB and replace the one which is not needed with the one just retrieved. The UpdateManagerSvc is the one who does it. I give you just few hints about how to use it (more details can be found on the talk I gave some time ago).

The idea is that if your GaudiAlgorithm/DetectorElement needs that a condition is kept up-to-date, it must tell that to the UpdateManagerSvc (UMS). (From GaudiAlgorithm you don't even need to know that there is an Update Manager Service)

The most common use cases are:

1. you just want that a Condition object is updated when needed.
2. you do some calculation on the data provided by a Condition object, so you need to do special operations when the content of the Condition object changes.

For the first use case it is enough to do (from a GaudiAlgorithm):

```
registerCondition<MyAlg>("Conditions/ReadoutConf/MyDetector/Status");
```

You can forget about the existence of the UMS, it will take care of updating the Condition object when needed.

For the second use case, let's assume that your algorithm of class MyAlgorithm needs to do an average of the thresholds values in the ReadOut status. Of course you do not want to recalculate the average every event, so you can put the instructions to calculate it in the method MyAlgorithm::calcAverage() returning StatusCode and prepare a data member to hold the pointer to the condition: MyAlgorithm::m_ROStatus. Then in the initialization method you can put:

```
registerCondition("Conditions/ReadoutConf/MyDetector/Status",m_ROStatus,&MyAlgorithm::calcAverage
```

and finish the initialization with:

```
return runUpdate();
```

to be sure that all the needed operations are performed also the first time.

Ex/DetCondExample is always a good place to look for more details. Try ExampleAlg.h and ExampleAlg.cpp.

The database

So far, I described how to prepare your Condition objects, put them in an XML file and use them from an Algorithm (or any another piece of C++ code). In this way, you will be able to set up the needed code, both XML and C++, to start using conditions, but you cannot benefit from the possibility of having the conditions changing during your job. In fact, even if Condition objects have an IOV, when they are read from an XML file they are considered as always vaild (IOV spans from 0ns, the minimum, to 9223372036854775807ns, the maximum).

In order to have Condition objects with a limited IOV, you need a database. We do not access a database

directly, but through the COOL API, which allows an easy handling of Condition objects.

The CondDB

I'm not going into details, I just need to give an overview of how the CondDB is structured.

The data in the CondDB are organized in a tree-like structure, made of two types of nodes:

- folder-sets: nodes of the tree that can contain other nodes (you can think of them as directories in a filesystem)
- folders: nodes that contain data objects (they are like files in a filesystem),

both of them identified by a name and a path (like in a filesystem). Folders can be of two types: single-version and multi-version.

Single-version folders can contain only one version of a data object at a given point in the time range, which means that if you decided to store a value valid from time t0 to t1, you cannot change your mind later, and replace that value. This kind of folders should be used for data like temperatures or H.V. readings which are read from hardware sensors.

Multi-version folders allow the user to store values that can be superseded by others (for the same point in time). The obvious example are alignments or calibrations, that you once measure for a given run and one year later you can re-measure them for the same run number and find different values which should be used instead of those previously measured. So, like in a CVS archive, you will end up, for a given point in time, with many possible values of the same quantity, and, always like in a CVS repository, you decide which one to use by specifying a symbolic tag id.

Data objects, both in single-version and in multi-version folders, consist of an IOV and a payload. The payload is the real datum. From the COOL API point of view, it is a set of data items (integers, floats, strings...). In LHCb, we use only one or more strings which contain the XML representation of the DataObject. The IOV tells you for which set of events the information provided into the payload has to be used. In COOL the IOV is limited by two ValidityKey instances (unsigned 63 bits integers), we use those integers to store two time points (*time* here is considered as the number of nanoseconds since 00:00:00 on January 1, 1970, i.e. a Unix-like time definition with nanosecond precision) and we select the Condition object to use for an event by the *event time*.

The back-end

COOL API stores (retrieves) the informations to (from) an SQL database. We can chose between three possible database engines (or back-ends):

- Oracle: it is the database engine chosen for all CERN databases, and for which CERN IT provides support.
- MySQL: open source relational database server which can be found on any Linux distribution.
- SQLite: open source C library which provides the functionality of a relational database without the need of setting up a server, since it uses a plain file (only from COOL 1.2)

Here, things are evolving quickly. The version of COOL used by LHCb v22r0 is 1.3.4 with SQLite support and Python bindings.

To test and experiment, I suggest to use SQLite files, which are easy to back up, delete and need no server configuration.

If you need a remote database, you can obtain an account on an Oracle server by asking the CERN IT DB

Group, but probably it's easier to ask me for an account on the MySQL server I set up for testing.

Prepare the DC06-compatible databases

Since few days, you can get a new package which aim is to allow us to move away from XML files and use a real database.

From lxplus, you should prepare the working area for a project. I use Brunel, but you can use anything that uses a version of LHCb > v22r0.

```
~ > setenvBrunel v31r0
~/cmtuser/Brunel_v31r0 > getpack Tools/CondDBUI v1r1
~/cmtuser/Brunel_v31r0 > getpack Det/SQLDDDB v1r0
~/cmtuser/Brunel_v31r0 > cd Det/SQLDDDB/v1r0/cmt
~/cmtuser/Brunel_v31r0/Det/SQLDDDB/v1r0/cmt > cmt run python ../python/create_db.py
```

This created 2 databases: one for XmlDDDB and one for XmlConditions.

Set up the Gaudi job to use a database

Once you followed the instructions above, you have to add the line `use SQLDDDB v* Det` to your requirements file, and `#include "$SQLDDDBROOT/options/SQLDDDB.opts"` at the end of your options. This is all you need to use an SQLite database instead of XML files.

Event time

Now you should be able to set up your job options to read from a database, but that's not enough yet: you need an event time!

To retrieve from the database a Condition object, the framework uses the time set as EventTime in DetectorDataSvc. Anyway, MonteCarlo events do not have a defined time, and if you are not reading events (as it happens for the tests in Ex/DetCondExample) it is even harder to think of an event time.

You can think of setting yourself that event time, but in most cases it is not needed. The package Det/DetDesc provides a service called EventClockSvc which uses a tool to generate the event time just before UpdateManagerSvc is called to check for updates.

The basic implementation of the tool is very simple and allows you to set a fixed event time or to increment it every event by a certain amount. Its options are:

- `EventClockSvc.FakeEventTime.StartTime` (longlong): The time (as defined above) to use for the first event
- `EventClockSvc.FakeEventTime.TimeStep` (longlong): The number of nanoseconds to add to the current event time every new event processed

Both options are 0 by default.

For the data, an implementation of the tool that uses the ODIN back is available, but feel fre to write your own for special cases.

Final remarks

I think that I wrote everything it is needed to start playing with CondDB, but I could have easily forgotten to mention some detail.

I intentionally omitted a few *advanced* features to avoid confusing the users too much.

If you think that something is missing or you need more information, feel free to ask me, and I'll add the missing details.

-- MarcoClemencic - 22 Feb 2007

This topic: LHCb > CondDBGettingStarted
Topic revision: r12 - 2007-10-17 - MarcoClemencic