# ICETRAY: A SOFTWARE FRAMEWORK FOR ICECUBE

T. DeYoung*, University of Maryland, College Park, MD 20742, USA
For the IceCube Collaboration

*Abstract*

IceCube is a cubic kilometre scale neutrino telescope under construction at the South Pole. The minimalistic nature of the instrument poses several challenges for the software framework. The IceCube collaboration has developed the IceTray framework, which meets these needs by blending aspects of push- and pull-based architectures to produce a highly modular system which nevertheless allows each software component a significant degree of control over the execution flow.

## THE ICECUBE DETECTOR

IceCube is designed to detect high energy (TeV – EeV) neutrinos from astrophysical sources. The deep detector will consist of 4800 digital optical modules deployed in a three-dimensional array of 80 strings buried at depths of 1450-2450 meters in the South Polar ice cap. Neutrinos that undergo charged-current interactions in the ice are detected via the Cherenkov light emitted by the resultant ultrarelativistic leptons. To maximize neutrino event rates, the optical modules are spaced over a cubic kilometre, instrumenting a gigaton of the ice. An array of 80 pairs of ice Cherenkov tanks on the surface, with one pair near the top of each string, will operate in coincidence with the deep detector for calibration, identification of backgrounds and related cosmic-ray physics. The primary backgrounds are penetrating muons produced in cosmic ray air showers above the South Pole, and 'atmospheric' neutrinos produced by meson decay in air showers anywhere in the atmosphere. These latter events also form a useful low-energy calibration beam. Further details regarding the IceCube detector are available in [1].

### *Requirements for the IceCube Framework*

The nature of the IceCube experiment is rather different from those in traditional high-energy physics, leading to some differences in the requirements for and constraints upon the experiment software. Nevertheless, many of the requirements are the same, and we believe that some aspects of the IceCube software may be of interest to the wider HEP community.

Firstly, the sparse IceCube detector produces much less information in an event than does a collider or fixed-target detector. A full IceCube event is expected to average between 1 and 2 kB, and the detector will trigger at about 1.7 kHz when fully operational. The size and performance constraints that have led many HEP experiments to develop data-on-demand systems and hierarchical levels of event representations are therefore absent in IceCube.

The sparse event data, however, also means that event reconstruction is quite difficult. With only minimal information from the detector, relatively sophisticated (and therefore time-consuming) methods must be used to reconstruct each event [2]. Moreover, events must be reconstructed many times with different hypotheses or methods, and the results compared. One wishes to have the capability to configure the reconstruction application so that different methods are applied or not, based on the results of previous reconstructions. The appropriate series of software components required to process an event varies considerably, and can be determined only at run time.

Furthermore, reconstruction algorithms are an area of very active development. The collaboration numbers about 150 scientists, and it is expected that many collaborators will take an active role in software development. However, the South Pole Station is inaccessible for most of the year, with only two winter-over IceCube scientists, who may not be software experts, remaining on-site. Satellite coverage (and thus network connectivity) is also quite limited, so installation of new online software may be conveniently performed only in a short window once a year. Together, these constraints make it imperative that a single software framework be used both online and offline, to minimize if not eliminate problems with installation of new online software. Moreover, that framework must be simple enough to learn and easy to use that it is the platform of choice even for non-experts for 'private' analysis code (which may be next year's online software).

A second fundamental difference is that there is no experimental 'heartbeat' similar to the beam-crossing time in a collider. Events occur at random times, and frequently overlap. This poses a problem for simulation in particular, since the time window that is to be simulated is not known in advance: the window may need to be extended repeatedly until 'quiet intervals' are found on either side of the window. Furthermore, the types of events which are to be overlaid may be different – for example, neutrino events and cosmic ray showers – and so the simulation must be able to synthesize the outputs of several long simulation chains in a 'Y' topology, as shown in Figure 1.

Another simulation requirement comes from the fact that it would be extremely time-consuming to track every photon produced by an ultrarelativistic lepton through a cubic kilometre active volume. Instead, a photon

*deyoung@icecube.umd.edu

propagation code is run in advance and the results tabulated for quick use during simulation production. These photon tables, however, are very large compared to the memory available on a typical processing node, so the simulation design calls for events to be held in a buffer and sorted for more efficient table access. A similar approach may be employed in some reconstruction programs, as well. This buffering further breaks down the traditional event-based processing paradigm.

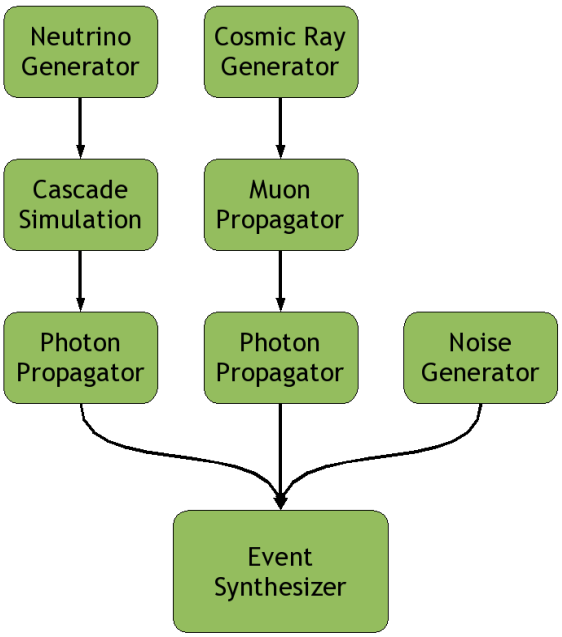


Figure 1: An example simulation application.

We wish to accommodate all of these requirements in the context of a modern software framework, with its advantages of flexibility, modularity, reusability, reliability, and commonality of software. However, simplicity is also of critical importance.

## THE ICETRAY FRAMEWORK

The variability of the event processing foreseen for IceCube implies that the traditional declarative ordering, where a series of processing steps are set up in advance and an event 'pushed' through them, is not well suited to the problem. The desired event processing will depend on the results of steps within the chain, and therefore cannot be declared in advance. Moreover, these results may not be available until after the steps would normally have been taken: in simulation, for example, whether an event from the cosmic ray simulation branch will be needed by the software module responsible for overlaying events cannot be determined until that module has received the event from the neutrino branch.

It would, of course, be possible to develop a workable declarative system to meet these requirements, relying on some type of mechanism for passing signals between modules or between modules and framework and possibly on preprocessing of some sort. However, this would impose additional complexity on both the software modules and on the framework, and possibly on the user or batch production system as well. It would also lead to some level of indirect coupling between otherwise independent modules, which would need to understand each other's signalling mechanisms or conventions.

The other paradigm which has become popular in recent years is "processing on demand" or discovery ordering, in which a module requests a particular type of data and relies on the framework to find a module which can provide it, so that data is 'pulled' through the chain by the final client. This requires both some system for modules to register the type of data they produce, and that client modules understand their place in the overall application. While this may be unavoidable or even desirable in the context of an accelerator-based experiment, with very large events consisting of many subtypes of data and with relatively well-established methods for processing that data, it is not necessarily appropriate to a neutrino telescope. In that case, the event size is small, so that a complete event record is always being supplied; but the processing methods are evolving rapidly, so it is quite likely that the developer of a given module may not know, when the module is being written, what steps should be taken before the module is executed.

Again, it is certainly possible to develop a pull-type system which would meet the needs of IceCube. However, such a system would, again, add complexity to the framework. It would also require some implicit coupling of modules, which would be required to know the context in which they were processing events in order to properly declare the "meaning" of the data they produced.

### *IceTray Execution Ordering*

The solution developed for the IceTray framework blends aspects of both declarative and discovery ordering. We rely primarily on a data-driven order of execution, similar to a push system. However, instead of a linear processing chain, we allow modules to send their output to one of several output queues, called *Outboxes*. Extremely complex chains of logic for how events should be processed can then be set up topologically, by connecting one module's outboxes to another's input queue (or *Inbox*). A simple example of such a chain is shown in Figure 2.

This topological system is completely transparent to each of the modules involved: rather than relying on signalling between modules or markers added to events, each module is assured that it should process any event presented to it. In fact, a module can play one role in one application, and a quite different role in another; the 'identity' of the outboxes suggested by the labels in Figure 2 is entirely implicit in the configuration of the chain. From the perspective of the individual module, the various queues may be much more generic; a muon reconstruction module may have an outbox for events in which the reconstruction failed to converge, which is connected by the user to some other reconstruction chain

(*e.g.* the 'Misidentified Showers' of Figure 2). This allows the application user to choose at configuration time what classes of input a module will receive, rather than requiring the module developer to specify the desired input at compile time, and therefore increases the flexibility and reusability of the module.

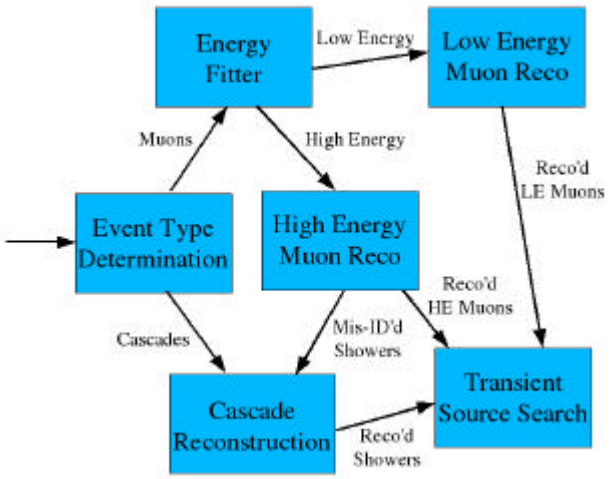


Figure 2: An example IceTray reconstruction application. The arrows show possible routes events can take through the chain of modules, shown as blue squares.

This topological execution ordering is also quite straightforward from the framework's perspective; events need only be transported from outbox to inbox in a perfectly predictable fashion. The responsibility for ensuring correct data flow is on the user, who is essentially giving the framework a flowchart at configuration time. (Presently configuration is handled via a Root macro; other user interfaces, including a GUI, are envisioned.) Besides reducing the load on both framework and module developers and ensuring maximum flexibility for module reuse, this corresponds to the expressed preferences of IceCube collaborators in regard to the proper division of responsibilities between framework developers, module developers, and end users.

Within this topological paradigm, it is relatively easy to accommodate the other requirements for the IceCube framework. As mentioned above, the inboxes and outboxes of a module are queues; there is no requirement that an event be immediately processed once it appears in an inbox. The module is free to defer processing, buffering events for more efficient access to resources such as the photon simulation tables. The in- and outboxes are not actually connected, but instead the framework transports events between the appropriate pairs. This allows the framework to maintain the correct event ordering, as well as allowing applications to run in a distributed fashion on multiple nodes transparently to the module.

Finally, the merging of event streams is handled in a pull-type fashion. In addition to multiple outboxes (which are filled at the module's discretion), modules are allowed to have multiple inboxes. Of these, one is the primary or *active* inbox, and module execution is triggered by the appearance of an event in that queue. The other inboxes are *passive*, meaning that events accumulate there until needed by the module in the course of processing. The module can then pull events out of the passive inbox at will.

## *Execution Environment*

It is foreseen that IceTray applications will run in a variety of environments, from the user's desktop to a production farm to the online environment at the South Pole. It is essential that application modules can be moved transparently between these environments and run identically without modification. For this reason, all module interactions with the outside environment are mediated by the framework via a *Context* object which is passed to the module at instantiation.

The IceTray module thus exists within an *analysis container*, isolated from its surroundings except for the framework. Within this container, the application appears to have its own flow-of-control, which simplifies the development process. Interactions with the framework are relatively simple, including requests for events or services such as random number generators, configuration, logging, and error conditions. It is up to the framework to handle these requests, and different implementations of these services may be provided via the context object in different environments. The module does not know, for example, whether an event in its inbox has come from another module within the same IceTray process or over the network from a remote node, nor does it know where the event will go once it is placed in the outbox.

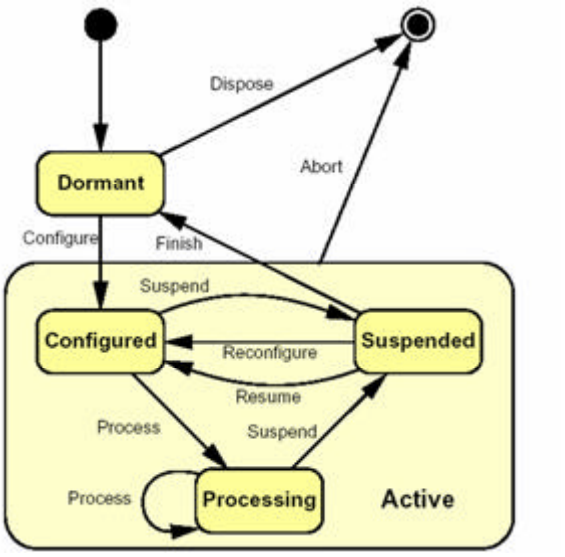


Figure 3: IceTray module state machine diagram.

In order to accommodate these various environments, an IceTray module is developed as a relatively simple yet flexible state machine, as shown in Figure 3. Default implementations of all of these transitions are provided

through a base Module class, which is inherited by all application modules. Module developers implement only those state machine transitions which are necessary; most simple event processing modules implement only the Configure and Process transitions. A module which interacts with external resources such as databases may also require Suspend and Resume in order to avoid resource blocking, while modules responsible for data monitoring may wish to use the Finish transition to package or record their accumulated data. Finally, Reconfigure is available for interactive processes.

## SUMMARY

The IceCube collaboration has developed the IceTray framework to meet its requirements for online and offline data processing. IceTray incorporates aspects of both declarative and discovery processing frameworks in a topological structure that is both intuitive and highly flexible, while remaining almost completely transparent to the individual software modules. Within this paradigm, IceTray provides for features not usually found in HEP frameworks, such as event buffering at the discretion of the individual module.

By wrapping individual modules in analysis containers, the framework will provide for modules to be redeployed without modification in a variety of environments, from the single-node user analysis to production or online processing. However, the framework-provided module base classes relieve most module developers of the burden of understanding the details of the module life-cycle, so that the typical collaboration member can contribute to production software.

Overall, IceTray reflects the division of responsibilities requested by members of the IceCube collaboration. The end user is easily able to reconfigure and rearrange modules to create new applications or add new processing steps to existing applications, and is responsible for providing a reasonable configuration. The module developer has a straightforward environment in which to write his or her code, with a minimal burden in terms of receiving events, getting configuration information, accessing services, and writing logging information. This allows the developer to focus on the task at hand, and will enable all collaborators to contribute to the experiment software. Finally, the framework is responsible for the underlying tasks of managing execution flow and data flow, providing services, and the like, which will allow modifications for adaptation to distributed and online environments to be handled entirely at the framework level.

## ACKNOWLEDGEMENTS

This research was supported by the following agencies: National Science Foundation – Office of Polar Programs, National Science Foundation – Physics Division, University of Wisconsin Alumni Research Foundation, Department of Energy, and National Energy Research Scientific Computing Center (supported by the Office of Energy Research of the Department of Energy), UC-Irvine AENEAS Supercomputer Facility, USA; Swedish Research Council, Swedish Polar Research Secretariat, and Knut and Alice Wallenberg Foundation, Sweden; German Ministry for Education and Research, Deutsche Forschungsgemeinschaft (DFG), Germany; Fund for Scientific Research (FNRS-FWO), Flanders Institute to encourage scientific and technological research in industry (IWT), and Belgian Federal Office for Scientific, Technical and Cultural affairs (OSTC), Belgium; FPVI, Venezuela.

## REFERENCES

[1] J. Ahrens *et al*. (the IceCube Collaboration), Astroparticle Physics 20/5 (2004) 507.

[2] J. Ahrens *et al*. (the AMANDA Collaboration), Nuclear Instrumentation & Methods A524 (2004) 169.