Meld

Exploring the Feasibility of a Framework-less Framework

Kyle Knoepfel[1,*]

[1]Fermi National Accelerator Laboratory, P.O. Box 500, Batavia, Illinois, United States

Abstract. HEP data-processing frameworks are essential ingredients in getting from raw data to physics results. But they are often tricky to use well, and they present a significant learning barrier for the beginning HEP physicist. In addition, existing frameworks typically support rigid, collider-based data models, which do not map well to neutrino-physics experiments like DUNE. Neutrino physicists thus expend significant effort working around framework limitations instead of using a framework that directly supports their needs.

Presented here is Meld, a Fermilab R&D project, which intends to address these limitations. By leveraging modern C++ capabilities, state-of-the-art concurrency libraries, and a flexible data model, it is possible for beginning (and seasoned) HEP physicists to execute framework programs easily and efficiently, with minimal coupling to framework-specific constructs. Meld aims to directly support the frameworks needs of neutrino experiments like DUNE as well as the more common collider-based experiments.

1 Introduction

The Deep Underground Neutrino Experiment (DUNE) is Fermilab's flagship scientific effort toward further understanding neutrino oscillations, measuring potential charge-parity violation in the neutrino sector, and detecting neutrinos from supernova bursts and beyond-the-Standard Model physics. The experiment's near detector is located at Fermilab, and the far detector is composed of one or more liquid Argon (LAr) time-projection chambers (TPCs) at the Sanford Underground Research Facility near Lead, South Dakota. The LAr active volume in the TPCs affords very high resolution in the detection of neutrino interactions.

Like other experiments, DUNE will use an offline computing framework to process the data collected by the online data acquisition system. However, whereas LHC experiments rely on hadronic collisions occurring within localized accelerator bunch crossings, the neutrino interactions recorded by DUNE will be distributed throughout very large detector volumes. Unfortunately, most reconstruction and simulation frameworks in HEP (e.g. see [1–4]) assume collider-physics concepts that are not always appropriate for neutrino experiments. Even DUNE's current offline framework *art* [4] reflects a collider-centric design as it originated as a fork of the CMS framework. A framework is therefore needed that can support DUNE without physicists having to work around the collider assumptions imposed by existing frameworks.

---

*e-mail: knoepfel@fnal.gov

The DUNE experiment has formulated a list of offline framework requirements [5] based on the physics goals mentioned above. Some of those requirements state that:

- physics algorithms should be framework-agnostic,

- the framework must be able to break apart events into smaller chunks for more granular processing, and then stitch those chunks back together into an event,

- the framework should support "sliding event windows" to provide "edge effect" coverage for extended time readouts during supernovae events, and

- the framework should make minimal assumptions about the data model.

To determine the feasibility of developing a framework that could meet such requirements, a laboratory-directed R&D project, now called Meld, was established in 2022 at Fermilab.

This article explains Meld's design by first discussing in Section 2 the logical foundations of data products, data hierarchies, and how data products are processed within those hierarchies. Section 3 presents the concept of higher-order functions and their roles within a dependency graph of user-defined operations. The consequences of a functional approach on interfaces are addressed in Section 4, which is followed in Section 5 by a description of the features supported by Meld's prototype implementation.

2 Logical representations of data

Reconstruction and simulation frameworks in HEP refer to their managed data objects as *data products*, each of which is:

1. Identifiable, to facilitate user-specification of what data to analyze,

2. A member of at least one data set (or *domain*) such as an *event*,

3. Immutable, as implied by the definition of an element of a mathematical set, and

4. Opaque to the framework, implying a separation of the user space from the framework.

Data products can be created from other data products through user-defined functions or *mappings*, the details of which are defined based on the use case. In the case of *art*, users do not create explicit mappings between data products but instead apply implicit mappings while interacting directly with the domain containing the data product. One of the goals of Meld is to replace implicit mappings between data products with explicit mappings that do not directly depend on the data-product domain.

2.1 Data hierarchies

As mentioned above, data products do not exist by themselves, but they are members of a domain. Such domains are often called events, but as events are often contained by supersets (such as *luminosity sections*, *subruns*, or *runs*), then a given event data product logically is also a member of any supersets of that event. The organization of data sets and their relationships with each other define a *data hierarchy*.

Commonly used data hierarchies include tree-like structures that honor relations such as $Run_i \supset Subrun_j \supset Event_k$, which allows an unambiguous identification of an event through the number triplet (i, j, k). Although frameworks may allow an experiment to specify the range of values taken by the identifiers i, j, and k, the hierarchical relation is rigid, thus constraining programs to data organizations that may not match well the processing needs of the user. The DUNE experiment has found this to be the case, where the lack of granularity
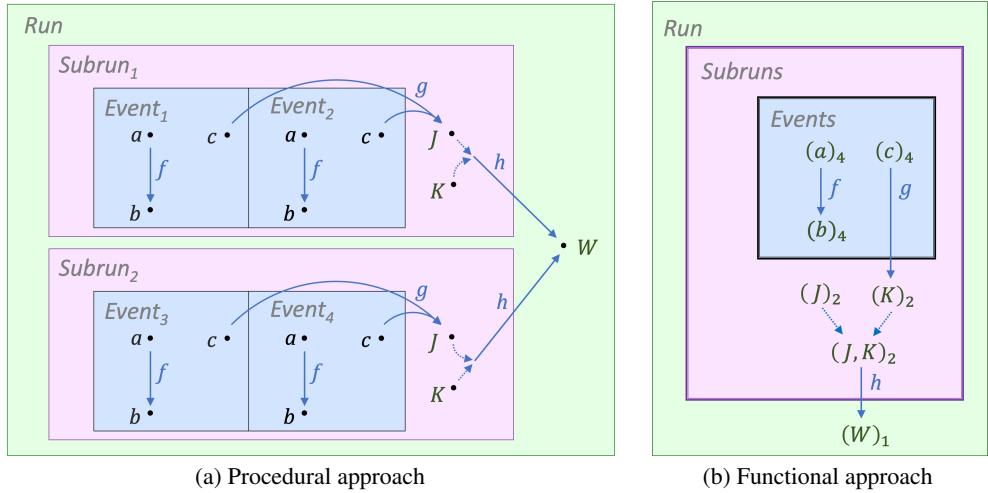
(a) Procedural approach   (b) Functional approach

Figure 1. Logical representations of processing data with (a) a procedural approach, and (b) a functional approach.

below the event domain yields processing complications and program-memory footprints that are difficult to manage. To satisfy DUNE's needs, a system is required that can support dynamic and more general hierarchies than those discussed above (some examples are shown in Section 5).

2.2 Processing data in two ways

Figure 1 depicts two ways of processing a tree-like data set that contains one run, two subruns within the run, and two events within each subrun (four events total). The run contains a data product (i.e. an object) labeled W, each subrun contains two data products J and K, and each event contains the data products a, b, and c. The mappings between the data products include:

- $f : a \mapsto b$ for each event,

- $g : c \mapsto J$ for each event such that only one data product J is created for each subrun, and

- $h : (J, K) \mapsto W$ for each subrun such that only one run data product W is created.

Whereas f is a simple mapping or *transform* of one data product to another within an event, g and h correspond to *reductions* (or *folds*) where multiple data products across multiple domains are aggregated into one data product of an encapsulating domain. Examples of reductions within the HEP community include histograms and event counts, which are commonplace ingredients used to obtain physics results.

Panel (a) of Figure 1 shows a procedural way of processing data, where the domain of each data product is explicit as is each invocation of the user-defined mappings. This approach is common among HEP frameworks, which often require users to interact with the domain containing the data instead of transparently providing the requested data. A consequence of this programming paradigm is that users tailor their code to work with stateful objects that represent the domains. The significant boilerplate code required for such a pattern often leads to poor separation of physics algorithms (which generally can be framework-agnostic) from the code required to register those algorithms with the framework.

Meld, however, adopts a functional processing paradigm, as illustrated in panel (b) of Figure 1. With a functional approach, the framework forms sequences of data products across common domains (e.g. all events) and applies the user-defined mapping to each data product of the sequence. For example, the mapping f is applied to each data product a contained in events 1 through 4. Using a notation similar to Ref. [6], this operation can be expressed using a *higher-order function*:

$$f * (a)_4 = (b)_4 \tag{1}$$

where the $*$ infix operator accepts a function f and the sequence $(a)_4$, which signifies a sequence of 4 data products, each labeled a. The framework still records the domain of each data product, but the domain is retained by the framework as a bookkeeping artifact that is accessible by the user only when necessary.

Note that the formation of data-product sequences in the functional approach need not be "strict" in the sense that all data products of a sequence must exist before the framework can apply the user-defined mapping. In some cases (e.g. a data acquisition environment), the full sequence cannot be known at any given time, and even when the full sequence can be known, memory limitations of the machine may constrain the framework to process only a subset of the sequence at a time.

3 Higher-order functions and graphs

As mentioned above, Equation 1 gives an example of a higher-order function, which is any function that (a) receives at least one function as one of its function parameters, or (b) returns a function. Case (a) applies in framework contexts, where the user provides a callable entity or *operator* to be applied to sequence elements, specifies the pattern used to form that data-product sequence, and chooses the kind of higher-order function desired (e.g. transform).

Table 1 lists the higher-order functions for some computing patterns commonly used within HEP and the broader computing industry. Although it is typical among HEP applications to transform, filter, monitor, and reduce data products or their domains, it is less common to split them. However, as mentioned in Section 1, the splitting of data-product domains is one of DUNE's framework requirements. Meld was therefore designed to support such splitting.

Framework jobs generally consist of many user-defined functions that must be applied to data in a particular composition order. This order results in a dependency graph that can be exploited to optimize the processing of the data products. Both panels in Figure 1 can be considered dependency graphs. However, whereas panel (a) depicts a graph of many identically named nodes and edges, panel (b) shows a significantly smaller graph where nodes and edges have unique names. A functional approach thus enables a simpler graphical description of the processing required than the procedural approach.

3.1 Side effects and concurrency support

The ability to support concurrent processing of data depends strongly on how many side effects are present in the program (e.g. accessing global state, or manipulating objects shared among threads). Table 1 assumes that the user-defined operators contain no side effects, thus making the invocation of such operators automatically safe within a multi-threaded or multi-process context.

The functional approach does permit some stateful objects, such as the result object of a reduction (e.g. a histogram). However, it is possible in some cases for the framework to manage and update the result object in a safe manner without placing that burden on the framework user.

Table 1. Higher-order functions and their operator signatures. For each pattern, $a \in \alpha$ and $b \in \beta$, where α and β are mathematical sets representing data products. The mathematical set $\mathbf{1}$ (or `void` in the C and C++ languages) contains one element called *void*. The return type of $\mathbf{1} + \alpha \times \beta$ denotes a *tagged union* whereby the return value can either be *void* or of the set $\alpha \times \beta$.

Computing pattern	Higher-order function	Operator signature
Transform (map)	$f * (a)_n = (b)_n$	$f : \alpha \to \beta$
Filter	$f \triangleleft (a)_n = (a)_m$ where $m \leq n$	$f : \alpha \to$ Boolean
Monitor (absorb)	$f \blacktriangleleft (a)_n = ( )$	$f : \alpha \to \mathbf{1}$
Reduce (fold)	$f\,{}^n\!/ (a)_n = b$	$f : \beta \times \alpha \to \beta$
Split (unfold)	$f\,{}_n\!\backslash\, b = (a)_n$	$f : \beta \to \mathbf{1} + \alpha \times \beta$

4 Declarative interfaces

By treating framework jobs as mappings of data products through higher-order functions, a declarative user interface is possible. Figure 2 shows such an interface where a user-defined, framework-agnostic physics algorithm called `make_tracks` is registered with Meld through the `DEFINE_MODULE` C++ preprocessor invocation. The registration occurs at runtime (upon loading the corresponding shared object library), instructing the framework to apply the user-defined `make_tracks` function to the `"GoodHits"` data product in each event and to then store the result as the `"GoodTracks"` data product.

The framework uses type deduction to match the input and output data product names with their corresponding C++ types—namely, the `"GoodHits"` data product is presented as type `Hits` to the user, and the return value of type `Tracks` is stored as the `"GoodTracks"` data product. The user may specify the allowed concurrency level when invoking the function—an unsigned integer that represents the number of data products that can be processed concurrently by the framework's execution of the registered function. The special keyword `unlimited` instructs the framework that it may use the maximum available concurrency provided by the system on which the `make_tracks` function is invoked.

Understandably, the user may not wish to hard-code the data-product labels (`"GoodHits"` and `"GoodTracks"`), the domain name (`"Event"`), and the allowed concurrency number. In such cases, the `config` object may be queried for the value of a configuration parameter specified by the user at runtime.

4.1 More complicated models

An objection can be made that the example shown in Figure 2 is too simplistic to be useful in a realistic HEP computing context. An important capability of Meld is, therefore, the ability to register with the framework functions that receive arguments corresponding to data products from different domains. Figure 3 shows such an example, where the `make_tracks` function receives an additional function parameter with calibration information provided through the `"CalibrationEntry"` run data product.

In some cases, the `make_tracks` function may require only an object (e.g. an offset) that is created from a `"CalibrationEntry"` data product and not the full calibration information. If creating that offset is time-consuming, it is undesirable to repeatedly create the object for each invocation of `make_tracks`. The user can therefore register a separate `make_offset` function with the framework, which creates the offset once for each run. That offset is then available as a data product that can be used by the `make_tracks` function. Based on the data products required and produced by each algorithm, the framework ensures that algorithms are processed in the correct order. Figure 4 shows this use case, using the `to_temporary` clause to denote a data product that should not be persisted to disk.

```
#include "meld/module.hpp"

namespace expt {
  Tracks make_tracks(Hits const& hits) { ... }
}

DEFINE_MODULE(m, config) {
  m.with(expt::make_tracks, concurrency::unlimited)
    .transform("GoodHits").in_each("Event").to("GoodTracks");
}
```

Figure 2. Example of registering a user-defined function `make_tracks` with Meld, such that all event data products labeled `"GoodHits"` are transformed to data products labeled `"GoodTracks"`. More complicated scenarios are described in Section 4.1.

```
#include "meld/module.hpp"

namespace expt {
  Tracks make_tracks(Hits const& hits, Entry const& entry) { ... }
}

DEFINE_MODULE(m, config) {
  m.with(expt::make_tracks, concurrency::unlimited)
    .transform("GoodHits"_in_each("Event"), "CalibrationEntry"_in_each("Run"))
    .to("GoodTracks");
}
```

Figure 3. Similar scenario as Figure 2, but using the `"CalibrationEntry"` run data product as an additional function argument to `make_tracks`.

```
#include "meld/module.hpp"

namespace expt {
  Offset make_offset(Entry const& entry) { ... }
  Tracks make_tracks(Hits const& hits, Offset const& offset) { ... }
}

DEFINE_MODULE(m, config) {
  m.with(expt::make_offset, concurrency::unlimited)
    .transform("CalibrationEntry").in_each("Run")
    .to_temporary("CalibrationOffset");
  m.with(expt::make_tracks, concurrency::unlimited)
    .transform("GoodHits"_in_each("Event"), "CalibrationOffset"_in_each("Run"))
    .to("GoodTracks");
}
```

Figure 4. Similar scenario as Figure 3, but a separate function `make_offset` is used to perform the calculation of `"CalibrationOffset"` once per run instead of once per event.
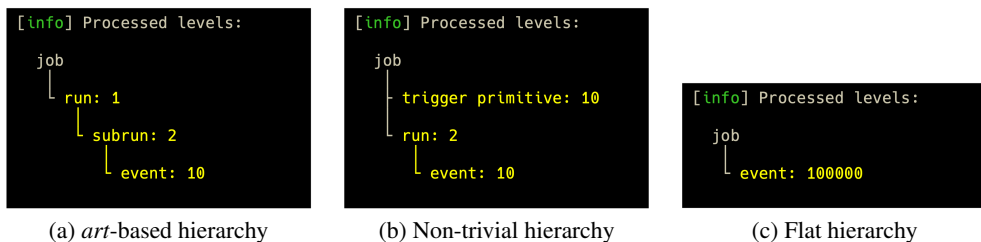
```
[info] Processed levels:

  job
   └─ run: 1
        └─ subrun: 2
             └─ event: 10
```

(a) *art*-based hierarchy

```
[info] Processed levels:

  job
   ├─ trigger primitive: 10
   └─ run: 2
        └─ event: 10
```

(b) Non-trivial hierarchy

```
[info] Processed levels:

  job
   └─ event: 100000
```

(c) Flat hierarchy

Figure 5. Sample hierarchies supported by Meld. The numbers for nested levels correspond to aggregations across all supersets (e.g. for the *art*-based hierarchy, a total of 10 events across 2 subruns were processed).

5 Meld implementation and execution backend

Meld itself does not prescribe a particular implementation, but a C++ prototype is available [7] that uses Intel's oneTBB flow graph library [8] as the multi-threading backend. Each of the computing patterns listed in Table 1 is supported by the prototype. In addition, a zip utility is provided so that multiple data products can be presented to a user-defined function that takes more than one argument (e.g. the *h* reduction in Section 2.2).

Figure 5 shows three sample hierarchies that Meld supports. The hierarchy summaries are printed to the terminal at the end of each Meld program, which records the domains processed and their relations to each other. Hierarchy (a) follows the tree-like structure (see Section 2.1) currently used by the *art* framework. Hierarchy (b) is non-trivial in that runs and trigger primitives are orthogonal domains, but events are subsets of runs. The flat hierarchy (c) processes only events, which are contained only by the "job" domain.

Features yet to be supported by Meld include:

- a "sliding window" over domains, where multiple data products from different domains can be presented to the user at the same time,

- a conditions system where calibration constants do not necessarily belong to only one data hierarchy,

- asynchronous processing of algorithms executed on non-CPU resources (e.g. GPUs), and

- optimizations in scheduling the processing of time-ordered data *vs.* that of a complete data set, whose elements are known ahead of time.

6 Conclusion

Meld's functional approach of processing data products enables a decomposition of a framework job into a granular set of basic computing idioms. The presented declarative interface avoids significant entanglements between user code and framework boilerplate, resulting in a much simpler (even if not framework-less) framework model for experiments. This avenue of exploration has been received favorably by members of the DUNE collaboration, and the successes achieved thus far suggest that higher-order functions have more of a place in HEP data-processing frameworks than has been previously recognized.

7 Acknowledgments

The author thanks the DUNE experiment for many fruitful discussions about their framework needs. He is also grateful to Marc Paterno for his helpful input and careful reading of this

References

[1] Charles Leggett, *et al*, J. Phys. Conf. Ser. **898**, 042009 (2017)

[2] E. Sexton-Kennedy, *et al*, J. Phys. Conf. Ser. **608**, 012034 (2015)

[3] J. Adam, *et al* [ALICE Collaboration], "Technical Design Report for the Upgrade of the Online-Offline Computing System", CERN-LHCC-2015-006, ALICE-TDR-019 (2015)

[4] C. Green, *et al*, J. Phys. Conf. Ser. **396**, 022020 (2012)

[5] A. Abed Abud, *et al* [DUNE collaboration], DUNE Offline Computing Conceptual Design Report, arXiv:2210.15665 [physics.data-an] (2022)

[6] R. Bird, Logic of Programming and Calculi of Discrete Design, NATO ASI Series **36** https://doi.org/10.1007/978-3-642-87374-4_1 (1987)

[7] Meld, GitHub repository, https://github.com/knoepfel/meld (2023)

[8] oneAPI Threading Building Blocks (oneTBB), https://oneapi-src.github.io/oneTBB