Conditions Database Overview

org.lcsim Conditions Database

Overview

The conditions database is designed to allow a running analysis or reconstruction module to access information about the run "conditions". In our current environment, conditions include the entire detector description.

The org.lcsim conditions framework is designed to be very flexible, both in use and in implementation. Our current implementation looks up conditions based on the detector name.

> **When to use Conditions versus Drivers**
>
> The conditions framework should be used for information that is associated with one or more detectors. It should not be used for detector-independent settings on analysis algorithms, as this would force end users to create or locate a properties file for that algorithm. Instead, algorithm settings should use "getter" and "setter" methods on a Driver, e.g. **getX()** and **setX()**. This will allow these settings to be easily set from user code and from lcsim xml files.

Conditions Formats

The conditions framework API makes no assumptions about the format of the conditions data. However, the default implementation of the condition database assumes that the data is stored in zip files or file system directories. The package includes facilities for downloading zip files from the web and caching them, so no internet connection is required when running analysis or reconstruction jobs, as long as the required conditions have been previously accessed.

The conditions themselves are stored as an arbitrary collection of files, either stored in a filesystem or in a zip file. The most common files used are property files (name, value pairs) or XML files (e.g. for geometry). However, there are no constraints as to which file formats can be used for conditions information.

If storing conditions in zip files is not suitable for your project, there are two options. You can either implement the conditions framework interfaces from scratch, or use the standard implementation with custom conditions reader. If the *detector.properties* file in the conditions zip file or directory contains a line in the "ConditionsReader: *ClassName*" format, the framework will instantiate *ClassName* class and use it in place of the standard reader. The latter option makes it possible to keep some conditions in zip files while fetching others from a database, for example.

Properties Files

Properties files contain line-delimited key, value pairs with the 'equals' as a '=', ':' or ' ' (space). The file extension should be '.properties'.

For instance, any of the following will assign the value 'strange' to the key 'charm'.

```
charm = strange
charm: strange
charm  :strange
```

Multiple strings can also be associated with a single key, as follows.

```
charm    beauty, truth, strange
```

The value of the key 'charm' will be 'beauty, truth, strange'. These strings can be read as individual values using the method *String.split()* .

Accessing Conditions

For the following sections, we will use as an example the SDJan03 detector with corresponding tag of *sdjan03* for conditions lookup.

Detector Alias Files

Alias files are properties files, named **alias.properties**, that list the location of conditions information for specific detector tags.

Alias files have the the following format.

```
[detector_name]: [conditions]
```

The value of **conditions** can be one of the following.

1. Zip file on a website.
   ```
   sdjan03: http://www.example.com/path/to/sdjan03.zip
   ```
2. Local directory.
   ```
   sdjan03: file:/path/to/my/sdjan03/
   ```
3. Local zip file.
   ```
   sdjan03: file:/path/to/my/sdjan03.zip
   ```
4. Alias to another detector name.
   ```
   sdjan03_local: sdjan03
   ```

Multiple detector tags can be used used in the same **alias.properties** file, but each detector tag should eventually resolve to a single location.

Minimally, this directory or zip file should contain a file called **compact.xml** with the compact geometry description for the detector.

It may also contain sampling fractions, track smearing parameters, and other conditions for specific org.lcsim drivers.

Recursive Name Translation

If the value is not a URL, then it is assumed to be an alias. This is resolved recursively until the "base" value is obtained for that key.

Suppose the following is listed in the **alias.properties**.

```
a: b
b: c
c: http://www.example.com/d.zip
```

The final value of **a** will be

```
http://www.example.com/d.zip
```

The value after translation need not be a URL or file. It can be a detector tag.

In the following case, the value of **a** will be resolved to **d**.

```
a: b
b: c
c: d
```

The **d** name will be resolved using the conditions lookup algorithm.

Alias File Locations

Alias files are stored at one or more of the following locations.

1. The LCSim work directory in the user home directory.

```
~/.lcsim/alias.properties
```

2. Within the lcsim.jar file at the following path.
```
org/lcsim/detector/alias.properties
```
3. At a URL on the LCSim website.
```
http://www.lcsim.org/detectors/alias.properties
```

Your custom aliases belong at

```
~/.lcsim/alias.properties
```

as this is likely the only place to which you'll have write access, and your own aliases should be kept separate from the common ones.

On Windows, the default location of the .lcsim directory is usually

```
C:/Documents and Settings/username/.lcsim/
```

.

Conditions Lookup

Conditions information for a detector tag is retrieved as follows.

1. If the tag value is a URL with a protocol of **file**, the local file (zip format) or directory path specified is assumed to contain the conditions.
2. If the tag value is the URL of a zip file, an attempt is made to download the file from that location, unless the zip file is already in the local cache (~/.lcsim/cache). In this case, the local copy is used instead.

In the case that the tag value resolves to a detector tag, the following "canonical" locations are searched for a directory or zip file *with the same name as the tag*:

1. In the user home directory.
```
~/.lcsim/detectors
```
2. Within the lcsim.jar file.
```
/org/lcsim/detector/
```
3. On the LCSim website.
```
http://www.lcsim.org/detectors/
```

So the following locations would be scanned for *sdjan03* conditions.

1. Within the home directory.
```
~/.lcsim/detectors/sdjan03.zip
~/.lcsim/detectors/sdjan03/
```
2. In the jar file.
```
/org/lcsim/detector/sdjan03.zip
/org/lcsim/detector/sdjan03/
```
3. On the LCSim website.
```
http://www.lcsim.org/detectors/sdjan03.zip
```

If the lookup process does not result in a valid set of conditions, the program will terminate with an error. (In Java, this is a *ConditionsNotFoundException*).

Java Example

Here is an example of accessing conditions of the sdjan03 detector from Java code.

First, retrieve the default instance of the ConditionsManager.

```
ConditionsManager mgr = ConditionsManager.defaultInstance();
```

Then look up the conditions. The base location is http://www.lcsim.org/detectors/sdjan03.zip.

```
mgr.setDetector("sdjan03", 0);
```

Conditions are stored in sets, usually organized by single files or directories.

For example, sampling fractions can be found in the *SamplingFractions.properties* file, which is referred to as *SamplingFractions* when using the ConditionsManager.

```
ConditionsSet cs = mgr.getConditions("SamplingFractions");
```

Now, the sampling fractions are available by their keys.

This code simply iterates over the keys and prints their keys and values.

```
for ( Object o : cs.keySet() )
{
    System.out.println(o.toString() + "=" + cs.getString(o.toString()) );
}
```

Most likely, the typed values from the ConditionsSet need to be retrieved in order to do anything useful.

Here is an example showing how to convert conditions in a ConditionSet to their typed values, one-by-one.

```
for ( Object o : cs.keySet() )
{
    String k = (String) o;

    Class typ = cs.getType(k);

    if ( typ == double.class )
    {
        double dblVal = cs.getDouble(k);
    }
    else if ( typ == int.class )
    {
        int intVal = cs.getInt(k);
    }
    else if ( typ == java.lang.String.class )
    {
        String strVal = cs.getString(k);
    }
}
```

Presumably, an algorithm will do something with the value once it is retrieved.

Retrieving the Magnetic Field Information

The org.lcsim.geometry.FieldMap interface provides information about the magnetic field of the detector.

The B-field is retrievable from within a Driver's process method.

```
void process(EventHeader header)
{
    // Give a position for the B-field measurement to be retrieved.
    double[] pos = {0,0,0};

    // Provide your own b array to be filled.
    double [] b = {0,0,0};

    // Get the FieldMap from the detector.
    FieldMap field = event.getDetector().getFieldMap();

    // Fill your array with the B field data.
    field.getField(pos, b);

    // Print B field data to screen.
    System.out.println("bfield (" + b[0] + "," + b[1] + "," + b[2] + ")");
}
```

Setting the Cache Directory

Sometimes the default location of the user home directory is not a good place to download and store conditions information. For instance, grid jobs do not generally run under a user's credentials but a user name associated with the VO, so the default location will not work.

There are two ways to change the base cache directory, depending on the method being used to run lcsim.

If the lcsim jar is being run using the *java* command, set the system property **org.lcsim.cacheDir** to point to the directory where conditions information will be cached.

```
java -Dorg.lcsim.cacheDir=/my/cache/dir [...rest of options...]
```

The recon XML format provided by lcsim also has a setting for this.

```
<control>
    <cacheDirectory>/my/cache/directory</cacheDirectory>
</control>
```

The user's **alias.properties** file should also go in this directory instead of the home area.

Implementation Caveats

At the moment, there is a couple of implementation peculiarities you should be aware of.

The framework API lets you register listeners on ConditionsManager or on any Conditions, but the Conditions implementation simply forwards registration to the ConditionsManager. The result is that if, for example, you register your listener on two ConditionsSets, you will receive notifications of any conditions changes, not just those involving the two sets you are interested in, and you will receive them twice. For now, register all your listeners on the ConditionsManager to avoid this problem.

Once the conditions have changed, asking for data from an existing CachedConditions object will get you updated information. However, asking for data from an existing ConditionsSet will return old data - you have to get a new instance of the ConditionsSet from the ConditionsManager.