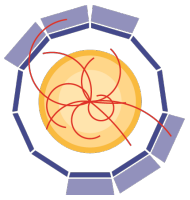

AIDA 2020

Advanced European Infrastructures for Detectors at Accelerators

DDCond

Conditions 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*. To properly interpret the electronic signals which form the response of particle collisions inside these subdetectors other auxiliary data are necessary. These auxiliary data typically are time dependent - though normally at a much longer scale than the event data itself. The conditions part of the `DD4hep` toolkit, called `DDCond` , addresses the management and the access of such conditions data. The Manual describes a solution, which pools groups of these time dependent data according to its validity. This approach firstly allows to quickly access all relevant data for a given particle collision. efficient caching mechansims and allows to quickly determine which data items need to be accessed from a persistent medium. 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	**10/4/2014**	**Markus Frank CERN/LHCb**

Contents

1 Introduction

In a high energy physics experiment the data originating from particle collisions (so called *Event-data*) in most cases require supplementary, mostly environmental, calibration- or alignment data to extract the physics content from the recorded event data. These supplementary data are time-dependent and typically called *conditons*. The ability of an experiment to produce correct and timely results depends on the complete and efficient availability of needed conditions for each stage of data handling. This manual should introduce to the `DDCond` toolkit, which provides access to conditions data within the `DD4hep` data structures [1].
The `DDCond` extensions to the `DD4hep` toolkit formalize both the access and the management to time-dependent data necessary to process the event data. In this manual we will shortly describe the model used to organize and manage the conditions data internally and then describe the user interface to actually perform the required actions. These conditions data typically are stored in a database. Nearly every high energy physics experiment has strong feelings how to efficiently read and store the conditions data in terms of tablespace organization and data format. For this reason `DDCond` does not attempt to solve the persistency problem, but rather defines an interface used to load missing data items from the persistent medium. Any persistent data representation, which can fulfill the requirements of this interface may be adopted by the `DDCond` conditions caching and access mechanism.
At the end of this manual a walk-through of an example is discussed to illustrate, which steps have to be performed to use the `DD4hep` conditions store within a client application.

1.1 Definition of Conditions Data

Conditions data are firstly

- raw data values. Raw data values are recorded from measurement devices such as thermometers, pressure devices or geometrical parameters resulting from survey parameters and others. These data may change with time, but have one and only one version.
- Secondly there is the large group of data derived from the raw values. These derived values are transformed from one or several raw values into new data items with an interval of validity being the intersection of the intervals of validity of its ingredients. Effectively every raw measurement requires calibration to represent a useful value. Hence, nearly all raw values require such a transformation. Since these data are re-calibrated, not only one version exists, but many e.g. as a result of different calibration algorithms.

Typically one data processing application predefines for all events to be processed the corresponding versions of the conditions data to be used. This time span typically is much larger than the intervals of validity of single data value. The collection of all individual data item version for such a large time interval is called a "global tag". Within such a global tag, several conditions values of the same data item, but with a different interval of validity may be accessed.
Given this definition it is evident that the division between raw values and derived conditions is rather fluent. Derived data as a result of a calibration process are technically identical to raw values in an analysis application using these re-calibrated constants. Hence, as soon as derived data enter the conditions database they are technically identical to raw values.
To support such calibration processes producing derived conditions data, `DDCond` provides a mechanism to project new values given a set of recipes provided by the user. This recipes can project a set of coherent new conditions for a given event time accordingly.

1.2 Conditions Slices

Conditions slices define a coherent set of conditions data valid for an event recorded at a specific time. Each of the individual conditions of the slice has a certain interval of validity, hence the validity of the entire slice is defined as the intersection of these validities. As a corollary, the slice may be valid for more than one event as long as the event's time stamp is within this intersection. To maximize

the flexibility, and to allow users to implement private slice implementations, slices have a common interface, the *ConditionsMap* (See section 2.2). For most practical purposes and to share tools between slice implementations, this interface is sufficient.

2 Generic Concepts and Design

The DD4hep conditions mechanism was designed to be very flexible concerning back-end implementations. Most of the conditions and alignment utilities offered by DD4hep are available if a minimal interface is respected. This minimal interface includes a container called *ConditionsMap* (See section 2.2) and the layout of the conditions objects (See section 2.1). The conditions objects contain a flexible user defined payload and a generic, interface used to interact with tools and the generic container object or conditions slices as described in section 1.2.

2.1 Condition Objects and Conditions Data

A conditions objects serves two purposes:

- Firstly, it supports the basic functionality which is generic to any condition – independent of the actual user payload. This information includes access to the interval of validity and the key to uniquely identify the condition.
- Secondly, the objects hosts and manages a user payload, the actual conditions data. These data are freely user defined. An automatic parsing mechanism from a string representation is supported if the payload-class can properly described using a boost::spirit parsing structure. Default type implementations are defined for
  - all primitive data types,
  - ROOT::Math::XYZVector, ROOT::Math::XYZPoint, ROOT::Math::PxPyPzEVector.
  - a number of STL containers of the above basic data types:
    std::vector<TYPE>, std::list<TYPE>, std::set<TYPE>,
    std::map<int,TYPE>, std::map<string,TYPE>,
    std::pair<int,TYPE>, std::pair<string,TYPE>.

  Additional types can easily be implemented using boost::spirit if the basic representation is in the form of a string. Dummy boost::spirit parsers may be implemented if the conversion to and from strings is not required.
- Thirdly, it supports the basic functionality required by a conditions management framework, which implements the *ConditionsMap* interface.

For completeness we include here the basic data access methods of the conditions class (see DD4hep/Conditions.h):

```
class Condition: public Handle<detail::ConditionObject> {
  /** Interval of validity                                    */
  /// Access the IOV type
  const IOVType& iovType()  const;
  /// Access the IOV block
  const IOV& iov()  const;

  /** Conditions identification using integer keys.           */
  /// Hash identifier
  key_type key()  const;
  /// DetElement part of the identifier
  detkey_type  detector_key()  const;
  /// Item part of the identifier
  itemkey_type item_key()  const;
```

```
  /** Conditions meta-data and handling of the data binding  */
  /// Access the opaque data block
  OpaqueData& data()  const;
  /// Access to the type information
  const std::type_info& typeInfo() const;
  /// Access to the grammar type
  const BasicGrammar& descriptor() const;
  /// Check if object is already bound....
  bool is_bound()  const  {  return isValid() ? data().is_bound() : false;  }
  /// Bind the data of the conditions object to a given format.
  template <typename T> T& bind();
  /// Set and bind the data of the conditions object to a given format.
  template <typename T> T& bind(const std::string& val);
  /// Generic getter. Specify the exact type, not a polymorph type
  template <typename T> T& get();
  /// Generic getter (const version). Specify the exact type, not a polymorph type
  template <typename T> const T& get() const;
  ...
};
```

Please be aware that the access to the IOV and the IOVType is only possible if supported by the caching mechanism.
Using the *OpaqueData* data structure and its concrete implementation, the user can map any data item to the conditions object using the *bind*() method and retrieve the data back using *get*(). Clearly, the left should know what the right does and the types to be retrieved back must match be bound data types.
The following code-snippet shows how to bind conditions data:

```
Condition cond = ...;
// Fill conditions data by hand:
std::vector<int>& data = cond.bind<std::vector<int> >();
data.push_back(0);
data.push_back(1);    .....

// Fill conditions data from the string representation using boost::spirit:
std::string str = "[0,1,2]";
std::vector<int>& data = cond.bind<std::vector<int> >(str);
int i = data[0]; ....
```

This is an example how to access already bound data:

```
Condition cond = ...;

// Fill conditions data by hand:
std::vector<int>& data = cond.get<std::vector<int> >();
```

2.2 The ConditionsMap Interface

The *ConditionsMap* interface (see defines the lowest common denominator to allow tools or clients to interact with conditions of a given slice. This interface defines the interaction of clients with a conditions slice. These interactions cover both the data access and the data management within a slice. The interface allows to

- access individual conditions by the detector element and a given item key. The interface allows
- to scan conditions according to the detector element or
- to scan all conditions contained. Further it allows
- insert conditions to the mapping and
- to clear the content.

The provision of these basic interaction mechanisms allows us to build very generic tools firstly for conditions, but also later for the management and th computation of alignment data as described in the `DDAlign` manual [2].
The *ConditionsMap* interface class, which supports this basic functionality has the following entry points:

```
class ConditionsMap   {
public:
  /// Insert a new entry to the map. The detector element key and
  /// the item key make a unique global conditions key
  virtual bool insert(DetElement         detector,
                      Condition::itemkey_type key,
                      Condition          condition) = 0;
  /// Interface to access conditions by hash value. The detector element key
  /// and the item key make a unique global conditions key
  virtual Condition get(DetElement              detector,
                        Condition::itemkey_type key) const = 0;
  /// Interface to scan data content of the conditions mapping
  virtual void scan(const Condition::Processor& processor) const = 0;

  /// No ConditionsMap overload: Access all conditions within
  /// a key range in the interval [lower,upper]
  virtual std::vector<Condition> get(DetElement              detector,
                                     Condition::itemkey_type lower,
                                     Condition::itemkey_type upper)  const;

  /// Interface to partially scan data content of the conditions mapping
  virtual void scan(DetElement                  detector,
                    Condition::itemkey_type     lower,
                    Condition::itemkey_type     upper,
                    const Condition::Processor& processor) const;
};
```

Such *ConditionsMap* implementations can easily be constructed using standard STL maps. The lookup key is constructed out of two elements:

- The detector element this condition belongs to and
- an identifier of condition within this detector element.

An efficient implementation of a longword key would consist of the tuple:

$$[hash32(conditions\ name), hash32(det-element\ path)],$$

which resembles to an ordered sequence of conditions according to their detector element. A special implementation, which implements this user interface is the *ConditionsSlice* implemented in the `DDCond` package (See section 3 for details).

2.3 Common Conditions Tools

- **ConditionsPrinter** A tool to print conditions by scanning conditions for a single *DetElement* or the entire sub-tree. See DDCore/ConditionsPrinter.h for details).
- **ConditionsProcessor** A wrapper to support condition functors implementing the default callback:

  $$int\ operator()\ (Condition\ consition);$$

  The return value may be used to e.g. collect counters.

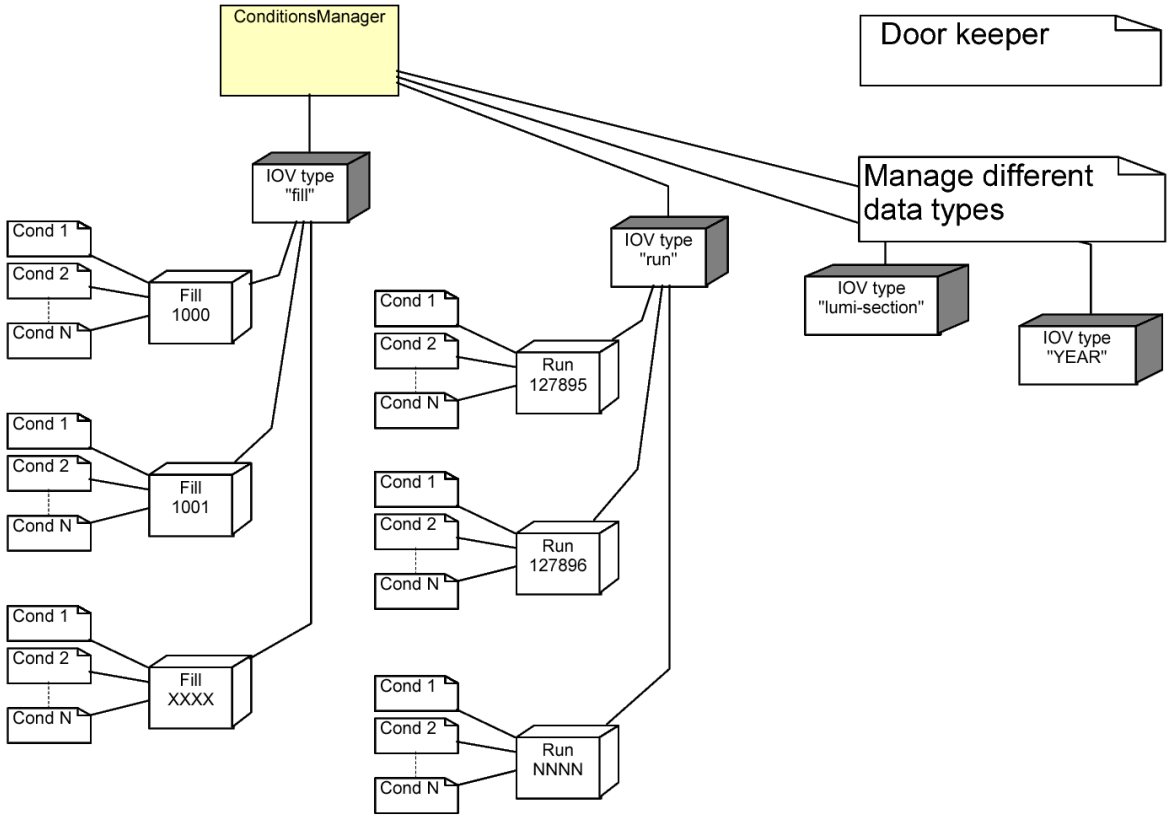

Figure 1: The graphical representation of the organisation of the conditions data in `DD4hep` .

3 DDCond Conditions Store and Slices

The *ConditionsMap* interface allows tools to work with various conditions data stores. `DDCond` provides an efficient implementation of such a store, which is described in the following chapters.

3.1 Data Organization

The basic assumption of the `DDCond` conditions store to optimize the access and the management of conditions data can be very simply summarized: it is assumed, that groups of data items exist, which have a common interval of validity. In other words: given a certain event, valid or invalid conditions can quickly be identified by checking the so called "interval of validity" of the entire group with the time stamp of the event. This interval of validity defines the time span for which a given group of processing parameters is valid. It starts and ends with a time stamp. The definition of a time stamp may be user defined and not necessarily resemble to values in seconds or fractions thereof. Time stamps could as well be formulated as an interval of luminosity sections, run numbers, fill numbers or entire years.
Groups of parameters associated to certain intervals of validity can be very effectively managed if pooled together according to the interval of validity. This of course assumes that each group contains a significant number of parameters. If each of these pools only contains one single value this would not be an efficient.
This assumption is fundamental for this approach to be efficient. If the data are not organized accordingly, the caching mechanism implemented in `DDCond` will still work formally. However, by construction it cannot not work efficiently. Resources both in CPU and memory would be wasted at run-time. The

necessity to properly organize the conditions data becomes immediately evident in Figure 1: Users can organize data according to certain types, These types are independently managed and subdivided into pools. Each of these pools manages a set of conditions items sharing the same interval ov validity.
The internal organization of the conditions data in `DDCond` is entirely transparent to the user. The description here is contained for completeness and for the understanding of the limitations of the implemented approach. If different requirements or access patterns concerning the access to conditions data arise, it should though be feasible to implement these fairly straight forward using a suited approach.

3.2 Slice Configuration and Data Access

As defined in section 1.2, the conditions slice is the main entity to project conditions suitable to process a given particle collision (see DDCond/ConditionsContent.h for details). Figure **??** shows the data content of a conditions slice. As shown also in Figure 3, there are several steps to be performed before a conditions slice is ready to be used:

1. Create the conditions data slice.
2. Setting up the data content of the slice by attaching an object of type *ConditionsContent*.
3. Preparing the conditions data slice.
4. Using the conditions data slice.

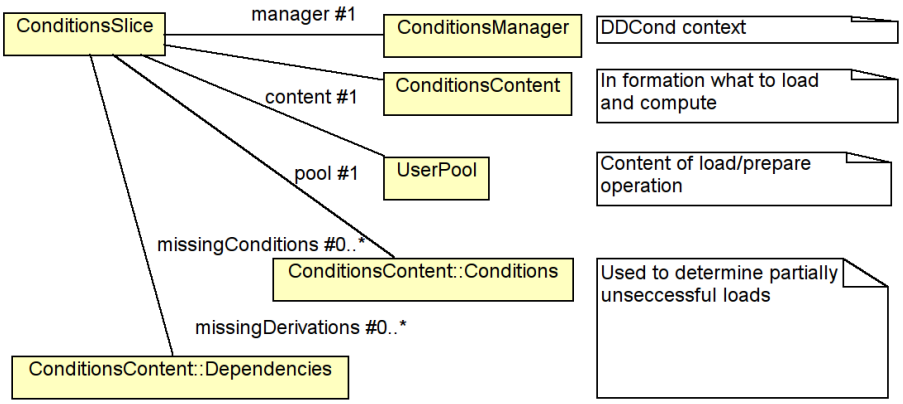

Figure 2: The data content of a *ConditionsSlice* containing the desired content (*ConditionsContent*), the pool to access the conditions data by key (*UserPool*) and optional containers showing the status of partial or unsuccessful load and prepare operations.

The *ConditionsContent* (see DDCond/ConditionsSlice.h for details) is a simple object, which contains load addresses to identify persistent conditions within the database/persistent schema used and a set of dependency rules to compute the corresponding derived conditions.
The *ConditionsSlice* holds a consistent set of conditions valid for a given interval of validity, which is the intersection of the intervals of validity of all contained conditions. The has the following consequences for the client when using a prepared *ConditionsSlice*:

- *ConditionsSlice* objects are prepared by the client framework. Specific algorithms and other code fragments developed by physicist users should not deal with such activities. In multi-threaded applications the preparation of a *ConditionsSlice* may be done in a separate thread.
- Once prepared, the slice nor the contained conditions may be altered. All contained conditions must be considered read-only.

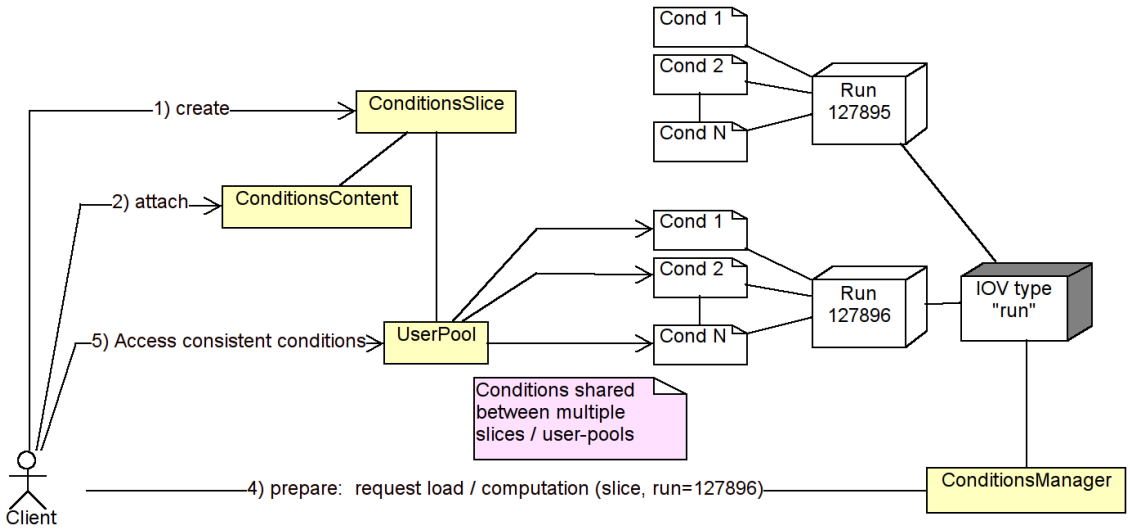

Figure 3: The interaction of a user with the conditions data store using the *ConditionsSlice* and the *ConditionsManager* interface to define the conditions content, prepare the data and then access the loaded data from the slice projected according to the required interval of validity.

- Since the slice is considered read-only, it can be used by multiple clients simultaneously. In particular, multiple threads may share the same slice.
- A *ConditionsSlice* may only be re-used and prepared according to a different interval of validity once no other clients use it.
- At any point of time any number of *ConditionsSlice* objects may be present in the client framework. There is no interference as long as the above mentioned requirements are fulfilled.

The fact that multiple instances of the conditions slices may be present as well as the fact that the preparation of slices and their use is strictly separated makes then ideal for the usage within multi-threaded event processing frameworks. As shown in figure 4, the following use cases can easily be met:

- Mulitple threads may share the same slice while processing event data (thread 1-3) as long as the time stamp of the event data processed by each thread is contained in the interval of validity of the slice.
- At the same time another thread may process event data with a different time stamp. The conditions for this event were prepared using another slice (thread 4-N).

3.3 Loading Conditions Data

The loading of conditions data is highly experiment specific. Different access patterns and load implementations (single threaded, multi-threaded, locking etc.) make it close to impossible to implement any solution fitting all needs. For this reason the loading of conditions is deferred to an abstract implementation, which is invoked during the preparation phase of a conditions slice if the required data are not found in the conditions cache. This data loader interface (see ConditionsDataLoader.h for details), receives all relevant callbacks from the framework to resolve missing conditions and pass the loaded objects to the framework for the management. The callback to be implemented by the framework developers are:

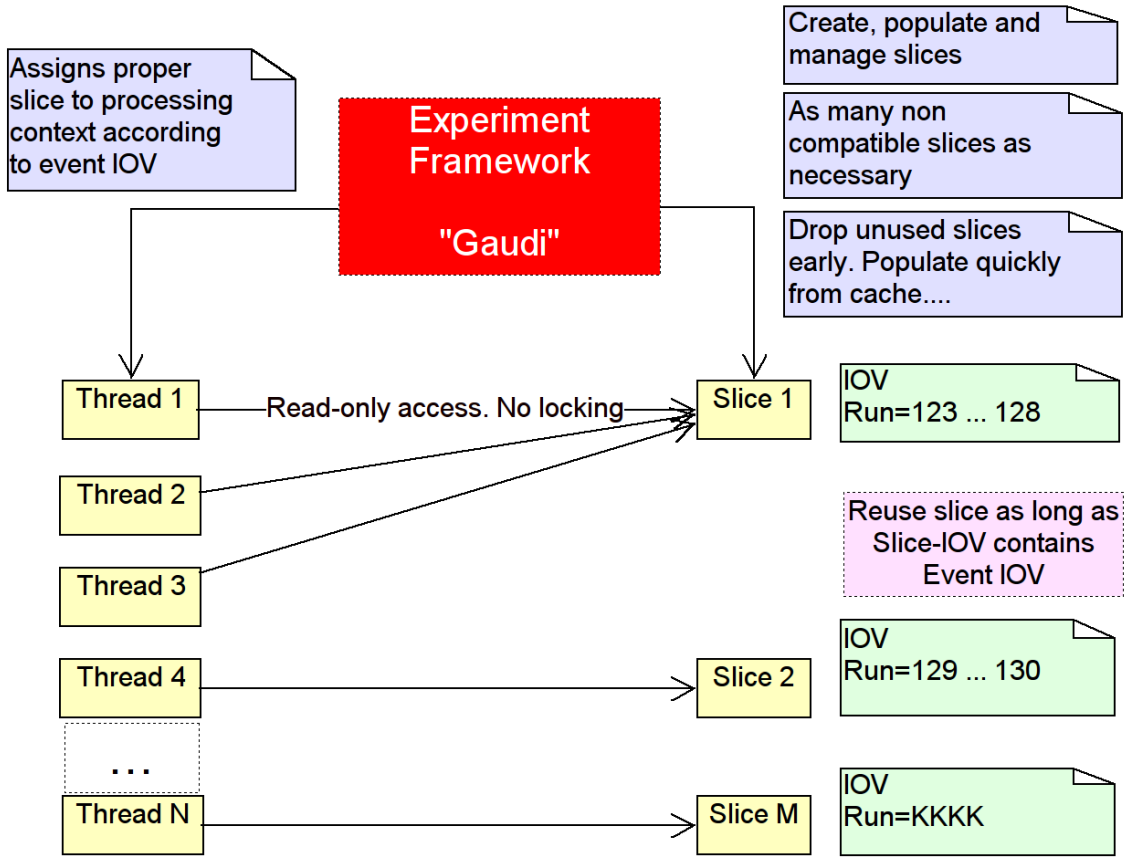

Figure 4: MT.

```
/// Interface for a generic conditions loader
/**
 *  Common function for all loader.
 */
class ConditionsDataLoader : public NamedObject, public PropertyConfigurable   {
  typedef Condition::key_type                                    key_type;
  typedef std::map<key_type,Condition>                           LoadedItems;
  typedef std::vector<std::pair<key_type,ConditionsLoadInfo*> > RequiredItems;
public:
  ....
  /// Load a number of conditions items from the persistent medium according to the required IOV
  virtual size_t load_many(  const IOV&       req_validity,
                             RequiredItems&   work,
                             LoadedItems&     loaded,
                             IOV&             combined_validity) = 0;
};
```

The arguments to the callback contain the necessary information to retrieve the requested items.

4 Example Walkthrough

4.1 Example to Save Conditions to a ROOT File

To illustrate the usage of the DDCond package when saving conditions, an example is discussed here in detail. The full example is part of the conditions unit tests and can be found in the DD4hep examples. (See examples/Conditions/src/ConditionExample_manual_save.cpp).
The examples uses conditions names and detector element names explicitly and hence requires a fixed detector description being loaded in memory. For simplicity we use here the Minitel example from the examples/ClientTests. However, the example is very generic and also the conditions are "generic", hence any geometry would work.

Prerequisites:

A valid compact geometry description to be loaded during the program startup.

Plugin Invocation:

A valid example conditions data file is required. Then use the default way to invoke plugins:

```
$ > geoPluginRun -destroy   \
                 -plugin DD4hep_ConditionExample_manual_save   \
                 -input file : ${DD4hep_DIR}/examples/AlignDet/compact/Telescope.xml   \
                 -conditions Conditions.root -runs 10
```

Example Code:

```
1  int          num_run    = <number of condition loads>;
2  const string conditions = <conditions file name>;
3  const string geometry   = <geometry compact xml description";
4
5  description.fromXML(geometry);
6  description.apply("DD4hep_ConditionsManagerInstaller",0,(char**)0);
7
8  ConditionsManager manager = ConditionsManager::from(description);
9  manager["PoolType"]       = "DD4hep_ConditionsLinearPool";
10 manager["UserPoolType"]   = "DD4hep_ConditionsMapUserPool";
11 manager["UpdatePoolType"] = "DD4hep_ConditionsLinearUpdatePool";
12 manager.initialize();
13
14 shared_ptr<ConditionsContent> content(new ConditionsContent());
15 shared_ptr<ConditionsSlice>   slice(new ConditionsSlice(manager,content));
16
17 const IOVType*    iov_typ = manager.registerIOVType(0,"run").second;
18 if ( 0 == iov_typ )
19   except("ConditionsPrepare","++ Conditions IOV type registration failed!");
20
21 Scanner(ConditionsKeys(*content,INFO),description.world());
22 Scanner(ConditionsDependencyCreator(*content,DEBUG),description.world());
23
24 // Have 10 run-slices [11,20] .... [91,100]
25 for(int i=0; i<num_run; ++i)  {
26   IOV iov(iov_typ, IOV::Key(1+i*10,(i+1)*10));
27   ConditionsPool*   iov_pool = manager.registerIOV(*iov.iovType, iov.key());
28   // Create conditions with all deltas. Use a generic creator
```

```
29    Scanner(ConditionsCreator(*slice, *iov_pool, INFO),description.world(),0,true);
30  }
31
```

Explanation:

Line	
1-3	Definition of processing parameters.
5	Load the detector description using the compact notation.
6	Install the conditions manager implementation using the plugin mechanism.
8	Access conditions manager instance from the Detector interface.
9-11	Configure the properties of the conditions manager.
12	Initialize the conditions manager instance.
14-15	Create an empty *ConditionsSlice* instance the container with the desired conditions content.
17	Register IOV type the Conditions Manager. The IOV types are part of the conditions persistency mechanism. They may not change with time and have to be defined by the experiment once and for all!
18-19	This is example specific and only a shortcut to fill the required conditions content and the derivation rules. In real life this would be intrinsic to the experiment's data processing framework.
18	Populate the *ConditionsContent* instance with the addresses (keys) of the conditions required: We scan the *DetElement* hierarchy and add a couple of conditions for each *DetElement*
19	Add for each *DetElement* 3 derived conditions, which all depend on the persistent condition derived_data. In the real world this would be very specific derived actions.

4.2 Example to Load and Prepare Conditions(Slices)

To illustrate the usage of the DDCond package when loading conditions, an example is discussed here in detail. The full example is part of the conditions unit tests and can be found in the DD4hep examples. (See examples/Conditions/src/ConditionExample_manual_load.cpp).
The examples uses conditions names and detector element names explicitly and hence requires a fixed detector description being loaded in memory. For simplicity we use here the Minitel example from the examples/ClientTests. However, the example is very generic and also the conditions are "generic", hence any geometry would work.

Prerequisites:

A valid example conditions data file is required, since in this example we load the conditions and inject them to the store from an already existing root file. To obtain such a file for a given geometry, execute the example plugin:

```
$ > geoPluginRun -destroy  \
                 -plugin DD4hep_ConditionExample_manual_save  \
                 -input file : ${DD4hep_DIR}/examples/AlignDet/compact/Telescope.xml  \
                 -conditionsConditions.root -runs 10
```

Plugin Invocation:

A valid example conditions data file is required. Then use the default way to invoke plugins:

```
$ > geoPluginRun -destroy  \
                 -plugin DD4hep_ConditionExample_manual_load  \
                 -input file : ${DD4hep_DIR}/examples/AlignDet/compact/Telescope.xml  \
                 -conditions Conditions.root -runs 10
```

Example Code:

```
1  int          num_run    = <number of condition loads>;
2  const string conditions = <conditions file name>;
3  const string geometry   = <geometry compact xml description";
4
5  description.fromXML(geometry);
6  description.apply("DD4hep_ConditionsManagerInstaller",0,(char**)0);
7
8  ConditionsManager manager = ConditionsManager::from(description);
9  manager["PoolType"]       = "DD4hep_ConditionsLinearPool";
10 manager["UserPoolType"]   = "DD4hep_ConditionsMapUserPool";
11 manager["UpdatePoolType"] = "DD4hep_ConditionsLinearUpdatePool";
12 manager["LoaderType"]     = "root";
13 manager.initialize();
14
15 shared_ptr<ConditionsContent> content(new ConditionsContent());
16 shared_ptr<ConditionsSlice>   slice(new ConditionsSlice(manager,content));
17
18 Scanner(ConditionsKeys(*content,INFO),description.world());
19 Scanner(ConditionsDependencyCreator(*content,DEBUG),description.world());
20
21 const IOVType* iov_typ = manager.iovType("run");
22 for ( int irun=0; irun < num_runs; ++irun )  {
23   IOV iov(iov_typ,irun*10+5);
```

```
24    ConditionsManager::Result r = manager.prepare(iov,*slice);
25    if ( r.missing != 0 )  {
26       except("Example",
27              "Conditions prepare step for IOV %s FAILED. There are %ld missing conditions.",
28              r.missing, iov.str().c_str());
29    }
30    Scanner(ConditionsPrinter(slice.get(),"Example"),description.world());
31  }
```

Explanation:

Line	
1-3	Definition of processing parameters.
5	Load the detector description using the compact notation.
6	Install the conditions manager implementation using the plugin mechanism.
8	Access conditions manager instance from the Detector interface.
9-12	Configure the properties of the conditions manager.
13	Initialize the conditions manager instance.
15-16	Create an empty *ConditionsSlice* instance the container with the desired conditions content.
18-19	This is example specific and only a shortcut to fill the required conditions content and the derivation rules. In real life this would be intrinsic to the experiment's data processing framework.
18	Populate the *ConditionsContent* instance with the addresses (keys) of the conditions required: We scan the *DetElement* hierarchy and add a couple of conditions for each *DetElement*
19	Add for each *DetElement* 3 derived conditions, which all depend on the persistent condition derived_data. In the real world this would be very specific derived actions.
22-31	Emulate a pseudo event loop: Our conditions are of type "run".
23	This is the IOV we want to use for this "processing step". The conditions filled into the slice during the prepare step will satisfy this IOV requirement.
24	Conditions prepare step. Select the proper set of conditions from the store (or load them if needed). Attach the selected conditions to the user pool.
25-29	Check the result of the prepare step. If anything would be missing, the parameter r.missing of the return code would be non-zero.
30	Emulate data processing algorithms: Here we only scan the *DetElement* tree and print all conditions. We know what we expect since we defined the content the same way! In the printer we can access the conditions directly from the slice, since the slice implements the *ConditionsMap* interface.

References

[1] M.Frank, `DD4hep` manual.

[2] M.Frank, `DDAlign` manual.