art framework overview

Kyle J. Knoepfel

HSF/DUNE framework requirements mini-workshop

2 June 2021

art software products

- The *art* project distributes several software products.

Full-feature framework and its underlying packages.

Lightweight library that allows reading of *art*/ROOT files; does not create new data products.

Ported from CMSSW's FWLite by a CMS developer.

Lightweight package that provides a development sandbox for testing new user-defined *art* modules.

art concepts

- Hierarchical data processing ($run \supset subrun \supset event$)
- *Experiments decide* how to define the processing levels (e.g. event)
- All processing elements are plugins, loaded at run-time via user configuration
  - Input source
  - Data-processing modules
  - Output modules
  - Tools, user-loadable plugins providing extra flexibility
  - Other utilities that facilitate data-processing
- *art* provides various input sources and output modules, but all processing elements can be user-defined
- Workflows are assembled by a configuration file loaded at run-time
  - Adjustments to workflows do not require recompilation of C++ source code

Highlighted features

- Concurrent processing of events supported within a subrun (inspired by CMS)
  - Scheduled (art) vs. on-demand (CMS)
- Data-product management is thread-, type-, and `const`-safe
  - Early deletion of products supported for memory mitigation
- Core framework functionality does not depend on ROOT
  - We support a separate package (art-root-io) that provides a ROOT I/O layer
- Secondary input (backing) files

Features discussed further

- Configuration description and validation suite
- Graph of data dependencies between modules
- Output file rollover based on user-defined criteria (e.g. max. events processed)
- Implicit data-product aggregation for non-event products (e.g. POT accounting)

Configuration language

- Fermilab Hierarchical Configuration Language (FHiCL)
  - **Not** Turing complete; it's a data-description language
  - "JSON on steroids"
    - Removes extraneous syntax (quotation marks and trailing commas)
    - Supports comments
    - Facilities for ensuring single points of maintenance (splicing, substitution, etc.)
    - Designed to be scripted and not scripting

Configuration language

- Fermilab Hierarchical Configuration Language (FHiCL)
  - **Not** Turing complete; it's a data-description language
  - "JSON on steroids"
    - Removes extraneous syntax (quotation marks and trailing commas)
    - Supports comments
    - Facilities for ensuring single points of maintenance (splicing, substitution, etc.)
    - Designed to be scripted and not scripting

- What would we do differently?
  - FHiCL has served our users well (generally speaking)
  - Has become a maintenance burden for FNAL
  - Will likely support Jsonnet in the future (didn't exist when *art* was born)
    - Gaining momentum across industry as configuration language

Configuration description and validation

- Common problem: how do I configure my program? What if I make a mistake?

Configuration description and validation

- Common problem: how do I configure my program? What if I make a mistake?
- Users can provide C++ documentation/validation structures:

```
class art::Prescaler : public SharedFilter {
public:
  struct Config {
    Atom<size_t> prescaleFactor{
      Name("prescaleFactor"),
      Comment("This module filters one of every n events, where n\n"
              "is the 'prescaleFactor'.")};
    Atom<size_t> prescaleOffset{
      Name("prescaleOffset"),
      Comment("An offset is allowed--i.e. the sequence of events\n"
              "does not have to start with the first event."),
      {}};
  };

  using Parameters = Table<Config>;
  explicit Prescaler(Parameters const&, ProcessingFrame const&);
```

```
Prescaler::Prescaler(Parameters const& config, ProcessingFrame const&)
  : SharedFilter{config}
  , n_{config().prescaleFactor()}
  , offset_{config().prescaleOffset()}
{
  async<InEvent>();
}
```

Configuration description and validation

- Common problem: how do I configure my program? What if I make a mistake?
- Users can provide C++ documentation/validation structures:

```
class art::Prescaler : public SharedFilter {
public:
  struct Config {
    Atom<size_t> prescaleFactor{
      Name("prescaleFactor"),
      Comment("This module filters one of every n events, where n\n"
              "is the 'prescaleFactor'.")};
    Atom<size_t> prescaleOffset{
      Name("prescaleOffset"),
      Comment("An offset is allowed--i.e. the sequence of events\n"
              "does not have to start with the first event."),
      {}};
  };

  using Parameters = Table<Config>;
  explicit Prescaler(Parameters const&, ProcessingFrame const&);
```

```
Prescaler::Prescaler(Parameters const& config, ProcessingFrame const&)
  : SharedFilter{config}
  , n_{config().prescaleFactor()}
  , offset_{config().prescaleOffset()}
{
  async<InEvent>();
}
```

```
module_type : Prescaler (or "art/Framework/Modules/Prescaler")

    provider: art
    type    : filter
    source  : /home/knoepfel/art/art/Framework/Modules/Prescaler
    library : /home/knoepfel/scratch/builds/debug/art/lib/libart

Allowed configuration
---------------------

    ## Any parameters prefaced with '#' are optional.

    <module_label>: {

       module_type: Prescaler

       errorOnFailureToPut: true  # default

       ## This module filters one of every n events, where n
       ## is the 'prescaleFactor'.

       prescaleFactor: <unsigned long>

       ## An offset is allowed--i.e. the sequence of events
       ## does not have to start with the first event.

       prescaleOffset: 0  # default
    }
```

Configuration description and validation

- Common problem: how do I configure my program? What if I make a mistake?
- Users can provide C++ documentation/validation structures:

```
class art::Prescaler : public SharedFilter {
public:
  struct Config {
    Atom<size_t> prescaleFactor{
      Name("prescaleFactor"),
      Comment("This module filters o
               "is the 'prescaleFacto
    Atom<size_t> prescaleOffset{
      Name("prescaleOffset"),
      Comment("An offset is allowed-
               "does not have to star
      {}};
  };

  using Parameters = Table<Config>;
  explicit Prescaler(Parameters cons
```

```
Prescaler::Prescaler(Parameters cons
  : SharedFilter{config}
  , n_{config().prescaleFactor()}
  , offset_{config().prescaleOffset(
{
  async<InEvent>();
}
```

```
module_type : Prescaler (or "art/Framework/Modules/Prescaler")

    provider: art
    type    : filter
    source  : /home/knoepfel/art/art/Framework/Modules/Prescaler
              /knoepfel/scratch/builds/debug/art/lib/libart

                                                          on
                                                          ---

                                                          rs prefaced with '#' are optional.

                                                           {

                                                           Prescaler

                                                          reToPut: true  # default

                                                          le filters one of every n events, where n
                                                          rescaleFactor'.

                                                          or: <unsigned long>

                                                          is allowed--i.e. the sequence of events
                                                          have to start with the first event.

       prescaleOffset: 0  # default
    }
```

```
================================================================

!! The following modules have been misconfigured: !!

----------------------------------------------------------------


Module label: prescaleEvents
module_type : Prescaler

Any parameters prefaced with '#' are optional.
Unsupported parameters:

 + prescaleoffset                  [ ./test.fcl:6 ]


================================================================
```


Fermilab

Configuration processing and persistence

- The fully processed configuration is persisted to output files.
  - Self-describing output files
  - Can reproduce an output file solely by its persisted configuration (assuming same environment is setup)

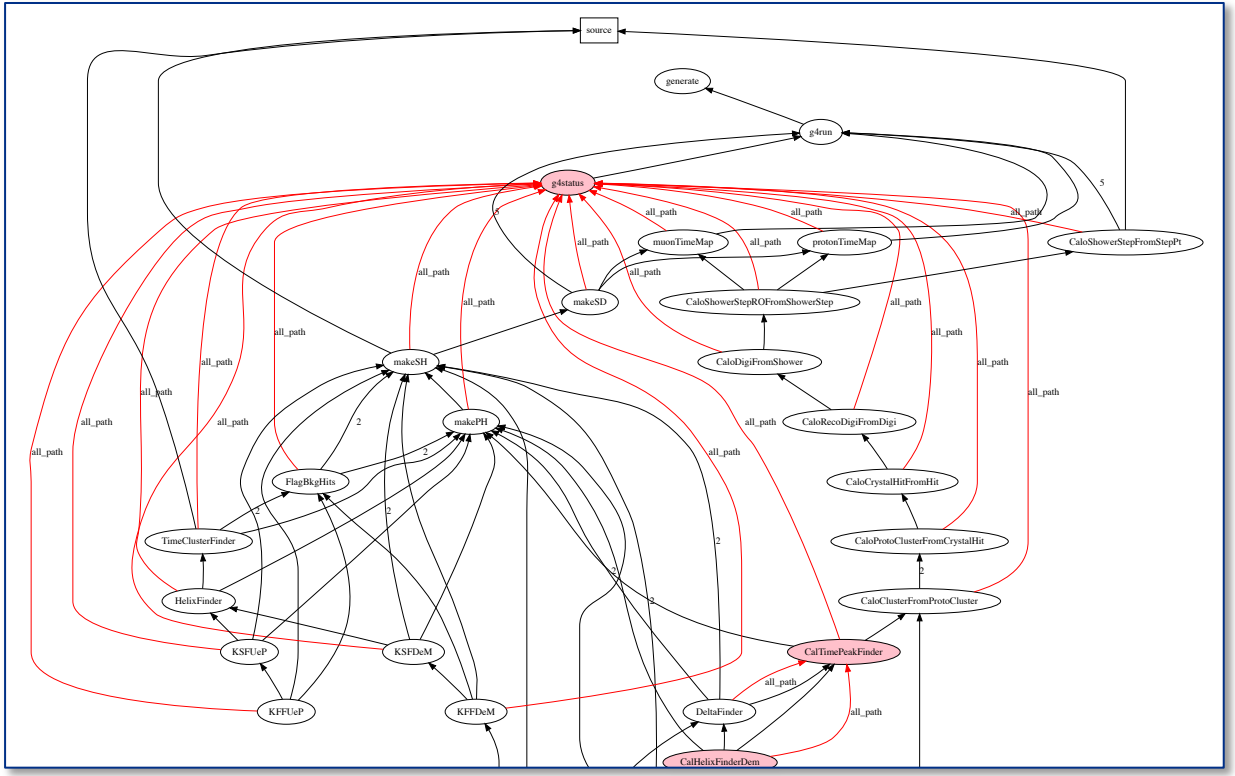
Configuration processing and persistence

- The fully processed configuration is persisted to output files.
  - Self-describing output files
  - Can reproduce an output file solely by its persisted configuration (assuming same environment is setup)

- During the configuration-processing stage, *art* prunes modules from the configuration that are not used in event-processing.
  - Reduces memory footprint of job
  - Reduces size of persisted configuration
  - Simplifies debugging
  - Allows for better equivalence relationships among configurations

Data-dependency graph

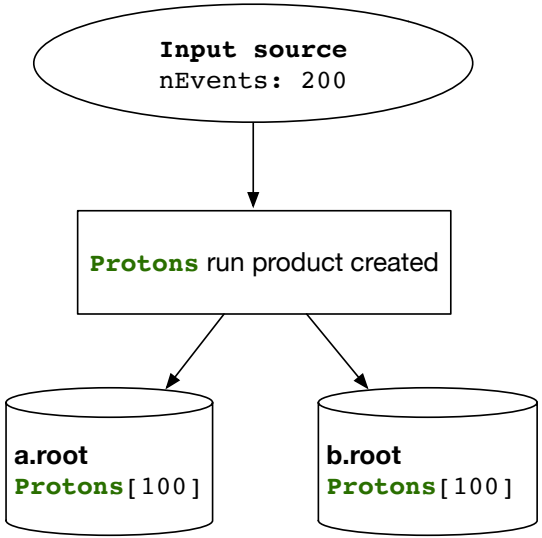
- *art* assembles the graph of data dependencies between modules.
- Data-dependency errors are caught at job start-up time, just after module constructors have been called.
- We have not yet used the graph to optimize event processing.



Fermilab

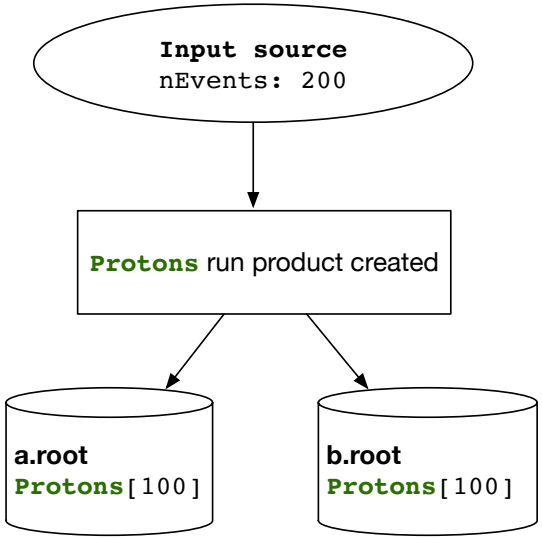
Implicit data-product aggregation

- Users can configure output modules to rollover to a new output file when a condition is met (max. number of events, file size, time open, etc.).
- (Sub)run products can be spread across multiple files
- Whenever the files are concatenated together, *art* can combine the products according to an aggregation behavior (e.g.):
  - Count of protons-on-target are summed
  - Map of particle species are combined via insert
  - Any user-defined aggregation function



Implicit data-product aggregation

- Users can configure output modules to rollover to a new output file when a condition is met (max. number of events, file size, time open, etc.).
- (Sub)run products can be spread across multiple files
- Whenever the files are concatenated together, *art* can combine the products according to an aggregation behavior (e.g.):
  - Count of protons-on-target are summed
  - Map of particle species are combined via insert
  - Any user-defined aggregation function



- *art* infrastructure necessary for this makes some multi-threading issues easier:
  - Distinction between a (e.g.) full run vs. a run fragment
  - Set of events corresponding to a given product (to avoid double counting)

Conditions information

- No system specifically dedicated to conditions information.

- *art* supports several facilities that are often used in providing such info:
  - Worked with DB experts at FNAL to design a C++ API that meshes DB access with framework use
  - Services frequently used to retrieve DB information
  - Caching system with concurrent entry insertion, deletion, and retrieval
    - Users define intervals-of-validity

art support model

- Decisions are jointly made by the development team and stakeholders
  - Representatives that convey the needs of the experiment/project.
  - All stakeholders' votes have equal weight

art support model

- Decisions are jointly made by the development team and stakeholders
  - Representatives that convey the needs of the experiment/project.
  - All stakeholders' votes have equal weight

- Potential for experiments to want conflicting framework behaviors/features
  - True in principle; but it has not happened in 10 years
  - We strive hard for stakeholder consensus on any given feature.
  - We provide enough flexibility in the framework that each experiment's needs can be met.

art support model

- Decisions are jointly made by the development team and stakeholders
  - Representatives that convey the needs of the experiment/project.
  - All stakeholders' votes have equal weight

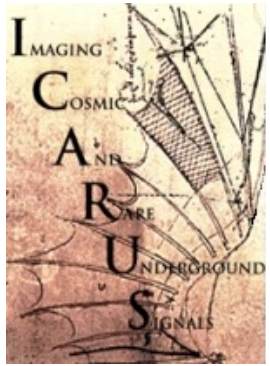
- Potential for experiments to want conflicting framework behaviors/features
  - True in principle; but it has not happened in 10 years
  - We strive hard for stakeholder consensus on any given feature.
  - We provide enough flexibility in the framework that each experiment's needs can be met.

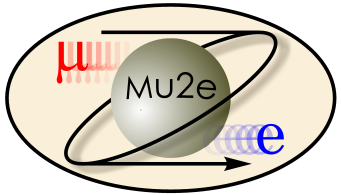
- All stakeholders use the same *binary* executable
  - Some minute variations wrt default configurations and executable names

art provides the framework needs for ~2k physicists

artdaq project

LArIAT experiment





art provides the framework needs for ~2k physicists

ArgoNeuT

60,498

artdaq project

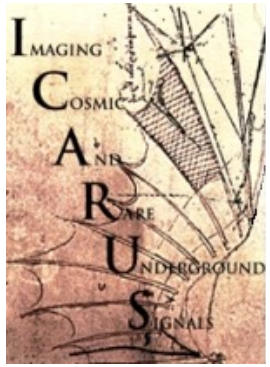
40,647

DUNE DEEP UNDERGROUND NEUTRINO EXPERIMENT

296,392

LArIAT experiment

56,051



65,641

LArSoft

411,891

SHORT-BASELINE NEAR DETECTOR SBND

49,889

Mu2e

276,590

g-2

1,126,526

NOvA

1,929,129

µBooNE

575,789

4.9M LOC in aggregate

Some learned lessons: **Know your users**

Not only do you need to know what the framework is intended to accomplish, but *who* are the individuals that will be using it.

In our experience, framework users…

Some learned lessons: **Know your users**

Not only do you need to know what the framework is intended to accomplish, but *who* are the individuals that will be using it.

In our experience, framework users…

- Are often willing to learn
- Are anxious to try new language features as soon as possible
- Prefer lightweight, simple-to-use systems

Some learned lessons: **Know your users**

Not only do you need to know what the framework is intended to accomplish, but *who* are the individuals that will be using it.

In our experience, framework users…

- Are often willing to learn
- Are anxious to try new language features as soon as possible
- Prefer lightweight, simple-to-use systems
- Rarely read documentation (which is sometimes a blessing)
- Often work under supervisors who have little regard for robust coding practices
- Often suggest solution X when they need to solve problem Y

Some learned lessons: **Sociology is the hardest part**

- How do you encourage users to try something new?

Some learned lessons: **Sociology is the hardest part**

- How do you encourage users to try something new?

In our experience, an HEP experiment will willingly consider using a new software product if each of the following are true:

a. **There is a perceived technological benefit**
b. **There is a clear support model that benefits the experiment**
c. **There is a practical migration path**

Some learned lessons: **Sociology is the hardest part**

- How do you encourage users to try something new?

  In our experience, an HEP experiment will willingly consider using a new software product if each of the following are true:

  a. **There is a perceived technological benefit**
  b. **There is a clear support model that benefits the experiment**
  c. **There is a practical migration path**

- It is not always easy to achieve all three.

Some learned lessons: **framework-specific**

As presented (with minor adjustments) 10 March 2020 at Jefferson Lab's Software & Computing Roundtable
- https://indico.jlab.org/event/356/#4-the-art-framework-what-it-is

- *art*'s rigid processing hierarchy has been an awkward fit for neutrino experiments
  - Would probably pursue something more flexible next time
- *art* supports a class a plugins (services) that can be accessed from anywhere.
  - This has led to many thread-safety issues that experiments must deal with.
- *art* users can access metadata and provenance about data products
  - Many users do not look at this information
  - Not clear how much of this has been worth it; might think of something else next time.
- Framework limitations are not bad! Know what they are.

Some learned lessons: **framework-specific**

As presented (with minor adjustments) 10 March 2020 at Jefferson Lab's Software & Computing Roundtable
- https://indico.jlab.org/event/356/#4-the-art-framework-what-it-is

- *art*'s rigid processing hierarchy has been an awkward fit for neutrino experiments
  - Would probably pursue something more flexible next time
- *art* supports a class a plugins (services) that can be accessed from anywhere.
  - This has led to many thread-safety issues that experiments must deal with.
- *art* users can access metadata and provenance about data products
  - Many users do not look at this information
  - Not clear how much of this has been worth it; might think of something else next time.
- Framework limitations are not bad! Know what they are.

Thanks