ICEDUST: The Software for COMET

Ben Krikler
Imperial College London
(Dated: February 24, 2015(v1))

We cover an introduction to ICEDUST, with an overview of its structure and some of the key components. We then go through in more depth how the simulation and reconstruction stages work. This document should be periodically expanded and elaborated as the software is developed to act as a hand-book for new users.

Contents

I. HISTORY

Until recently, most simulations for the COMET experiment have been stand-alone with many people studying different aspects of the experiment separately. With data taking fast approaching, the decision to build a unified approach to the offline software was taken. In order to minimize the effort required to build such software, it was decided to base the COMET software on the framework used by ND280, the near detector for the T2K experiment [1]. ICEDUST, as the COMET framework is now known, has therefore inherited the general structure and low-level aspects (in particular, the online / offline data formats) from a framework that has been well tested on real data. ICEDUST Can Efficiently Do Useful Software Things and stands for the Integrated COMET Experiment Data User Software Toolkit.

II. ICEDUST

The ICEDUST framework splits up the code base into packages of which there are currently around 65. A package, in principle, has a single responsibility or task which it performs by providing a library, an executable or occasionaly both. Such responsibilities include defining an interface or data format, providing access to some external tool or library, and running some part of the complete processing chain.

The flow of information along the processing chain is shown in fig. 1. The naming of the packages should help to identify those that sit within the same collection. For example, the prefix 'Sim' means the package sits within the simulation collection, whereas 'Recon' means it belongs to reconstruction.

Packages that provide interfaces and data structures are considered 'lower level' and begin with the prefix 'oa' [1].

A Conventions document [2] sets out the full package naming scheme as well as other more detailed conventions for code and scripting style.

III. OBTAINING AND WORKING WITH THE CODE

A. Installing

B. Repository

C. Building (CMT)

IV. KEY CLASSES AND PACKAGES

When working with a framework it is important to understand the basic support structure and classes. In ICEDUST, most of these low-level packages are found in the 'oa' collection.

For instance, communication between packages is nearly entirely managed through a common file format known as 'oaEvent'. Since this format also contains the

[1] In ND280, this stood for 'off-axis', 'on-axis' or ***((CHECK: ...))*** . In COMET, we take it to be 'On Aluminium'

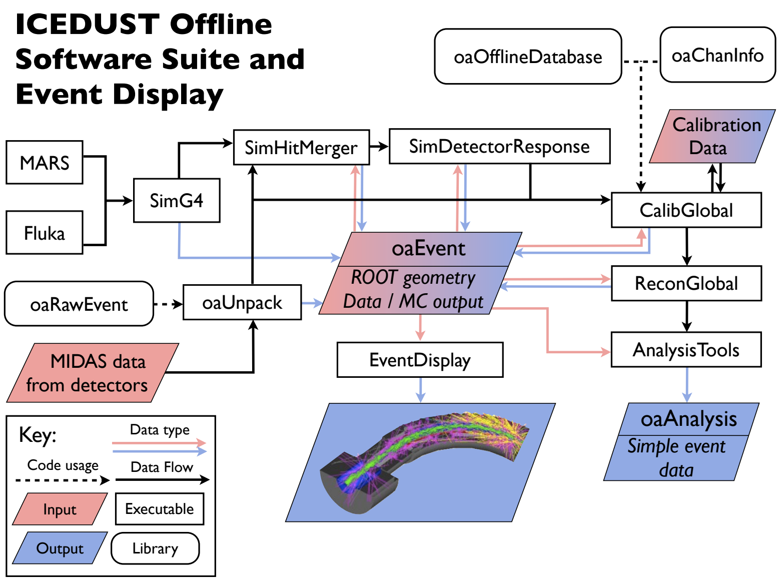

FIG. 1: The interaction between the key packages of the ICEDUST framework. Black arrows show the direction of information flow from one package to another. Blue arrows show the format of output data whereas pink arrows show the format of input data to a given package.

actual data tuples, there is less need for book-keeping since the data should be self-descriptive. An oaEvent file is built using a ROOT [3] TFile which contains conditions information, a representation of the geometry and the actual data itself which is stored in a TTree, giving a high level of compression and simple routines for data retrieval.

A. oaEvent

1. *Data*

Data in the the oaEvent format is built by a heirarchical structure of objects from the class called `IDatum`. This is a light-weight extension to ROOT's `TNamed` class that provides a stronger memory management policy and avoids the need for global pointers to access the output data hierarchy. Collections of data are stored in `IDataVectors` which derive from `IDatum` and can be used similarly to the STL's map class, with iterators and key-value access provided by the three methods `Has`, `Use` and `Get`.

The Doxygen documentation [4] provides more information on this.

2. *Geometry*

Geometries are stored alongside the data they relate to. Although this was inherited directly from ND280, it is particularly relevant for COMET since the exact geometry and setup will change a great deal between Phase-I and Phase-II and even within Phase-I there will be multiple setups with two detectors and different target and absorber configurations.

The geometry in oaEvent is stored using ROOT's TGeo format. This library provides methods to deal with material budget calculations, visualisation of the experiment and overlap checking. The exact geometries themselves are created in SimG4, the Geant4 package which shall be discussed in more depth later.

For real data, the geometry is actually persisted as a hash which is used to obtain the correct geometry file. The hash is formed using the SHA1 algorithm, so each hash is able to uniquely identify the geometry it corresponds to. These geometry files are pulled from a remote database at the start of every ICEDUST session and the correct geometry is loaded the first time the file is read in so that the difference between simulated and real data is again transparent.

3. *Conditions and Fieldmaps*

Finally, the header of oaEvent files contains other information describing the environment of the experiment. These include run, sub-run and event identifiers, as well as DAQ and Trigger configurations like timestamps and bit masks.

Slow control data is not contained in the oaEvent file but accessed through the oaSlowControlDatabase package.

Lastly, information on the fieldmap used for the data is not currently stored in the oaEvent file. For ND280 this was not a big issue as the fieldmap is not particularly complicated and is less critical to the experiment. For COMET on the other hand, knowing the field accurately across the entire experiment is both difficult, given the number and variety of solenoids, and important, since it directly relates to background and trigger rates.

Furthermore, as with the geometry, the field itself is likely to change quite a lot during the lifetime of the experiment. It is therefore important that this information be managed carefully and as such we plan on introducing into oaEvent a similar arrangement as for the geometry, where each fieldmap file is given a hash and stored in it's own ROOT format. These hashes and the various scaling and transformations that get applied to each fieldmap are then stored inside the oaEvent file so that the field that corresponds to a set of data is deducible from the data files themselves.

4. Reading and Writing with oaEvent

Given an oaEvent file, the simplest way to run over the contained data is by using the `cometEventLoop` functionality. This provides a simple way to produce a program, complete with command line arguments, that loops over every event in the file.

To work with this, create a class that inherits from `ICOMETEventLoopFunction` and overloads the `operator()` method, which takes an instance of COMETEvent. You can use the various getters for a COMETEvent to obtain the data for the current event.

A complete tutorial on the comet event loop approach can be found at: `http://www.hep.ph.ic.ac.uk/~bek07/comet//oaEvent/eventLoop.html`. For a full list of getters on the COMETEvent, you should also check the doxygen documentation.

If you want finer control over the event access, for instance if you do not want to just iterate over all events, then you can use the `COMETInput` class to handle reading from the oaEvent file. Give it the name of the root file or a pointer to the TFile that's been already opened and it will set up the internal methods to be able to get the geometry, the data and all the other contents of the file. You can then access any event using the `ReadEvent(int)` method or move between events with `NextEvent(int)`.

Similarly, for making oaEvent files, the best is to use the `COMETOutput` class. This makes it easier to write the geometry to file and fill the internal data tree with the output data.

B. Utility Classes

1. Logging

Finer control over input and output streams is provided within ICEDUST by the use of the `COMETLog` and `COMETError` macros. Since they are macros, they can be switched off at compile time so that they introduce no performance cost for production scale runs without the need to edit the code. The error macros also print the line and filename from where the error message originates which helps when trying to find the cause of the problem. There are several other related macros which will only output at higher verbosities.

The named versions of these macros allow for fine grained debugging of processes without the need to edit code. For example, the macro `COMETNamedTrace` takes two arguments, the name to associate this output to, and a streamable object that provides the message. The code for this looks like:

```
// From SimG4's
// COMETInteractiveCombinationParameter.cc
COMETNamedTrace("InteractiveCombo",
    "Making value for "
    << this->COMETVParameter::GetName());
```

Then in a file, which by default should be called 'comet.config' and sit in your current working directory, adding the line:

```
error.InteractiveCombo.level=TraceLevel
```

will set the error level for that named stream.

Possible log values, in order of increasing verbosity, are: QuietLevel, LogLevel, InfoLevel, VerboseLevel. Similarly error values, in order of increasing verbosity, are: SilentLevel, ErrorLevel, SevereLevel, WarnLevel, DebugLevel, TraceLevel. For more information on these macros, look in the source code at: `oaEvent/src/ICOMETLog.hxx` or at the Doxygen at [4]

2. Exception Handling

Exceptions are a very helpful way to flag up issues during run-time. ICEDUST provides a very convenient set of macros that make it easier to declare exceptions that are meaningful to the user, have a hierarhical categorisation and can even provide an on-the-spot backtrace.

To create exceptions with these macros in your code you add:

```
OA_EXCEPTION(EmyClass,EoaCore);
OA_EXCEPTION(ESpecificError,MyClass);
```

which creates two exception classes with the following inheritance heirarchy: `EoaCore` ← `EmyClass` ← `ESpecificError`. This heirarchical structure is useful when the exception gets thrown as it allows us to avoid name-clashes with similar exceptions for a different part of the code. To get these macros you will need to include the `EoaCore.hxx` header in your code.

When you want to throw one of the above exceptions, you simply call the constructor with no arguments:

```
throw ESpecificError();
```

The catching block can then get the exception name and backtrace just by calling `what()` on the caught exception. This will work in catch blocks for `std::exception`, since `EoaCore` derives from this. It is still useful to test first for `EoaCore` though since this will help differentiate exceptions from our code and those coming from elsewhere, such as `std::out_of_range` from trying to obtain an index beyond the limits of a given vector. A `try...catch` block for trapping these exceptions would have the form:

```
try{
  // something that may throw an exception
}catch( EoaCore& e){
  // Handle ICEDUST exceptions
  // Maybe just:
  COMETError( e.what() );
}catch( std::exception& e){
  // Handle std::exception
  COMETError( "Unknown problem: "<<e.what() );
}
```

3. Memory Management

Both a strength and a headache in c++ data analysis is the need to define your own memory management scheme. Normally this is stated that for every `new` operator, somewhere in the code there should be a `delete` operation, but implementing this is left up to the developer. In ICEDUST, a set of classes implement a different approach, known as reference counting. This is where an object is wrapped by a class who's destructor takes care of deleting the object once and only once all references to the object have finished with it.

This class goes by the name of `IHandle` and since it is designed to wrap any type of object, it is a template class. It can be used to replace code like:

```
ISomething* aThing = new ISomething();
```

with:

```
IHandle<ISomething> aThing(new ISomething());
```

The IHandle can then be treated as if it were a normal pointer:

```
// Implicit checking
if(aThing){
  // operator->
  aThing->DoSomething();
  // dereference / operator*
  ISomething aCopy = *aThing;
}
```

If the IHandle goes out of scope, the internal reference count is decremented. Once the count reaches 0 the contained memory is deallocated (`delete`d). If however you return the IHandle at the end of some function, which is then copied and used elsewhere, the internal reference counter will remain above 0 and the memory will remain allocated.

Warning: Reference counting comes with some drawbacks (check the web for examples). For instance, be careful not to make an IHandle to something that contains an IHandle to the first object. It may seem contrived but can occur easily when the reference counted objects are buried inside other objects creating a self-referencing loop (known as a reference cycle) which prevents the reference counters from ever reaching 0. Also be aware that many of ROOT's objects (such as TH1s and TTrees) provide their own memory management so deleting those objects yourself can cause memory upsets.

4. Runtime Parameters

The package oaRuntimeParameters provides a simple interface for reading in parameters from a text file. These can be used to control the way a program runs via a configuration file as well as through command line options. It also provides for the interpretation of many SI units.

Parameter files should sit in a directory called 'parameters' in a file named: `<packageName>.parameters.dat`. Within the file, parameters occur as key value pairs separated by ' = ' (a space, an equals sign and another space) and are contained in a pair of angle brackets. The key must always be prefixed by the package name and a full stop (which is used to identify the parameter file to use). For example, to set a parameter called 'MaxIterations' to the value '100' in a package called 'SolveComet' you would write `< SolveComet.MaxIterations = 100 >`. Outside of the angle brackets, the parser ignores whatever it finds so you can document the parameters very easily.

C. Running with IcedustControl

The IcedustControl package is a big python-based tool which allows a single user to run all the various packages in a single go. It is the main tool for production data and so manages all the various book-keeping of passing data along the processing chain. It's input is a text based 'cfg' file which the program uses to set up the control files for the individual stages. IcedustControl is quite high-level so that it can be run easily without an in-depth knowledge of the individual packages that it calls and the options and formats they require.

V. PROCESSING CHAIN

The general flow of data through ICEDUST is shown in fig. 2. Notably, data from the experiment and the simulation are processed in the same way, following the same steps through the chain.

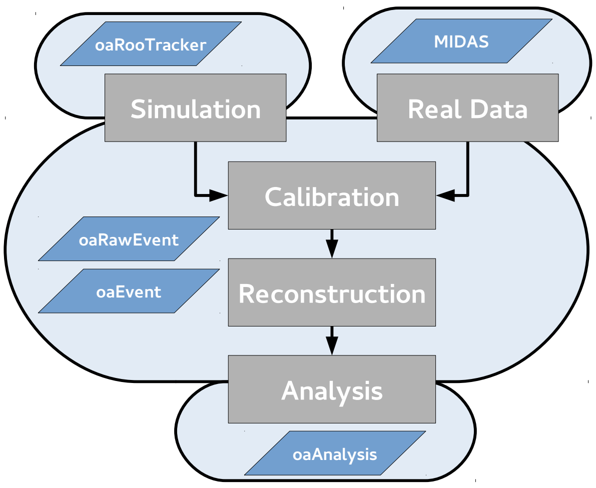

FIG. 2: The flow of data through ICEDUST. The larger blue regions represent parts of the framework that share a common data format, which is specified in the paralellogram.

A. Simulation

The ICEDUST simulation starts with a realistic description of the incoming proton beam and finishes with a set of data that should look like the outputs of all electronics channels. Fig. 3 shows the packages that play a role in the simulation and their order of processing.

The first step of simulating the particle disbtributions after the production target is crucial as this is a key factor in determining the total muon yield and radiation safety for COMET. Given that we will operate in a previously untested proton energy regime and with a novel capture mechanism [2] using hadronic physics models is a useful way to check the uncertainty of the different models. Currently we have 3 possible external libraries that can simulate this area: MARS [6], PHITS [7] and Geant4 [8]. In the past we have also supported Fluka [9] studies but support for this has not been maintained in ICEDUST. From this stage of the simulation a 'oaRooTracker' file is written which is a single ROOT TTree containing information on the individual particles produced at the target.

Outgoing particles from the production target are then fed into a Geant4 simulation where particles are tracked through to the sensitive detectors using Geant4. The tracking uses the magnetic field description to transport particles but also simulates the various material interactions and decays so that the beam composition changes in a realistic manner. This stage in the simulation produces several outputs, all of which are saved in an oaEvent file, alongside the geometry and other conditions. This includes truth information such as the actual trajectories of selected 'interesting' particles which allows one to reconstruct the decay chain that lead up to a hit. Whether or not a particle is interesting is determined by several factors including particle type, location, energy and whether or not it, or one of its decay products, has deposited energy in a sensitive detector. Such energy deposits are themselves also stored in a different container to the truth information and are known as G4Hits.

The Hit Merger package allows us to reshuffle events. This is one of the ways that we deal with backgrounds and signals which have a very low rate compared to most of the processes that Geant4 will have simulated. It works from a catalogue of events and is able to take G4Hits caused by different processes so that a rare background can be inserted over the standard 'noise' from simple beam simulations.

Finally, all the various G4Hits are processed to give realistic detector responses such as shaped electronic waveforms or hit information containing a single charge and time. To accuately simulate pile-up, this stage will need to combine hits from different primary proton events, produce outputs that are ordered in time and ideally maintain a list of the tracks that contributed to the Hit so that the various signals can be debugged and studied.

The DetectorResponse package is also tasked with running the trigger simulation, which is new to ICEDUST and not inherited from ND280 since the trigger there was simpler. Since this information will be used to determine the DAQ outputs, such as when to stop and start a waveform readout, this will have to be done in parallel or at least before the final Digitised readout can be produced.

Table I shows a summary of the data types, how they are persisted and where they are made and used.

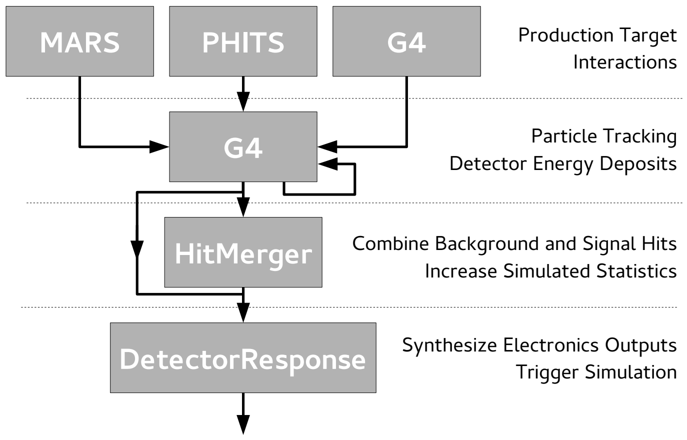
MARS
PHITS
G4
Production Target Interactions
G4
Particle Tracking Detector Energy Deposits
HitMerger
Combine Background and Signal Hits Increase Simulated Statistics
DetectorResponse
Synthesize Electronics Outputs Trigger Simulation

FIG. 3: The package structure making up the ICEDUST simulation. The collection's prefix ('Sim') is ommitted for simplicity. Each package provides an executable that operates on an input file from a previous part of the chain. SimG4 is capable of simulating the production target interactions directly using Geant4 [8], but given the uncertainty on the various models' validities for the COMET setup we also simulate this aspect using MARS [6] and PHITS [7].

[2] The MuSIC experiment at the RCNP in Osaka [5] uses a similar system of capturing backwards going pions and bent solenoids, but runs at 400 MeV proton energy whilst COMET will use 8 GeV.

Output	oaEvent Container	Made by	Used by	Description
Primary	primary, oaRooTracker*	Turtle, SimMARS, SimPHITS, SimG4	SimG4	Possibly an incoming proton beam description or products from the pion production target
Truth	truth	SimG4	Analysis	A linked list of particle trajectories, including positions, momenta, Particle ID (PID) and parent information
G4Hit	G4Hit	SimG4	SimHitMerger, SimDetectorResponse	A single energy deposit from one particle in one part of the detector
Waveform	waveforms, trigger_waveforms	SimDetectorResponse	Analysis	The raw waveform output from an electronic or trigger channel output. Since many detectors have two waveform streams (one for read-out, one for the trigger decisions) waveforms are used in two ways.
Trigger	triggers	SimDetectorResponse	Analysis	A timestamp with information about the pre-triggers that contributed
Digit	digits	SimDetectorResponse	Reconstruction	A synthesized waveform output from an electronic channel that has used trigger information
Hit	hits	SimDetectorResponse	Reconstruction	A single charge and time read-out that uses trigger information

TABLE I: The different data types persisted from the simulation. The final outputs, digits and hits, should be useable by the rest of ICEDUST as if it were real data. * Primary particles are initially stored in a separate file type known as a RooTracker file format, as well as being put into the oaEvent file.

B. Reconstruction

This is an area of ICEDUST that is rapidly being developed. The proposed package structure is shown in fig. 4. The intention is to have a single executable that runs detector specfic reconstruction over each subsystem, the details of which are implemented in separate packages. Each of these detector specific packages has access to a common set of algorithms for track finding, fitting and clustering.

Each stage of the reconstruction process will be designed to use multiple approaches in order to reduce systematic uncertainties and bias due to reconstruction. The persisted results will therefore use physical quantities so that the subsequent analysis can work independently from the choice of reconstruction algorithm.

Track fitting will need to consider multi-turn fitting in order to track particles to the hodoscopes and get the required resolution. Genfit is one of the external fitting packages that will be used, but requires some modification in order to perform multi-turn fitting.

[1] K. Abe, N. Abgrall et al. “The T2K experiment”. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 659(1):(2011) 106 – 135. doi:10.1016/j.nima.2011.06.067.

[2] B. Krikler, A. Kurup et al. “ICEDUST Conventions”. *Internal Document*.

[3] R. Brun and F. Rademakers. “ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP’96 Workshop, Lausanne, Sep. 1996,”. *Nuclear Instruments and Methods in Physics Research Section A*, 81.

[4] oaEvent Doxygen. “`http://www.hep.ph.ic.ac.uk/~bek07/comet/oaEvent/index.html`”.

[5] Y. Hino, Y. Kuno et al. “A Highly intense DC muon source, MuSIC and muon CLFV search”. *Nuclear Physics B (Proceedings Supplements)*, 253-255:(2014) 206–207. doi:10.1016/j.nuclphysbps.2014.09.051.

[6] N. V. Mokhov. “The MARS code system user’s guide version 13(95)”.

[7] H. Iwase, K. Niita and T. Nakamura. “Development of general-purpose particle and heavy ion transport Monte Carlo code”. *Journal of Nuclear Science and Technology*, 39(11):(2002) 1142–1151.

[8] S. Agostinelli, J. Allison et al. “Geant4 - a simulation toolkit”. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):(2003) 250 – 303. doi:10.1016/S0168-9002(03)01368-8.

[9] A. Ferrari, P. R. Sala et al. “FLUKA: A multi-particle transport code (Program version 2005)”.

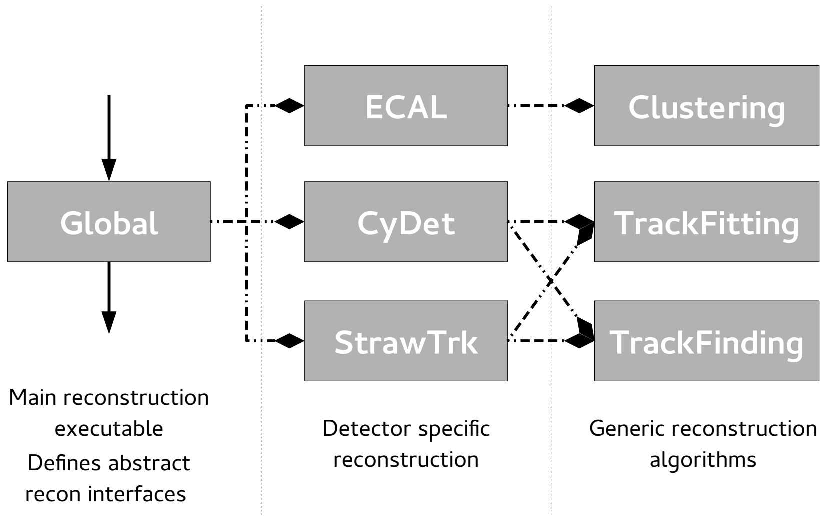
ECAL
Clustering
Global
CyDet
TrackFitting
StrawTrk
TrackFinding
Main reconstruction executable
Defines abstract recon interfaces
Detector specific reconstruction
Generic reconstruction algorithms

FIG. 4: The package structure and communication for the ICEDUST reconstruction. The collection's prefix ('Recon') is ommitted for simplicity. Only the ReconGlobal package produces an exectuable, the other packages provide libraries that are compiled in to the executable.