lcsim xml

Processing Data in Batch Mode using LCSim XML

Overview

If you have not gotten here by following the LCSim Tutorials, then you might want to backup and review, as necessary.

This tutorial explains how to run org.lcsim in a batch computing environment like a Unix command line or from a shell script that could be run on the Grid or your local batch computing system. The user provides what is typically called a "steering" file in HEP. It specifies all the parameters of the batch job. These steering files may have the extension *.xml*, but it is recommended to use *.lcsim* instead, to avoid ambiguity.

Running from the Command Line

Before starting you need to install org.lcsim on your local machine.

You can now run *lcsim* from the command-line using the *java* command.

```
cd trunk/distribution # where is your lcsim?
java -server -jar ./target/lcsim-distribution-[VERSION]-bin.jar myjob.lcsim
```

The **VERSION** is replaced by your lcsim build version to point to the actual "bin" file in your target directory.

The **myjob.lcsim** argument is an example name of a file in the lcsim reconstruction XML format.

Subsequently, in this documentation, the runnable jar will be referenced to as *lcsim-distribution-bin.jar* but the actual jar will have the version number in it.

LCSim Command Line Options

Running the jar without any arguments will print the usage instructions.

```
java -jar lcsim-distribution-bin.jar [options] steeringFile.xml
usage:
 -D    Define a variable with form [name]=[value]
 -n    Set the max number of events to process.
 -p    Load a properties file containing variable definitions
 -q    Turn on quiet mode.
 -s    Set the number of events to skip.
 -v    Turn on verbose mode
 -w    Rewrite the XML file with variables resolved
 -x    Perform a dry run which does not process events
```

These options should be mostly self-explanatory.

Variable Definitions

The LCSim XML format allows variables to be defined using the **-D** switch or within properties files specified by the **-p** option.

For instance, an LCIO input file could be defined using a variable.

```
<file>${inputFile}</file>
```

Then this file could be specified at the command line.

```
java -jar lcsim-distribution-bin.jar -DinputFile=myInputFile.slcio steeringFile.xml
```

This variable could also be set in a properties file.

```
java -jar lcsim-distribution-bin.jar -pmySettings.prop steeringFile.xml
```

The file *mySettings.prop* could contain the following.

```
inputFile=myInputFile.slcio
```

An unlimited number of definitions and properties files can be used.

Simple Job Example

Here is a simple example which will print the event number.

```
<lcsim xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
       xs:noNamespaceSchemaLocation="http://www.lcsim.org/schemas/lcsim/1.0/lcsim.xsd">
    <inputFiles>
        <file>./myEvents.slcio</file>
    </inputFiles>
    <control>
        <numberOfEvents>100</numberOfEvents>
    </control>
    <execute>
        <driver name="EventMarkerDriver"/>
    </execute>
    <drivers>
        <driver name="EventMarkerDriver"
                type="org.lcsim.job.EventMarkerDriver">
            <eventInterval>1</eventInterval>
        </driver>
    </drivers>
</lcsim>
```

The **inputFiles** section is a list of one or more LCIO input **file** paths that will be processed. There are actually multiple ways to specify input files (covered below).

The **control** section sets the jobs run parameters. Here we set the maximum **numberOfEvents** to 100.

The **execute** section is a list of drivers to be executed *in order*. The **name** field of the **driver** element must correspond with a valid driver.

Finally, the **drivers** section describes the drivers that will be run on the input file. Certain types of Driver parameters can be set in this section. Here the interval for event printing is set as **eventInterval**, which is an integer.

The signature for this Driver method looks like this.

```
public void setEventInterval(int eventInterval);
```

LCSim is able to convert from XML parameters to method calls on Drivers.

LCSim XML Format

The pseudo-XML below shows all possible elements in the LCSim format.

```
<lcsim xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
       xs:noNamespaceSchemaLocation="http://www.lcsim.org/schemas/lcsim/1.0/lcsim.xsd">
    <inputFiles>
        <fileUrl />
        <file />
        <fileSet>
            <file />
        </fileSet>
        <fileList />
        <fileUrlList />
        <fileRegExp />
    </inputFiles>
    <control>
        <dryRun>true</dryRun>
        <logFile>/path/to/mylog.txt</logFile>
        <cacheDirectory>/path/to/mycache/</cacheDirectory>
        <skipEvents>1</skipEvents>
        <numberOfEvents>1000</numberOfEvents>
        <verbose>true</verbose>
        <printDriverStatistics>true</printDriverStatistics>
        <printSystemProperties>true</printSystemProperties>
        <printUserClassPath>true</printUserClassPath>
        <printDriversDetailed>true</printDriversDetailed>
    </control>
    <classpath>
        <jarUrl />
        <jar />
        <directory />
    </classpath>
    <define>
        <anExampleVariable>1234</anExampleVarible>
    </define>
    <execute>
        <driver name="ExampleDriver" />
    </execute>
    <drivers>
        <driver name="ExampleDriver" type="org.lcsim.example.ExampleDriver">
            <exampleParam>1234</exampleParam>
            <exampleArrayParam>1 2 3 4</exampleParam>
            <exampleArray2DParam>1 2 3 4; 5 6 7 8</exampleArray2DParam>
        </driver>
    </drivers>
</lcsim>
```

The format is completely described by the LCSim XML Schema. At run-time, the actual schema is not read from the internet but from an embedded resource in the LCSim jar file. If your XML file does not follow this format, the job will fail, and a trace back will be printed with information about the error.

Input Files

The **<inputFiles>** section contains a list of local or remote files to be processed. It may contain a mixture of any of the elements described below, but it may not be empty. And it must result in at least one input file being found or the job will fail.

file

The **<file>** element is a relative or absolute path to a file on the local file system.

```
<inputFiles>
    <file>/path/to/local/datafile.slcio</file>
</inputFiles>
```

Or it may be a publically accessible URL.

```
<inputFiles>
    <file>ftp://example.org/datafile.slcio</file>
</inputFiles>
```

Some batch systems may not support remote file access via a URL. Check with your administrator.

These remote files will be downloaded to the cache directory, which is **~/.cache**, by default. A different local cache directory can be specified using the **<cacheDirectory>** tag in the **<control>** section.

fileSet

Sets of files on the local filesystem with the same base directory can be specified by using the **<fileSet>** element.

```
<fileSet baseDir="/my/data/dir">
    <file>events1.slcio</file>
    <file>events2.slcio</file>
</fileSet>
```

When processing these files, the base direcotry "/my/data/dir" will be prepended to each file to make a complete file path.

fileList

The **<fileList>** element should point to a text file containing a list of files, one per line.

For instance, say that you had a local text file at */example/mylciofiles.txt* containing paths to local LCIO files.

```
/my/data/dir/events1.slcio
/my/data/dir/events2.slcio
```

This can be fed into LCSim using this XML code.

```
<fileList>/example/mylciofiles.txt</fileList>
```

fileRegExp

The **<fileRegExp>** element will include files that match a regular expression.

Here is an example that would match files similar to input1.slcio, input2.slcio, etc. in the current directory.

```
<fileRegExp baseDir=".">input*[0-9].slcio</fileRegExp>
```

See http://docs.oracle.com/javase/tutorial/essential/regex/ for more information about regular expressions in Java.

Job Control

The **<control>** section contains parameters that control the batch job, including the number of events to run and whether various debugging output should be printed.

dryRun

Setting **<dryRun>** to true means that the job manager will create the drivers but will not run the job. This can be used to check that your driver setup and arguments are correct. No events will be processed when this argument is set to **true**.

logFile

The **<logFile>** element is used to specify a log file location. If no log file is specified, the job output goes to the terminal screen. The text needs to point to a valid path on the local file system.

cacheDirectory

The **<cacheDirectory>** specifies the root directory to be used for caching remote data files.

numberOfEvents

The **<numberOfEvents>** is the total number of events that will be run before the job ends. All events will be processed if this argument is left blank or if it is set to a negative number.

skipEvents

The **<skipEvents>** argument tells the job manager to skip a number of events up-front before processing the rest.

print

The "print" tags can also be set to true to print out additional information about the job: **<printDriverStatistics>**, **<printSystemProperties>**, **<printUserClassPath>**, **<printDriversDetailed>**, and **<printInputFiles>**. The meaning of each should be self-explanatory.

The following will turn on all verbose output but turn off the printing of the system properties.

```
<control>
    <verbose>true</verbose>
    <printSystemProperties>false</printSystemProperties>
<control>
```

The settings of individual "print" commands will always override the verbose setting for that particular print out.

verbose

The **<verbose>** tag should be set to true to enable verbose debugging output when the XML input file is processed. This turns on all of the "print" elements described above, which can still be turned off individually by setting them to false after verbose has been turned on.

Variable Definitions

The job manager has very limited support for "free" variable definitions, using the **<define>** block.

At the moment, this is limited to single doubles, which **can** include expressions to be evaluated.

Here is an example of a simple double parameter.

```
<define>
    <aDoubleParam>1.1</aDoubleParam>
</define>
```

Variables defined here can be included in expressions by using their name.

```
<define>
    <aDoubleParam1>1.1</aDoubleParam1>
    <aDoubleParam2>2.2</aDoubleParam2>
    <aDoubleParam3>aDoubleParam1 + aDoubleParam2</aDoubleParam3>
</define>
```

Variables defined here are also available when passing values to Drivers (covered in the next section).

Class Path

The **classpath** section is for adding external jar files that contain Driver classes.

Here is an example pointing to a (non-existant) jar at a URL.

```
<classpath>
    <jarUrl>http://www.example.org/something/myjar.jar</jarUrl>
</classpath>
```

The same thing can be done with local jar files and directories.

```
<classpath>
    <jar>/path/to/myjar.jar</jar>
    <directory>/path/to/myclassfiles</directory>
</classpath>
```

LCSim does not have the ability to determine the dependencies of the jar files listed here, so all required dependencies need to be included here.

Driver Execution

The **<execute>** section specifies the order in which the drivers will be called for each event. Each **<driver>** tag must have a unique **name** attribute value that matches the name of a driver defined in the **<drivers>** section (see next section).

Driver Definition

The **<drivers>** section contains definitions for all drivers that will be called in the job. These drivers need to be defined in the LCSim package jar or any of the jars in the **<classpath>**.

Driver Arguments

LCSim can convert XML text into arguments for Driver methods. Only method signatures with single arguments are supported, and there is a limited amount of types included in this binding.

Here is a table of supported parameter types.

type	array1d	array2d	expression
int	yes	yes	yes
String	yes	no	no
double	yes	yes	yes
float	yes	no	yes
boolean	yes	no	no
Hep3Vector	no	no	no
File	no	no	no
URL	no	no	no

Types with a "yes" in the *array1d* or *array2d* columns support arrays of those dimensions. Arrays beyond two dimensions are not supported and would need to be read in manually by user code, perhaps using a method with a File or URL argument. Types that support expression evaluation have a "yes" in the expression column.

Driver Example

The easiest way to understand how the driver parameter conversion works is to study an example.

Here is an example Driver class with a number of setter methods.

```
package org.lcsim.example;

public class MyDriver
{
    public void setX(int x);
    public void setX1(int[] x1);
    public void setX2(int[][] x)2;

    public void setFile(File f);
    public void setUrl(URL url);
    public void setVector(Hep3Vector vec);
}
```

Implementation of these methods, which would set private variables to the passed arguments, is left out for brevity.

This is the corresponding XML code in **<drivers>** that would pass values to each of these methods.

```
<driver name="MyDriver" type="org.lcsim.example.MyDriver">
    <x>1</x>
    <x1>1 2 3</x1>
    <x2>1 2 3; 4 5 6</x2>
    <file>/path/to/a/file.txt</file>
    <url>http://example.org/file.txt</url>
    <vector>1.0 2.0 3.0</vector>
</driver>
```

There are several important things to notice in this example.

The set methods are matched to parameter names by removing the "set" string from the method name and making the first letter of the parameter lower case. The Driver set methods **must** begin with "set", or they will be ignored and not matched with any input parameters.

Multi-dimensional arguments are space delimited, meaning String arguments should not have spaces.

The rows in 2D arrays are separated by semicolons.

In the above example, integers are used for the 1D and 2D arrays, but other types support arrays, also. See the types table for specifics.

Expression Evaluation

Simple expression evaluation is supported for a limited set of the supported parameter types, including int, double, and float, plus 1D or 2D arrays of these types. Supported symbols include *, /, +, (,), and -, which have their usual mathematical meaning, plus trig functions like sin and cos. Variables created in **<define>** can also be accessed by their name. Expressions may have units, also.

The GNU JEL library provides this capability. Refer to its documentation for further information on the expression format.

Units

LCSim supports the named units defined by CLHEP's SystemOfUnits.

The names of the units are the same, but the actual values may **not** be the same. For instance, in LCSim, the basic energy unit is GeV, whereas it is MeV in CLHEP.

Refer to the LCsim SystemOfUnits documentation to see which units are defined.

Guidelines for Creating Compatible Drivers

Drivers that will be accessed via an LCSim XML file need to follow these guidelines.

- The Driver class must be public.
- The Driver class must have a public constructor that takes no arguments.
- The Driver's constructor should not do any initialization. It should instead use the *detectorChanged()* or *startOfData()* methods, which are called after all input parameters are processed.
- The set methods to be accessed in the XML should always be of the form

  ```
  public void set[ParameterName]([type] [varName])
  ```

  Set methods not of this form will not be accessible as XML parameters.
- The use of sub-drivers is discouraged due to these being inaccessible by the XML format, though it is still possible to use them. Any dependence of a child Driver on its parent's XML input parameters can be handled by using the *startOfData()* method to add a new child Driver instance.

How to Run a Specific Release

You do not need to build lcsim yourself in order to run a specific release. The SLAC Nexus Repository can be searched for all lcsim-distribution releases which will display a table including downloadable links. The *bin.jar* links are the runnable jars which can be downloaded to your machine and run as per the above instructions.