LCIO
PERSISTENCY AND DATA MODEL FOR LC SIMULATION AND RECONSTRUCTION

F. Gaede, T. Behnke, H. Vogt, DESY, 22607 Hamburg, Germany
R. Cassell, N. Graf, T. Johnson, SLAC, Stanford CA 94309, USA

Abstract

LCIO is a persistency framework and data model for the International Linear Collider. Its original implementation, as presented at CHEP 2003, was focused on simulation studies. Since then the data model has been extended to also incorporate prototype test beam data, reconstruction and analysis. LCIO defines a common abstract user interface (API) in Java, C++ and Fortran in order to fulfill the needs of the global linear collider community. It is designed to be lightweight and flexible. We present the design and implementation of LCIO, focusing on the data model and new developments.

OVERVIEW

LCIO is a persistency framework for simulation and test beam studies for the International Linear Collider (ILC). It has first been presented at CHEP03 [1]. LCIO aims at allowing the exchange of data and algorithms among Linear Collider working groups and thus provide a basis for common software development.

Requirements

LCIO has to define a data model that fulfills the current needs of the global linear collider community for ongoing simulation and test beam studies. As Java, C++ and Fortran are used in ILC software, LCIO has to provide user interfaces in all three languages. In order to make it easy for existing frameworks to adopt LCIO it has to be lightweight and flexible without introducing additional dependencies on other software packages.

Implementation and Design

LCIO defines a common API for Java and C++ using the AID [3] tool. Two independent implementations exist for Java and C++ in order to benefit from either language's advantages. As is common practice today user code is completely separated from the actual I/O format and code, so more advanced formats can be incorporated in the future. In order to support legacy software a Fortran API to LCIO is provided through a set of wrapper functions to the C++ implementation. Details of the design and implementation are described in [1]. A schematic overview of the software architecture is shown in fig. 1

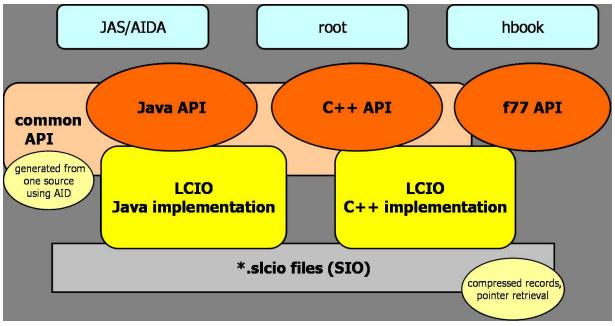

Figure 1: Schematic overview of the software architecture of LCIO. Two independent implementations in C++ and Java are used to provide common user interfaces in Java, C++ and Fortran thus making it easy to analyze LCIO data files with the analysis tool of choice.

I/O Format

SIO (*Simple Input Output*) has been chosen as a first I/O format for LCIO. It offers on the fly data compression and pointer retrieval. As SIO does not offer direct access functionality a fast skip mechanism is implemented in LCIO to allow reading of selected events only.

Users

A number of groups have chosen LCIO as their data model and format, some of which are: Mokka simulation and Brahms reconstruction [5, 6] , Lelaps fast Monte Carlo [7], hep.lcd reconstruction[1] [4], JAS3 analysis tool and WIRED event display [9, 8], TPC and Calorimeter test beam groups. Other groups have also expressed their willingness to use LCIO in the future. Thus LCIO is about to become a de-facto standard for ILC software development.

DATA MODEL

Figure 2 shows the event data model of LCIO. Central to the design is the class LCEvent that serves as a container for all data that are related to one event. It holds an arbitrary number of named collections (class LCCollection) of data objects (class LCObject). Run related information is stored in LCRunHeader. Run-headers, events and collections have an instance of LCParameters that allows to store

[1] currently being rewritten as *org.lcsim*

meta data (see section 'New Developments').

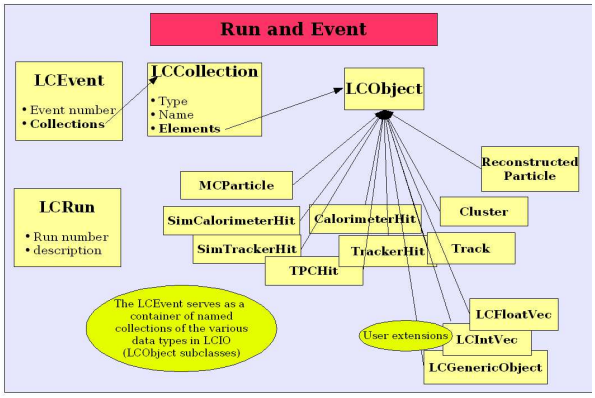

Figure 2: The main classes that define the event (and run) data model in LCIO. Event data is stored in collections of LCObject subclasses. Generic named parameters allow to store meta data for the run, event and collection.

In figure 3 the LCIO data entities are shown with the implemented relationships between the objects.

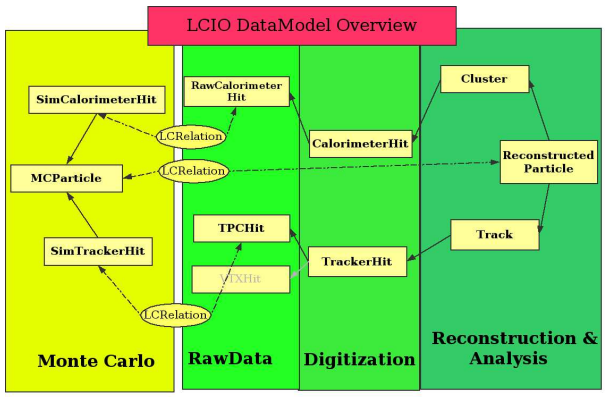

Figure 3: Schematic overview of the main data entities defined by the LCIO data model and the relationships between them. Objects are shown in the order of the processing flow, starting with pure Monte Carlo quantities on the left, followed by raw data and digitization to reconstruction and analysis classes on the right.

In the following sections we describe the entities that are defined at every processing level from the Monte Carlo generator to analysis.

Monte Carlo

The main class at the Monte Carlo level is MCParticle. There will be exactly one collection with name "MCParticle" in every event that holds the Monte Carlo truth particles as created by the generator program. Particles that are created during simulation will be added to the existing list of MCParticles. Adding particles with their correct lineage ceases when a particle decays or interacts in a non-tracking region. Otherwise the number of MCParticles would explode in calorimeter shower development. Two generic hit classes, one for tracker and one for calorimeter hits are used to store the simulated detector response. All energy depositions are assigned to particles in the list, i.e. particles seen in the tracking region or that entered the calorimeter and started a shower. If a particle from a calorimeter shower is scattered back into the tracking region it is also added the list with the resulting tracker hits assigned. A simulator status word is used to store the details about creation, interaction and decay of the particle. For the MCParticle only parent relationships are stored. When reading the data back from the file the daughter relationships are reconstructed from the parents. This is to ensure consistency. Care has to be taken when analyzing the particle 'tree'. Because a particle can have more than one parent the particle list in fact does not consist of a set of trees (one for each mother particle) but forms a 'directed acyclic graph'. Thus the user has to avoid double counting in his code. Of course this only matters at the parton level as real particles do not have more than one parent.

Raw Data

The classes at the raw data level have been introduced to make LCIO also suitable for storing 'real data' from test beam prototypes. For calorimeters there is the class RawCalorimeterHit that consists of an integer amplitude and a cell-id. It will most likely be suitable for all calorimeter type detectors. This is different from the situation for the tracking subdetectors. There the data formats will vary considerably with the particular type of the tracking device. Currently there is only the class TPCHit but others will be added as the need arises. If needed the raw data classes can also be used in simulation where one can link back to the Monte Carlo hits through special LCRelation objects.

Digitization

At the digitization level there are again two generic classes for tracker and calorimeter type subdetectors respectively. These classes will contain hit data after digitization and feature extraction. CalorimeterHit and TrackerHit are the types to be used in reconstruction and analysis code for Monte Carlo and test beam data. If needed users can access the corresponding original information for either data type.

Reconstruction and Analysis

Three classes are defined at the reconstruction and analysis level. Hits are combined into Clusters and Tracks by pattern recognition and reconstruction algorithms. Both classes point back to the contributing hits. Clusters can also be combined from other clusters allowing a tree-like structure, e.g. one could build clusters with a geometrical algo-

rithm and then combine some of these clusters to 'particles' applying some track-match criterion. Due to the imaging capabilities of the planned Linear Collider calorimeters, clusters have an intrinsic direction assigned to them. The following parameters are used for Tracks: **d0**: impact parameter of the track in the r-phi plane, **phi**: ϕ of the track at the reference point, **omega**: signed curvature of the track, **z0**: impact parameter of the track the in the r-z plane, **tanLambda**: λ is the dip angle of the track in the r-z plane at the reference point. By default LCIO tracks have the point of closest approach (PCA) as the reference point but any other point can be chosen as needed. ReconstructedParticle is the class to be used for every object that is reconstructed. This can be a single particle like a track identified as a pion or a compound object like a jet made from many particles. ReconstructedParticle has lists of Tracks, Clusters and ReconstructedParticles that have been combined to form this particle. Besides the kinematics including the corresponding covariance matrix any number of hypotheses describing the particle's identity (PID) can be stored. ReconstructedParticle is intended to be the working horse for most physics analyses, where only rarely the need arises to go back to tracks, clusters or even hits.

NEW DEVELOPMENTS

The first public version of LCIO (v01-00) has been released in Nov 2003. Since then the data model has been completed as described in section 'Data Model' and a number of new features have been developed and added to the current release (v01-03).

Relation Objects

In order to be able to relate objects with each other that do not have a build-in relation in the LCIO data model a new class LCRelation has been introduced. Typically LCRelation objects are used to store the links between raw data and the Monte Carlo truth information. This ensures that there is a clear separation between data classes that are to be used in analysis and reconstruction and the Monte Carlo classes that are used for developing and checking algorithms (see fig.3). LCRelation objects can also be used to relate reconstructed objects back to the Monte Carlo truth, e.g. if the hits are dropped from the files in order to save disk space. LCRelations might also be used to store transient links between objects at runtime.

User Extensions

The LCIO data model as described above has been designed in a way that it should fulfill all the current needs for ongoing Linear Collider studies. Care has been taken to make the data model flexible enough so that it can well be used for a variety of different subdetector types. Nevertheless users will occasionally need to store information that is specific to to a detailed aspect of their ongoing work and that is not foreseen in the data model. This can be done by using collections of LCIntVec, LCFloatVec and LCTStringVec objects, i.e. arbitrary vectors of type *int, float* and *string*. Even though this mechanism is fairly generic it can be become somewhat cumbersome to handle user extensions, in particular if the information at hand involves more than one data type.

A new interface LCGenericObject allows users to store any self defined class with LCIO that implements that interface. Every LCGenericObject has an arbitrary number of *int, float* and *double* attributes, where the numbers might be fixed among one collection or vary from object to object. Data stored in LCGenericObjects can be retrieved from an LCIO file either by using the user class implementation or through the generic interface. Thus to read an arbitrary LCIO file no additional knowledge or library is needed. This is different from other persistency systems, where typically a dictionary with the class definition is needed[2].

Transient Collections

The LCIO data model has been designed such that it can also serve as the transient data model in an application. Listener objects support a modular design of such applications, where every module gets an LCEvent with all the collections existing at that point and adds one or more collections with its result to the event. Typically anything added to the event will be made persistent. However some intermediate collections might only serve as input to a computation performed by a subsequent module. These LCColections can be flagged as transient and will not show up in the output stream.

Default Collections

The LCEvent allows to store an arbitrary number of named collections of the same data type. For example there can be several lists of Track objects available in one event. Typically there will be a collection of tracks or track segments for every tracking subdetector and one collection that holds all combined track fits. This collection is the one that will be used for most analyses. In order to make it simple for the user this collection can be flagged as being the default list for Tracks. In general there should be exactly one default collection for every type. This list should be complete in the sense that every known information has been taken into account and unique in the sense that it does not double count energy.

Meta Data

LCIO will be used by a number of groups to store data for detectors with different features and capabilities. Thus it is necessary to describe the data that is stored in LCIO in a way that it can be interpreted by users from other groups without additional documentation and ideally even

[2]This is in particular true for C++ systems where for Java based systems one could in principle retrieve and access data of unknown classes using reflection.

without modifying existing code. This meta data description can now be stored as arbitrary named parameters of types *int*, *float* and *string* attached to run-headers, events or collections. A number of predefined attribute names exist that are used to describe and interpret type information in LCObjects, such as the type of ReconstructedParticles or Clusters. These attributes can be parsed and interpreted by applications whereas other - user defined - attributes can at least be printed and interpreted by another user, making additional sources of documentation unnecessary.

SUMMARY AND OUTLOOK

Since its first public release in Nov, 2003 LCIO has been adopted be a number of groups and is about to become a defacto standard for ILC software. The current release v01-03 provides the complete data model including reconstruction objects and user extensions. Future releases will provide additional functionality that makes analyzing LCIO data more convenient.

Currently we are investigating the possibility to develop a glue layer for LCIO that allows to use the C++ implementation from Java code which in turn is called from within a C++ application. This would enable the development of a mixed language analysis and reconstruction framework. As there already is existing code in both languages such a framework would make it easy to compare and benchmark algorithms without the need to rewrite the code that only exists in the other language. Even though such a glue layer is in principle straight forward to implement, using e.g. JNI, in practice this might become tedious and error prone. However using automatic code generation one might overcome these problems.

REFERENCES

[1] F.Gaede, T.Behnke, N. Graf, T. Johnson *CHEP03 March24-28, 2003 La Jolla, USA* Conference proceedings, **TUKT001**, *arXiv:physics/0306114*.

[2] LCIO Homepage: `http://lcio.desy.de/`

[3] AID Homepage: `http://java.freehep.org/aid/index.html`

[4] hep.lcd Homepage: `http://www-sldnt.slac.stanford.edu/jas/Documentation/lcd`

[5] Mokka Homepage: `http://polywww.in2p3.fr/geant4/tesla/www/mokka/mokka.html`

[6] Brahms Homepage: `http://www-zeuthen.desy.de lc_repository/detector_simulation/dev/BRAHMS/readme.html`

[7] Lelaps Homepage: `http://lelaps.freehep.org`

[8] Wired Homepage: `http://wired.freehep.org`

[9] Jas Homepage: `http://jas.freehep.org`