The art framework

C Green, J Kowalkowski, M Paterno, M Fischler, L Garren and Q Lu

Abstract. Future "Intensity Frontier" experiments at Fermilab are likely to be conducted by smaller collaborations, with fewer scientists, than is the case for recent "Energy Frontier" experiments. **art** is a C++ event-processing framework designed with the needs of such experiments in mind. An evolution from the framework of the CMS experiment, **art** was designed and implemented to be usable by multiple experiments without imposing undue maintenance effort requirements on either the **art** developers or experiments using it. We describe the key requirements and features of **art** and the rationale behind evolutionary changes, additions and simplifications with respect to the CMS framework. In addition, our package distribution system and our collaborative model with respect to the multiple experiments using **art** helps keep the maintenance burden low. We also describe in-progress and future enhancements to the framework, including strategies we are using to allow multi-threaded use of the **art** framework in today's multi- and many-core environments.

1. Introduction

art[1] is an *event-processing framework*. In the context of the experiments using **art**, an *event* is all the relevant data describing what happened in a particular time period of interest. In the case of a collider experiment, this is one beam crossing (which may represent multiple particle collisions). For Intensity Frontier experiments, the definition of the event is somewhat less obvious. Each experiment that uses **art** defines its appropriate period of interest.

The authors have been involved with the design and implementation of event-processing frameworks for several experiments, including DØ, BTeV, CMS and MiniBooNE. Although many of these experiments' framework requirements were substantially similar, they shared little development effort, and even less code. This resulted in significant duplication of effort. The **art** framework project is intended to avoid such duplication of effort for the experiments planned, and under consideration, at Fermilab.

The **art** framework began as an evolution of the framework of the CMS experiment[2], and has since been substantially adapted for the needs of the Intensity Frontier experiments. Trade-offs have been made to simplify the code in order for it to be maintainable and usable by much smaller groups. The current users of **art** include Mu2e, NOνA, g-2, and LArSoft (ArgoNeuT, μBooNE, and LBNE). Its use is being considered by other groups.

Because of the increasing importance of multi-core and many-core architectures, current development plans center around the migration of **art** to support parallel processing of independent events as well as to permit parallel processing within events.

2. Requirements

There are several key requirements that drove the overall design and philosophy of **art** and its precursor, the CMS framework. The choice of programming language, for example, was

an experimental requirement. Several of the developers have been involved in the past with experiments that required the framework allow domain experts to write analysis code in C++ or Fortran, but that was not the case for Intensity Frontier experiments. The modular nature of the framework is a consequence of the need to be able to handle substantially different categories of task (*e.g.*, simulation, production reconstruction or user analysis) without recompilation or undue duplication of code or infrastructure. The nature of HEP data to be organized into time-contiguous events and groups thereof (*subrun*, *run*) during data acquisition drives a similar philosophy with respect to the organization of the framework. Conversely, the need for certain information to be accessible across these boundaries leads to the concept of a *service*, to be explained below.

One major requirement made as a result of missteps in the past is that of reproducibility, which in this case can be refined into a need to understand how a particular result was obtained and the ability to marshal the same input data and processing steps to obtain the same result when required[1]. This leads to secondary requirements on configuration, metadata storage, inter-module communication, temporary and persistent data structures and (to some extent) error handling which will be elucidated in section 3.

It was felt by the initial client experiments of **art** that the use of a general-purpose language such as **Python** or **Tcl** to specify configuration represented more flexibility than was necessary, and thus complication that was not needed. Non-expert users likely would require more training and support to find and fix problems. Additionally, allowing too much flexibility potentially could compromise the requirement of reproducibility. This requirement drove the creation of a domain-specific language **FHiCL**[3] to handle the representation of hierarchical configurations which has found use outside **art**, C++ and HEP.

Another requirement is that the framework be capable of data output in a form that can be read into **ROOT**[4] in a manner convenient for end-user analysis and graphical representation. **art** must also be usable as an external product in order to avoid constraining an experiment to use a particular build system. Other, more specific requirements from experiments in the form of feature requests are discussed in section 3.

3. Architecture and Key Features.

An **art** program reads a sequence of events from some user-specified input source, invokes some number of user-specified *modules* to perform reconstruction, analysis, simulation, or event-filtering tasks, and may then write out the results to one or more files.

A *module* is a configurable unit of (usually) user-written code, with characteristics and entry points defined by its inheritance from one of the classes `InputModule`[2], `OutputModule`, `Producer`, `Filter` or `Analyzer`. The latter three are the module types used to implement the experiments algorithms.

A *service* is a configurable utility class registered with the framework whose methods may be accessed from modules via a `ServiceHandle`. It registers optionally one or more callbacks with an `ActivityRegistry` to enable behavior to be triggered at various points in the application workflow. The service system controls the time of creation and destruction of service objects, as well as assuring that the correct number of service objects are available.

Both modules and services are examples of *plug-ins*, which means they are classes that reside in dynamically-loaded libraries which are made available to a program only when the configuration file for that program names them. A simple naming scheme for the libraries in which these classes are found is used to allow the framework to identify which libraries need to be loaded for each program execution.

[1] As long as the same computing hardware and operating system is available, bit-wise reproducibility should be possible.

[2] All **art** classes are defined in the namespace `art`; we omit the namespace qualification in this paper for brevity.

The flow control of an **art** application is managed by a state machine, the `EventProcessor`. This handles initialization of the main application and the invocation of service callbacks and of the various stages of each module at the appropriate time.

The configuration of the application, including global behavior, the modules and services to be invoked and their particular configurations, and the order of execution of modules is specified using the **FHiCL** language. This is a simple language for describing configuration data: it describes hierarchical data structures and provides facilities to ensure that parts of such structures that are intended to be identical are identical. It also allows users to define a configuration that differs in specific ways from an experiment-provided "default" configuration. The hierarchical nature of configuration language allows the partitioning of configuration data with different purposes. An example snippet of **FHiCL** code is shown in figure 1.

```
#include "more_prolog.fcl"
BEGIN_PROLOG
stashed_parameter: true # Save this for later
END_PROLOG

process_name: RECO

source:
{
  module_type: RootInput
  fileNames: [ "f1.root", "f2.root" ]
}

scheduler.services: { @local::default_services }

physics.producers:
{
  trk1:
  {
    module_type: JetFinderA
    cone_size: 0.2
  }
}
...
```

Figure 1. An example of **FHiCL** configuration code.

Modules communicate with each other by way of *products*, which should be simple concrete classes (or collections thereof) for which a **ROOT** dictionary may be created. If persistency is required (recommended for reproducibility) the product must additionally avoid use of pointers or pointer-containing classes[3]. A module will declare a product to be placed into one of three entities of well-defined lifetime: the *event*, the *subrun* or the *run* upon construction. At the appropriate time, the module will place (or not) said object into the correct containing entity. Once a product has been placed into the event[4], it should be considered immutable: subsequent modules have only **const** access to products they retrieve. Derived or edited data must be

[3] Certain exceptions are permitted, such as `std::vector` which is serialized specifically by **ROOT**.

[4] Products can also be placed into a subrun or run.

saved as new product instances. One easily-missed but important attribute of this system is that products are separated from the algorithms that produce or consume them: algorithm code does not address the mechanics of persistency.

Products or objects within them may refer to other objects in a persistent collection by way of the `Ptr` class template. Alternatively, a bi-directional association may be created between two objects in collections already in the event using the `Assns` class template.

Exception handling is configurable: **art** or experiment-authored code throws exceptions which define the category of error that has been encountered. This categorization is distinct from the specification of the handling action. There is no "fatal exception"; rather, the framework can be configured so that the reaction to a given exception is to shut the program down (gracefully), or to continue with the next event, or one of several other choices of handling action.

Product mixing is possible: the ability to combine many instances of a product from a secondary input stream into the current event (*e.g.* to handle "pile-up" or cosmic rays) is supported by a module class template, `MixFilter`. Users implement the "detail" class that describes how multiple instances of one or more product types should be combined into single instances of each product. The class template takes care of the mechanics of obtaining the data from auxiliary files and for writing out the merged product.

In addition to the persisted data products, the configuration used is saved in a **SQLite**[5] relational database, which is embedded in the **art** output file. In addition, product provenance and other metadata are saved to enable experiments to track the steps that went into producing each data product.

A class template for writing input sources is provided (`ReaderSource`) which allows experiments to craft inputs that read specially formatted files (*e.g.* DAQ output) without needing to understand the complexities of how the framework interacts with the input system; the experiment's code is only responsible for delivering and populating new events, subruns, and runs as the input is read.

The **art** framework relies upon a number of external products (*e.g.*, the **Boost** C++ library[6] and **ROOT**); these products are built by the **art** team and deployed through a simplified **UPS**[7] package deployment system. The **art** framework is itself deployed via the same mechanism, and is treated by the experiments using it as just another external product upon which their code relies.

4. Collaborative development and deployment

The "**art** suite" is a collection of software packages, currently five in number, developed and maintained by the Scientific Software Infrastructure group of the Scientific Computing Division at Fermilab. The source code for each of the component packages resides in its own **git** repository linked to a **Redmine** project hosted by Fermilab's central facility for that purpose. In this way repository browsing, a release roadmap, wiki documentation and issue tracking are made available with zero developer effort beyond the actual authorship of the written material provided.

A small team of developers supports and extends **art**. The development team conducts weekly "stakeholder" meetings to obtain input on requirements, issues and priorities from experimental representatives and other interested individuals. In addition to the issue section of **art**'s **Redmine** page, mailing lists for community support and for developer advice are available for experimenters to use.

In contrast to previous frameworks, experiments are not constrained to use a particular build or source control system in order to use **art**. The **art** suite packages are developed, built and packaged using **cetbuildtools**[8], a **CMake**[9]-based build system produced by the authors of **art**. Experiments using **art** however, use a variety of build systems including **SoftRelTools**[10], **SCons**[11], and **cetbuildtools**.

Binary package releases are made for a small number of combinations of compiler, architecture,

and optimization level. The packages are delivered in the form of tarball collections of "relocatable **UPS**"[5] packages. The original and venerable Fermilab **UPS** package system centers around a products area and a database directory for version, architecture, and variant information. A user would set up a particular version and variant of a product and dependencies would be set up automatically as required. This enables us to develop **art** naturally as a collection of interdependent packages while allowing experiments to use it with a single command. When discussing the requirements of such a deployment system however, our user community found the product declaration procedure and syntax cumbersome and error-prone. The ability for **UPS** to operate without a central database area was added at our request, with the version information being provided with the tarball for each package thereby rendering the declaration procedure unnecessary.

The **UPS** delivery system allows us to decouple the repository and build system used in the development of **art** from the development systems used by the experiments that reply upon **art**. This decoupling allows development of **art** to proceed at its own pace, which may be different from that of any experiment that uses it, as well as allowing each experiment using **art** to move to new released versions at its own pace, independently of the other experiments. The ability of each experiment to dictate its own schedule for software releases is a critical requirement for all the experiments that use **art**.

External dependencies such as **Boost**, **ROOT**, *etc.*, are built by a member of our group to experimental requirements (for example, three different configurations of **ROOT** are built) in configurations compatible with those of the **art** suite and distributed as one or more "externals" tarballs for expansion into an experiment's **UPS** directory.

Each experiment writes its own modules, services, data product types and auxiliary code and incorporates them into the execution of an **art** program as shared libraries, built using their own build system and loaded by the **art** executable as plugins. Experiments can write their own executable to invoke the **art** framework if they wish to customize command-line options or alter certain default behaviors; we distribute as part of **art** the C++ `main` function used by our current user community.

In addition to implementing various requested features into **art** directly, the development team offer design and implementation advice and code review services to the experiments when requested. Also, major updates to dependencies (*e.g.*, **ROOT**, compilers) are coordinated with experiments, and migration assistance is provided as appropriate.

5. Future directions

We categorize ongoing and future work on **art** as *evolutionary enhancements*, *new features* or *major work*. In the first category we include several small development efforts.

- Use of the in-file **SQLite** database will be expanded to include all existing metadata.
- The concepts of *event*, *subrun* and *run* will be unified, allowing greater flexibility in use of inter-product references.
- The processing intervals will be enhanced to have more intuitive behavior across file boundaries.
- Internal use of **Reflex** will be excised to prepare for the release of **ROOT** 6.0 / **Cling**.
- ISO C++2011 compilation will be enabled across the **art** suite (already the case in development builds) and experiments will migrate as and when they are ready.

Expected new features will include the ability to store experiment-specified metadata in the in-file **SQLite** database and an event-display toolkit providing a well-defined graphical toolkit-agnostic interface to the framework. In addition, we plan to generalize and expand

[5] Also sometimes called "**UPS** Lite".

cetbuildtools and our package delivery system for use by experiments as an alternative to supporting their own build system.

In the category of major work, **art** will be upgraded to be able to process multiple events simultaneously in the same executable and allow for algorithm parallelization within modules. Moving to purely *on demand* module execution, in which the order of module reconstruction module execution is determined by the data dependencies implicit in the code, will further allow multiple modules to process the same event simultaneously. Developing a viable scheme for multi-thread and/or multi-process I/O should is also important in the context of utilizing maximally the multi- and many-core attributes of modern server systems.

Last but not least, we are prototyping multi-process DAQ event-building and triggering using **art** (**artdaq**[12]) for the DarkSide-50, Mu2e, μBooNE and NOνA experiments with very promising results so far[12, 13].

6. Summary

The **art** suite has been in use by several Intensity Frontier experiments since about January of 2011 and at time of writing is used by six experiments as their general purpose offline event processing framework. It is also under consideration by two more experiments. The **art** development team is able to develop and maintain the suite with an average effort expenditure of about two full-time equivalents. We are pressing ahead with features as requested by current clients, and are keeping abreast of developments in language facilities (C++2011, **GCC**, **Clang**) and third-party software (**ROOT**, *etc.*) to improve continuously the performance of the **art** suite. In addition, we are working with several experiments to leverage **art** in online event building and triggering applications and are working to leverage multi- and many-core technologies in multiple ways and at several layers of the application architecture. Finally, we are planning to package and release our build system to allow users to avoid having to maintain their own, and as experiments move towards data taking we will be continuing to improve **art**'s performance and add new features as requested by its users.

Acknowledgments

This work was supported in part by the U.S. Department of Energy, Office of Science, HEP, Scientific Computing.

References

[1] The **art** home page: `https://cdcvs.fnal.gov/redmine/projects/art`

[2] Jones C *et al.* 2007 *Proceedings of the Conference on High Energy Physics, Mumbai 2006* ed Banerjee S (Macmillan India)

[3] The **FHiCL** home page: `https://cdcvs.fnal.gov/redmine/projects/fhicl`

[4] The **ROOT** home page: `http://root.cern.ch/`

[5] The **SQLite** home page: `http://www.sqlite.org`

[6] The **Boost** home page: `http://www.boost.org`

[7] The **UPS** and **UPD** home page: `http://www.fnal.gov/docs/products/ups/`

[8] The **cetbuildtools** home page: `https://cdcvs.fnal.gov/redmine/projects/cetbuildtools`

[9] The **CMake** home page: `http://www.cmake.org`

[10] Amundson J 2001 *Computing in High Energy and Nuclear Physics (CHEP 2000): Proceedings.* (*Computer Physics Communications* vol 140) ed Mazzucato M and Michelotto M pp 731–732

[11] The scons home page: `http://www.scons.org/`

[12] Biery K, Green C, Kowalkowski J, Paterno M and Rechenmacher R 2012 *Conference Record of the 18th IEEE Realtime Conference, Berkeley 2012* (IEEE)

[13] Fischler M, Green C, Kowalkowski J, Norman A, Paterno M and Rechenmacher R 2013 *Conference on Computing in High Energy Physics, New York 2012* Journal of Physics Conference Series