Phlex

Release 0.3 (For Review)

Jul 30, 2025

CONTENTS

Preface

Phlex is a data-processing framework that supports the **P**arallel, **h**ierarchical, and **l**ayered **ex**ecution of data-processing algorithms.

This document gives a brief introduction to what Phlex aims to achieve, the conceptual design elements in support of those aims.

CHAPTER

ONE

INTRODUCTION

A resonable definition of a software framework is [Wiki-Framework]:

> *an abstraction in which software, providing generic functionality, can be selectively changed by additional user-written code, thus providing application-specific software.*

The framework orchestrates data flow, resource management, and parallel execution. It enables a scientific collaboration to write standardized workflows where physicists can insert their own algorithms. In a HEP context, this insertion often occurs by the framework dynamically loading libraries called *plugins*. Although not required, a framework often provides a program's *main(…)* function, which (directly or indirectly) invokes user code within the plugins as configured at appropriate points in the program's execution.

The following diagram illustrates three high-level computing stages commonly used in HEP to obtain physics results from detector signals.

Frameworks are typically used in a high-level trigger environment, for reconstructing physics objects from detector signals, or for simulating physics processes. Many analysis needs can also be met by a data-processing framework. However, the HEP community tends to perform final-stage analysis using standalone applications. Phlex, therefore, aims to satisfy the data-processing needs of only physics reconstruction and simulation.

1.1 Requirements Process and Framework Selection

Phlex provides facilities and behaviors that support the physics goals of its stakeholders, notably the DUNE experiment[1]. In a concerted effort in 2023 and 2024, DUNE established a set of high-level requirements or *stakeholder requirements*, which constrain the design of a framework in support of DUNE's needs. A dedicated tool [Jama-Connect] was subsequently selected to manage such stakeholder requirements, tracking them in a version-controlled manner, and creating logical dependencies among them. Additionally, *system requirements* were created to guide implementation in support of the stakeholder requirements.

After formulating its stakeholder requirements, DUNE evaluated whether existing HEP frameworks could also satisfy DUNE's needs. The frameworks considered included the *Gaudi* framework [Gaudi] (used by ATLAS and LHCb), *CMSSW*'s framework [CMSSW], ALICE's *O2* framework [O2], as well as the *art* framework [art], which is used by many of the intensity-frontier experiments at Fermi National Accelerator Laboratory. Over time, each framework has undergone substantial adjustments to take advantage of hardware and software developments (multi-threading, multi-processing, GPU usage, etc.), resulting in more efficient data-processing, and (in some cases) making possible some data-processing that may have been infeasible without adjustment.

[1] It is possible for additional experiments to become stakeholders of the Phlex framework. In such a case, the stakeholder requirements of one experiment may not negate those of another (particularly DUNE).

Each framework considered above, however, is designed according to event-centric, collider-physics concepts. The DUNE stakeholder requirements demand that any event-centric assumptions must be relaxed to permit more flexible data organizations (see Section 1.2.1). Specifically, the data groupings of interest must not be rigidly defined by the framework itself, but specifiable by the user [DUNE 22]. Even *art*, the framework used by the ProtoDUNE experiments, forces users to adopt awkward workarounds to process the relatively slow-evolving detector signatures of neutrino interactions (as compared to the very fast beam interactions of collider experiments).

To be sure, an existing HEP framework can be modified to eventually meet the needs of DUNE. However, such modification does not exist in a vacuum, and in DUNE's determination, adjusting an existing HEP framework to satisfy DUNE's stakeholder requirements was impractical. Significant changes would be required for any of the frameworks, and it is unclear the manner and extent to which such changes would be accepted by already operational experiments using the framework in question. DUNE thus decided to develop a new framework designed to directly support DUNE's framework requirements. Establishing a new framework also provides an opportunity to shed legacy coding patterns that have been problematic in achieving efficient data-processing.

1.1.1 Requirements Ownership

Each Phlex stakeholder owns its stakeholder requirements, which support the high-level needs of the experiment. System requirements, which are subservient to stakeholder requirements, are owned by the Phlex developers, who are free to adjust the implementation to satisfy all stakeholder requirements.

1.1.2 Requirements in This Document

The stakeholder requirements are listed in Appendix B for convenience. To more easily connect the design to the requirements, any design aspect influenced by specific requirements contains bracketed references to those requirements (e.g. [DUNE 22]).

Where possible, we limit references to stakeholder requirements to the conceptual design in Chapter 3. Some stakeholder requirements are referenced in the technical design *(under preparation)* if those requirements do not affect the conceptual framework model. No system requirements are currently referenced in this document.

1.2 Framework Philosophy

A framework is a tool that aids the scientific process of inferring accurate physics results from observed data. Maintaining data integrity is therefore paramount, as is retaining an accounting of how physics results were obtained from that data. The Phlex design therefore:

- treats all data presented to (or created by) Phlex as immutable for the remainder of a Phlex program's execution,
- requires recording the *provenance* of every created *data product* [DUNE 121], and
- enables, and—to the extent possible—ensures the *reproducible* creation of data products.

1.2.1 Flexibility

Physics results in HEP are obtained by processing large collections of data and making statistical statements from them. Each element of a collection generally contains the data corresponding to one readout of the detector. Such elements are often termed "events", which are treated as statistically independent observations of physics processes. It is common for experiments to define larger aggregations of data by grouping events into subruns (or, for LHC experiments, luminosity blocks), and by further grouping subruns into runs. These larger aggregations are typically defined according to when certain detector calibrations or accelerator beam parameters were applied.

Although frameworks supporting the *Run-Subrun-Event* (RSE) hierarchy have proved effective and flexible enough for collider-based experiments, the RSE hierarchy is not always appropriate:

- simulated data often do not need to be processed with an RSE hierarchy; a flat hierarchy (e.g. only the "event") is usually sufficient,

- framework interface is often explicitly couched in RSE terminology, making it difficult to apply to non-collider contexts, where a different data-grouping may be more appropriate (e.g. time slices for extended readout windows, each of which correspond to one "event"),
- calibration data is often described independently from an RSE hierarchy, requiring other means of accounting for systematic corrections that must be applied to the data.

Phlex does not prescribe an RSE hierarchy—it only requires that the hierarchy be representable as a directed acyclic graph (DAG) at run-time, with each grouping of data represented as a node in the graph, and the relationships between data-groupings represented as edges. This expression of the hierarchy greatly relaxes the constraints placed on experiments while still supporting the collider-based RSE hierarchy (see Section 3.2.2).

The hierarchy graph and its nodes (i.e. the data-groupings) are definable at run-time, thus allowing the specification of data organizations that are appropriate for the workflow [DUNE 22].

The flexibility in defining data-groupings and how they relate to each other necessitates further flexibility:

1. user-defined algorithms are not bound to statically-typed classes representing data-groupings—e.g. there is no direct dependency on a C++ "event" class, and
2. a framework program must be "driven" by a user-provided entity that expresses the hierarchy graph desired by the user, not a hierarchy that is prescribed by the framework.

These concepts are discussed more fully in Chapter 3.

1.2.2 Portability

Phlex is intended to be used on a variety of computing systems to take advantage of the disparate computing resources available to each stakeholder [DUNE 8]. This means the framework:

- must support data-processing by algorithms that execute on GPUs [DUNE 11], in addition to those that execute on CPUs,
- may not generally rely on hardware characteristics unique to a particular platform [DUNE 63],
- must favor standardized programming-language features.

1.2.3 Usability

Although usability is not a formal stakeholder requirement, physicists expect various behaviors and features that ease one's interaction with a data-processing framework. Phlex strives to meet this expectation in various ways:

minimizing boilerplate code
: Some data-processing frameworks in HEP adopt an object-oriented design, where stateful framework-dependent objects are required to register inherently framework-agnostic algorithms with a framework program. Phlex does not generally require physics algorithms to depend on any framework libraries [DUNE 43]. This design, therefore, substantially reduces the amount of code required for the interface between physics algorithms and the framework itself (see Section 1.4).

failing early
: To avoid needless computation, Phlex will fail as early as possible in the presence of an error. This means that, for C++ usage, compile-time failures will be favored over run-time exceptions.

meaningful error messages
: When failures within the scope of the framework occur[2], the reported error messages will be as descriptive as possible. Messages will typically include diagnostic information about the data being processed when the error occurred as well as the algorithms that were executed on that data.

[2] Any errors that occur within an algorithm must be handled by the algorithm itself, unless the intention of the algorithm author is to allow the error to propagate up to the code that invokes the algorithm.

graceful shutdown
: For run-time errors, the default behavior of Phlex is to end the framework program gracefully [DUNE 134]. A graceful shutdown refers to a framework program that completes the processing of all in-flight data, safely closes all open input and output files, cleans up connections to external entities (such as databases), etc. before the program ends. This ensures that no resources are left in ill-defined states and that all output files are readable and valid.

1.2.4 Reusability

The decision to create a new framework is appropriately met with skepticism. However, the selection of which framework design to pursue was strongly guided by past R&D efforts specifically targeted to explore DUNE's framework needs [Meld]. In addition, many ideas already represented in existing production-quality frameworks are also integrated into Phlex's technical design (*under preparation*):

- Existing parallel-processing libraries will be used (e.g. Intel's oneTBB),
- ROOT IO mechanisms will be in place for persisting data,
- The HEP-CCE experience on portability technologies will inform how CPU/GPU source-code portability is achieved.

The chance to develop a new framework also necessitates a re-examination of the knowledge acquired by the broader computing community, and how that knowledge can be applied to data-processing in HEP. Consequently, part of the Phlex design is simply a summary of prior research that has a ready application to DUNE's data-processing needs (e.g. see Chapter 2).

1.3 Programming Languages

The framework will support user *algorithms* written in multiple programming languages [DUNE 14]. Specifically, an algorithm may be written in either C++[3] [DUNE 81] or Python[4] [DUNE 82]. If there is a need to support user algorithms written in another programming language, a corresponding stakeholder requirement should be created.

Note that the language is left unspecified for the implementation of the framework itself.

1.4 Framework Independence

We define an algorithm as *framework-independent* if it contains no explicit dependencies on framework libraries—i.e. it is possible to build and execute the algorithm independent of a framework context. For framework-independent C++ algorithms, this means there are no direct or transitive framework libraries that are either included as headers in the algorithm code or linked as run-time libraries. Similarly, framework-independent Python algorithms import no direct or transitive framework packages.

Phlex is required to support the registration of user-defined, framework-independent algorithms [DUNE 43]. This does not mean that *all* framework-independent algorithms are suitable for registration, nor does it mean that all algorithms registered with the framework must be framework-independent. In fact, depending on what the algorithm is doing, some algorithms might require explicit framework dependencies.

1.5 Guide to Reading This Document

This document is intended to convey Phlex's conceptual design without delving into implementation specifics. We therefore adopt a set of high-level abstractions that model the data-processing the framework will perform. These ideas are discussed in Chapter 2, which summarizes well-known mathematical concepts already used in the computing industry. From there, we discuss the conceptual design itself in Chapter 3.

[3] As of this writing, Phlex supports the C++23 standard.
[4] Phlex adheres to SPEC 0 [SPEC-0] in supporting Python versions and core package dependencies.

Appendices are provided that give definitions (Appendix A) and list stakeholder requirements (Appendix B).

Attention

Any C++ or Python framework interface presented in this document is illustrative and not intended to reflect the final framework interface.

CHAPTER

TWO

HIGH-LEVEL ABSTRACTIONS

With Phlex, the general notion of data-processing must be expressed without relying on the concept of an *event* [DUNE 22]. We therefore avoid the "event" terminology altogether. Instead, we adopt a set of well-known abstractions that can be systematically applied to each of the processing behaviors required of Phlex. These ideas work together, and they are presented here as ingredients necessary for understanding the conceptual design given in Chapter 3.

This chapter is somewhat formal in mathematics. However, the formality used provides crisp descriptions of the data-processing concepts required of Phlex, enabling a computing model that is (a) naturally thread-safe, and (b) allows a close connection between user-defined algorithms and the physics such algorithms are intended to represent.

As will be discussed in Section 2.4, Phlex adopts a *functional programming* paradigm for the construction of workflows. To adequately describe what functional programming is, we first introduce function notation, the concept of the *type*, and mathematical sequences and indexed families. With these ingredients, we are then able to discuss functional programming and how it is supported by data-flow graphs.

2.1 Function Notation

The expression $f : A \to B$ represents the function f that takes an element of the set A to an element of the set B. For example, the function $flip : \mathbb{R} \to \mathbb{R}$ accepts a real number (e.g. 3.14) and multiplies it by -1, returning another real number (e.g. -3.14).

The types A and B are allowed to represent Cartesian products of sets (e.g. $A = A_1 \times \cdots \times A_n$), thus enabling multivariate functions. For example, the following are simple multivariate functions:

$$power : \mathbb{R} \times \mathbb{N} \to \mathbb{R}$$
$$halve : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$

where $\mathbb{N}$ is the set of natural numbers. Invoking $power(1.5, 2)$ results in the real number $1.5^2 = 2.25$, whereas invoking $halve(5)$ divides 5 by 2, returning the pair $(2, 1)$, where the first number is the quotient, and the second is the remainder.

It is conventional to use parentheses to denote the application of the function f to an argument x —i.e. $f(x)$. To avoid cluttered expressions in this document, however, we adopt an alternative whereby the parentheses are omitted and whitespace is used to delimit the function and its argument(s):

$$f(x) \Longrightarrow f\ x$$
$$h(g(y)) \Longrightarrow h\ (g\ y) = h\ g\ y$$

In mathematical expressions, parentheses are then reserved in this document to specify:

1. A tuple of arguments that serve as input to a single function invocation (e.g. $power\ (1.5, 2)$ above)[5].
2. The binding of user-defined algorithms to higher-order functions (see Section 2.5).

[5] The expression $f\ x$ can equivalently be written $f\ (x)$, where (x) is a single-element tuple. With this interpretation, the conventional function-invocation notation can be recovered.

2.2 Types

A reasonable description of a type is that it is a mathematical *set* of objects. For example, the type `int` is an approximation to the mathematical set $\mathbb{Z}$, although there may be technological limitations on what values an object of type `int` can take.

Suppose, however, that an algorithm $p(n_{\text{POT}})$ is configured to operate on an integer n_{POT} that corresponds to the number of protons on target. In such a case, specifying the function p as $p : \mathbb{Z} \to R$ (where R is an arbitrary return type) is too permissive. In a framework context, not all data of type `int` (the equivalent to $\mathbb{Z}$) are suitable for processing by the algorithm p.

In this document, the type therefore refers to a mathematical set that includes more than just the programming language's type `T`; it can also include various labels that identify which kind of `T` is desired.

2.2.1 Boolean Set

The Boolean values *true* (T) and *false* (F) are used frequently in computing. It is convenient to denote a set that contains both values:

$$\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$$

Although *true* and *false* are often represented by 1 and 0, respectively, we use the symbols T and F to avoid implicit comparisons between the members of the set $\mathbb{B}$ and (e.g.) $\mathbb{Z}$. With this convention, $\mathbb{B} \cap \mathbb{Z} = \emptyset$.

2.2.2 Representing `void` and `NoneType`

In HEP, it is common to encounter C++ functions like:

```
void f(int);
double g();
```

where the function either returns nothing (i.e. `void`) or it accepts no argument. Python supports similar behavior for its functions and methods, but using the keyword `None` instead of `void`.

The mathematical set that is used to represent C++'s `void` and Python's `NoneType` is the set $\mathbb{1}$, which contains only one element[6]. The above functions are thus represented in function notation as:

$$f : \mathbb{Z} \to \mathbb{1}\ , \text{ and}$$
$$g : \mathbb{1} \to \mathbb{R}\ .$$

This notation will be used as we discuss the operators required by Phlex's higher-order functions.

The single element of the set $\mathbb{1}$ can also be used to represent the value `nullptr` for C++ pointers (see Section 2.2.3). When necessary we will refer to that single element as the *null value*, or simply the open-closed parentheses `()`.

2.2.3 Representing Optional Types

It is occasionally necessary to represent a "nullable" or "optional" type $\text{Opt}(T)$, whose objects either contain a value type T or are null. Mathematically, this is represented by the coproduct $T \sqcup \mathbb{1}$, where a null or *disengaged* object of type $\text{Opt}(T)$ has a value equal to the single element of the set $\mathbb{1}$.

Table 2.1 gives examples of programming types in various languages that can be mathematically represented by $\text{Opt}(T)$. Although Phlex does not support algorithms written in Haskell, an example of the use of `Maybe T` is given as an illustration of how $\text{Opt}(T)$ is supported outside of C++ and Python.

[6] It is tempting to associate `void` and `NoneType` with the empty set $\emptyset$. However, a function must always be invoked with at least one object, and it must return at least one object. The empty set contains no such objects that can be presented to (or returned from) a function.

Table 2.1: Optional types in Haskell, Python, and C++. Due to Python's dynamic nature, a given name (e.g. `k`) can be bound to any value, thus emulating an optional type.

Language	Type T	Type Opt(T)	Engaged value	Disengaged value
Haskell	`Int`	`m :: Maybe Int`	`m = Just 42`	`m = Nothing`
Python	`int`	*See caption*	`k = 42`	`k = None`
C++	`int`	`std::optional<int> i;`	`i = 42;`	`i = std::nullopt;`
		`int const* j;`	`j = new int{42};`	`j = nullptr;`

2.3 Sequences and Families

A finite sequence can be represented as an ordered list of objects, all of which are from the same set or *type* (see Section 2.2). For example, the sequence $a = [a_1, a_2, \ldots, a_n]$ is written as

$$a = [a_1, a_2, \ldots, a_n] = [a_i]_{i\in\{1,2,\ldots,n\}} = [a_i]_{i\in\mathcal{I}}$$

where each $a_i \in A$, and $\mathcal{I} = \{1, 2, \ldots, n\}$ is the *index set* whose elements are used to identify objects within the sequence a.

The elements a_i can be ordered only if the index set itself is ordered[7]—e.g because the elements of the set $\mathbb{N}$ respect the order $1 < 2 < 3 < \cdots$, the elements of a are listed in that order, and a is, therefore, a sequence. However, if the index set has no ordering associated with it, then instead of a representing a sequence, it represents an *indexed family*, where the ordering of elements is not meaningful. Specifically, an indexed family b can be equivalently represented as:

$$b = [b_1, b_2, \ldots, b_n] = [b_n, \ldots, b_2, b_1] = [b_i]_{i\in\mathcal{I}}$$

or any of the $n!$ orderings possible. The index set $\mathcal{I}$ simply provides a mapping from an index to the data object itself.

Important

The order in which elements are presented to algorithms is not guaranteed in concurrent contexts. Phlex, therefore, processes indexed families of data and not sequences.

In some cases, it is convenient to represent a family's index as a tuple of numbers, thus representing layers of nesting in the data organization (see Section 3.2). As an illustration, to use the RSE hierarchy from Section 1.2.1 one could specify

$$c = [c_i]_{i\in\mathcal{I}_{\text{RSE}}} = [c_{rse}]_{(r,s,e)\in\mathcal{I}_{\text{RSE}}}$$

to refer to a family of data products c, where the variables r, s, and e correspond to the identifiers for specific *Run*s, *Subrun*s, and *Event*s. The values that r, s, and e are permitted to take are determined by the members of the index set $\mathcal{I}_{\text{RSE}}$, which contains triplets of numbers. The semantics of the indices depend on the use case, and it is usually best to denote the semantics along with the indices (e.g.):

$$c = [c_{rse}]_{(\text{Run } r,\ \text{Subrun } s,\ \text{Event } e)\in\mathcal{I}_{\text{RSE}}}$$

This avoids confusion when other data organizations are considered. For presentation purposes, however, we only include explicit semantics whenever it is necessary for describing the data-processing in question (e.g. Section 3.5.5).

The *length* of the family b above is denoted by $|b|$, and it equals the *cardinality* of the index set: $|b| = |\mathcal{I}|$. In this document, we also use the *empty family*, which has length 0, and whose elements can be indexed by the empty set:

$$[\,] = [b_i]_{i\in\emptyset}\ .$$

We also use the notation $\tilde{b}$ to denote a family whose elements are either Boolean *true* (T) or *false* (F), such as $\tilde{b} = [\mathsf{T}, \mathsf{T}, \mathsf{F}, \ldots]$.

[7] Specifically, the index set must be equipped with a strict total order.

2.4 Functional Programming

Functional programming is a paradigm that favors the use of *functions* instead of the direct manipulation of stateful objects. The processing of data happens by using chained operations, where the output of one function serves as the input to other functions.

For example, given two functions:

$$f : \text{Wires} \to \text{Hits}$$
$$g : \text{Hits} \to \text{Tracks}$$

a composite function $h : \text{Wires} \to \text{Tracks}$ can be constructed such that:

$$ts = h\ ws = g\ (f\ ws) = (g \circ f)\ ws$$

or $h = g \circ f$, where $ws \in$ Wires and $ts \in$ Tracks.

In reality, the creation of tracks from wire signals is much more complicated[8]. However, as seen above, functional programming permits a mathematical description of the data-processing to be performed. Expressing the processing needs according to mathematics enables:

- the use of mathematical rules to optimize the processing of the data,
- naturally reproducible results, assuming the functions are pure (see Section 2.4.1),
- parallel invocations of pure functions with no possibility of data races [DUNE 130].

2.4.1 Pure Functions

According to Wikipedia [Wiki-Pure], a pure function has the following properties:

- the function return values are identical for identical arguments, and
- the function has no side effects.

Phlex therefore encourages the use of pure functions for creating of *reproducible* data products, a principle of the framework philosophy as discussed in Section 1.2.

Favor free functions

Functions can additionally be classified as *free functions* or *member functions* (or *methods*). Whereas a free function has only explicit input parameters, a member function called on an object has access to the internal state of the object as well as the explicit function parameters. Both kinds of functions can be useful, but authors of classes must exercise special care to ensure that a class instance's member functions can be safely invoked in concurrent contexts. For this reason, framework users should favor free functions over classes and their member functions.

2.4.2 Challenges with Functional Programming

One drawback to functional programming is that it differs from what many in the HEP community are accustomed to when writing their own physics algorithms. Commonly used third-party libraries and computing languages can also make functional programming difficult to use in practice. We argue, though, that physicists often think in terms of functional programming when developing the high-level processing steps of a workflow. It is not until those processing steps need to be implemented that the functional steps are often translated into *procedural* ones.

Phlex aims to restore the functional programming approach as the natural way of expressing the data-processing to be performed. By leveraging commonly used processing patterns (see Section 2.5 on higher-order functions), we can mitigate any awkwardness due to initial unfamiliarity with functional programming paradigms. The framework

[8] As discussed in Section 2.6, the general topology of a data-processing workflow is a *graph* of functions.

also does not place constraints on the algorithm *implementations*—algorithm authors are free to employ imperative programming techniques within the implementations if doing so is convenient. The framework will simply schedule the algorithm as if it were a pure function without regard to its implementation.

2.5 Families of Data and Higher-Order Functions

Particle physics results are obtained by performing statistical analysis on families of data. Such analysis typically involves repeated invocations of the same kind of operation. For example, a relatively simple result is calculating the arithmetic mean of n numbers:

$$\overline{c} = \frac{1}{n} \sum_{i \in \mathcal{I}} c_i$$

where the sum is over the numbers $[c_i]_{i \in \mathcal{I}}$, and n is the cardinality of the index set $\mathcal{I}$.

The sum is an example of a data reduction or *fold*, where a family is collapsed into one result. In particular, the arithmetic mean above can be expressed as:

$$\overline{c} = \frac{1}{n} \text{ fold}(+,\ 0)\ \ [c_i]_{i \in \mathcal{I}}$$

where the fold accepts a binary operator ($+$ in this case) that is repeatedly applied to an accumulated value (initialized to 0) and each element of the family.

The fold is an example of a *higher-order function* (HOF) [Wiki-HOF] that accepts a family and an operator applied in some way to elements of that family. However, additional HOFs exist—for example, suppose the family $[c_i]$ was created by applying a function $w : E \to C$ to each element of a family $[e_i]$. Such a HOF is called a map or *transform*:

$$[c_i] = [w\ e_i] = \text{transform}(w)\ \ [e_i]$$

In such a scenario, the average $\overline{c}$ could be expressed as:

$$\overline{c} = \frac{1}{n} \text{ fold}(+,\ 0) \text{ transform}(w)\ \ [e_i] = \frac{1}{n} \text{ fold}(+ \circ w,\ 0)\ \ [e_i]$$

The second equality holds by the fold-map fusion law [Bird], which states that the application of a transform followed by a fold can be reduced to a single fold. The operator to this single fold is '$+ \circ w$', indicating that the function w should be applied first before invoking the $+$ operation. Relying on such mathematical laws permits the replacement of chained calculations with a single calculation, often leading to efficiency improvements without affecting the result.

Table 2.2: Higher-order functions supported by Phlex. Each family is represented by a single variable (e.g. a). Details of each HOF and its operators are in Section 3.5.

Higher-order function		Operator(s)	Output family length
Transform	$b = \text{transform}(f)\ a$	$f : A \to B$	$\lvert b\rvert = \lvert a\rvert$
Predicate	$\tilde{b} = \text{predicate}(f)\ a$	$f : A \to \mathbb{B}$	$\lvert\tilde{b}\rvert = \lvert a\rvert$
Filter	$a' = \text{filter}(\phi)\ a$	$\phi : \mathbb{B}^n \to \mathbb{B}$	$\lvert a'\rvert \leq \lvert a\rvert$
Observer	$[\ ] = \text{observe}(f)\ a$	$f : A \to \mathbb{1}$	0
Fold	$d = \text{fold}(f,\ init,\ part)\ c$	$f : D \times C \to D$	$\lvert d\rvert \leq \lvert c\rvert$
		$init : \text{Opt}(\mathcal{I}_d) \to D$	
		$part : \{\mathcal{I}_c\} \to \mathbb{P}(\mathcal{I}_c)$	
Unfold	$c = \text{unfold}(p,\ gen,\ label)\ d$	$p : N \to \mathbb{B}$	$\lvert c\rvert \geq \lvert d\rvert$
		$gen : N \to N \times C$	
		$label : \mathbb{1} \to L$	
Window	$y = \text{window}(f,\ adj,\ label)\ x$	$f : X \times \text{Opt}(X) \to Y$	$\lvert y\rvert = \lvert x\rvert$
		$adj : \mathcal{I}_x \times \mathcal{I}_x \to \mathbb{B}$	
		$label : \mathbb{1} \to L$	

A calculation using a HOF is then generally expressed in terms of:

1. The HOF to be used (fold, transform, etc.)
2. The operator(s) to be used by each HOF ($+$, w, etc.)
3. The family (or families) of data on which the HOFs are to be applied.

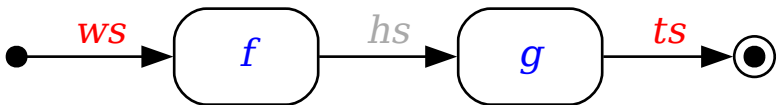
Phlex supports the HOFs listed in Table 2.2. As discussed later, each HOF's *operator* is an *algorithm* registered with the framework. Phlex will likely support other higher order functions as well.

2.6 Data Flow

In Section 2.4, the example was given for creating tracks from wires. The simplified expression for doing this was the chained application of two functions f and g such that:

$$ts = g\ (f\ ws) = (g \circ f)\ ws$$

where ws and ts are the wires and tracks, respectively. An alternative representation is a *directed graph*



where the functions f and g are vertices or *nodes* and the data ws and ts are passed along arrows or *edges* that connect the nodes[9]. The arrows indicate the flow of the data (or *data flow*) through the graph.

Some observations:

- The function f returns hits that are unlabeled in the equational form. For the graph above, we explicitly label those hits as hs to emphasize that data are always passed along edges. If, however, the hits are not needed by any other function in the graph, the two functions f and g can be replaced by their composition $h = g \circ f$, resulting in potential performance improvements in computational efficiency and program memory usage.

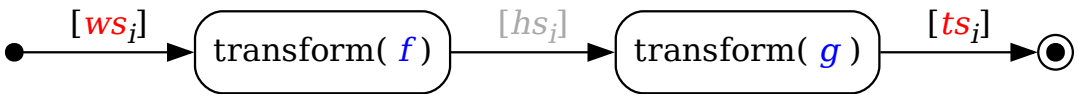
- Each edge of a directed graph must have a source (the tail of the arrow) and a target (the head of the arrow). This means that whereas the equation does not need to specify where the ws wires originate (or where the ts tracks end up), such locations must be specified in the graph. Any node that has only outgoing edges is called a *source* (denoted by a solid dot), and any node that has only incoming edges is a *sink* (denoted by an encircled solid dot).

[9] It is also possible to invert the view of the graph such that the data are treated as nodes and the functions as edges; such an approach yields a data-centric graph, which is the *line digraph* of the function-centric graph shown above.

2.6.1 Data Flow with Families

As mentioned earlier in Section 2.5, processing families of data is a critical aspect of obtaining physics results. The data-flow discussion in the previous section naturally maps to applying the functions f and g to elements of families. Specifically[10]:

- The families $[ws_i]$, $[hs_i]$, and $[ts_i]$ are passed along the edges instead of the individual objects ws, hs, and ts.
- The functions f and g map to transform(f) and transform(g), respectively.



The above graph does not specify an implementation—assuming f and g are pure functions (see Section 2.4.1), the same result is obtained if (a) full families are passed between the nodes, or (b) one element per family is passed at a time. Determining whether option (a) or (b) is more efficient depends on the data passed between the nodes and the overall constraints on the program.

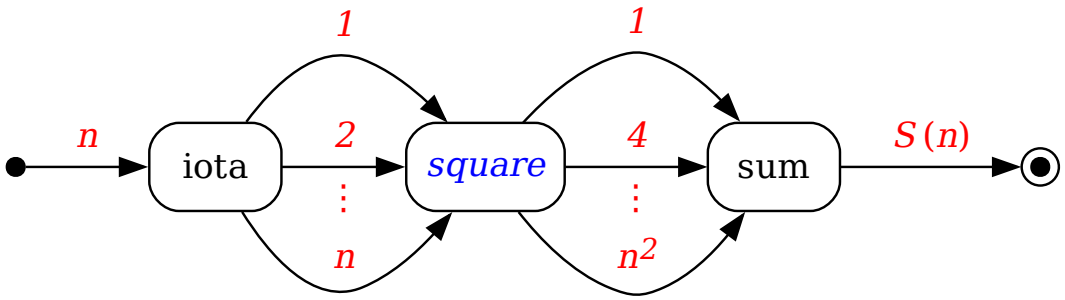
One benefit to using a graph representation using data families and higher-order functions is the ability to easily express folds and unfolds. As an illustration, consider the calculation of a sum of squares for all integers from 1 through n:

$$S(n) = \sum_{i=1}^{n} i^2$$

This calculation requires three separate steps:

1. an unfold called *iota* that, given an integer n, generates a sequence of integers from 1 through n,
2. a transformation that squares each integer in the sequence using an algorithm called *square*, and
3. a fold called *sum* that sums all squared integers.

The data-flow graph of individual objects looks like:



where each number is passed along its own edge to the nodes performing the calculation.

[10] In Haskell (and, similarly, in category theory), this transformation is achieved through the *List/fmap* functor.

The summation formula for $S(n)$, however, can be expressed in terms of higher-order functions that reflect the three steps above:

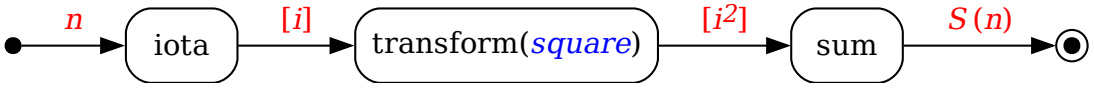
$$S(n) = \sum_{i=1}^{n} i^2 = \underbrace{\text{fold}(+,\ 0)}_{\text{3. sum}}\ \underbrace{\text{transform}(\textit{square})}_{\text{2.}}\ \underbrace{\text{unfold}(\textit{greater\_than\_zero},\ \textit{decrement})}_{\text{1. iota}}\ n$$

where:

1. iota or unfold($\textit{greater\_than\_zero}$, $\textit{decrement}$) is a function that returns a sequence given a value of n,
2. transform(*square*) is a function applied to the sequence generated in step 1.
3. sum or fold($+$, 0) is a function that returns a single result when applied to the result of step 2.

Note that the unfold takes two operators: the predicate $\textit{greater\_than\_zero}$, which tests whether the next generated number is greater than zero, and the $\textit{decrement}$ operator, which decrements the next generated number by 1.[11] Once the predicate returns false, the unfold stops generating numbers.

By adopting the HOF representation of the sum-of-squares problem, the data-flow graph is simplified:



In addition, the topology of the family-based graph remains the same regardless of the value of n; the topology of the object-based graph becomes intractable as n increases.

The vast majority of graphs included in this design document adopt the family-based data-flow representation.

[11] The same data flow can equivalently be represented with an unfold that uses the predicate $\textit{less\_than\_or\_equal\_to\_n}$ and the operator *increment*, but starts with an initial value of 1. This other representation, however, requires an unfold predicate that depends on the value of n.

CHAPTER

THREE

CONCEPTUAL DESIGN

Purpose

The conceptual design is not a reference manual; it is a high-level description of how the framework aims to satisfy the stakeholder requirements (see Appendix B). The audience for the conceptual design is the physicist, algorithm author, or framework program runner. More detailed design aspects in support of the conceptual model are given in the technical design *(under preparation)*.

Phlex adopts the data-flow approach discussed in Section 2.6.1. Instead of expressing scientific workflows as monolithic functions to be executed, workflows are factorized into composable algorithms that operate on data products passed among them [DUNE 1], [DUNE 111], [DUNE 20]. These algorithms then serve as operators to higher-order functions (HOFs) that operate on *data-product families*.

To guide the discussion of Phlex's conceptual model, we refer to Fig. 3.1, which shows a small fictitious workflow that creates vertices from simulated energy deposits. Various framework aspects are demonstrated by that figure:

data-flow graph
: The data-flow graph is formed by ingesting the configuration file and recording the data-product dependencies required of each algorithm (see Section 3.1).

data-product flow
: Data products (see Section 3.2) are passed along graph edges. As mentioned in Section 2.6.1, the data passed between HOFs are expressed as families. Fig. 3.1 thus formally passes families (e.g. $[GoodHits_{ijk}]$) between nodes[12].

framework driver
: The driver instructs the framework what to process (see Section 3.6).

  The driver in Fig. 3.1 is configured so that all *Spill*s in the specified ROOT input files are processed.

data-product providers
: Data-product providers are framework components that provide data products from external entities to downstream user algorithms (see Section 3.7). From a functional programming perspective they are transforms that map a data cell to one of the data products within that data cell.

  In the workflow, one provider reads a *SimDepos* data product from each *Spill* in the HDF5 input files, and the other reads a single *Geometry* corresponding to the *Job* from a GDML file.

HOFs and user-provided algorithms
: Arguably the most important aspect of the framework is how user-provided algorithms are bound to HOFs and registered with the framework (see Section 3.3, Section 3.4 and Section 3.5).

[12] In practice, elements of the family, not the full family itself, will be passed from one node to another. For memory purposes, it is also likely that each element of the family is a lightweight reference (in C++, a pointer) to the data of relevance.

All seven HOFs supported by Phlex (see Table 2.2) are used in Fig. 3.1. For the main processing chain of creating vertices:

- An *unfold* HOF is configured to create a family of *Waveforms* objects—creating one *Waveforms* object per *APA*—from one *SimDeps* data product in each *Spill*.
- A configured *transform* HOF is run on the family of *Waveforms* objects to create a family of *GoodHits* objects.
- To make a *GoodTracks* data product, a *window* algorithm is applied to pairs of *GoodHits* objects that come from adjacent *APA*s.
- Lastly, another *transform* algorithm operates on the *GoodTracks* data products to produce vertices.

There are additional parts of the graph that are not directly related to creating vertices:

- A *fold* algorithm is executed over the *GoodHits* data products to sum the hit energy (i.e. *TotalHitEnergy*) across all *APA*s for a given *Spill*.
- After a *filter* has been applied with the *predicate high_energy*, an *observe* algorithm is used to fill a histogram with hit-related information from the *GoodHits* data products.

data-product writers

Data-product writers are plugins that write data products to framework outputs (see Section 3.8)

Each of the five writers in Fig. 3.1 is responsible for writing to one or more output files.

resources

Most workflows require access to some external resource (see Section 3.9).

The histogramming resource in Fig. 3.1 enables the *observe* algorithm to fill and write histograms to a ROOT analysis file.

Note that in this workflow, the names *Spill* and *APA* are not special to the Phlex framework; they are names (hypothetically) chosen by the experiment. Each data product is also indexed, thus associating it with a particular data cell (e.g. $\mathit{GoodHits}_{3,5,9}$ denotes the *GoodHits* data product belonging to *APA* 9 of *Spill* 5 of *Run* 3).

3.1 Topology of the Data-Flow Graph

The graph shown in Fig. 3.1 is a close reflection of the graph used for scheduling algorithm execution. It is formed from multiple ingredients, the most crucial of which is the program configuration (see Section 3.10). It is the program configuration that specifies:

- the driver to use,
- the algorithms needed for creating, filtering, and observing data products,
- the specification of data products required and (where supported) created by each algorithm,
- which data products to write to output files,
- which data products to read from input files,
- any resources needed to process the data, and
- any other constraints to impose on the program.

The pattern used for constructing the data-flow graph is described in detail in the technical design *(under preparation)*. Here it is sufficient to note that the graph is primarily driven by which data products are specified to be written to output (i.e. *precious* data products). By default, the framework will schedule for execution only those algorithms needed to write precious data products. This means that not necessarily every algorithm in the program configuration will be used in a general workflow.

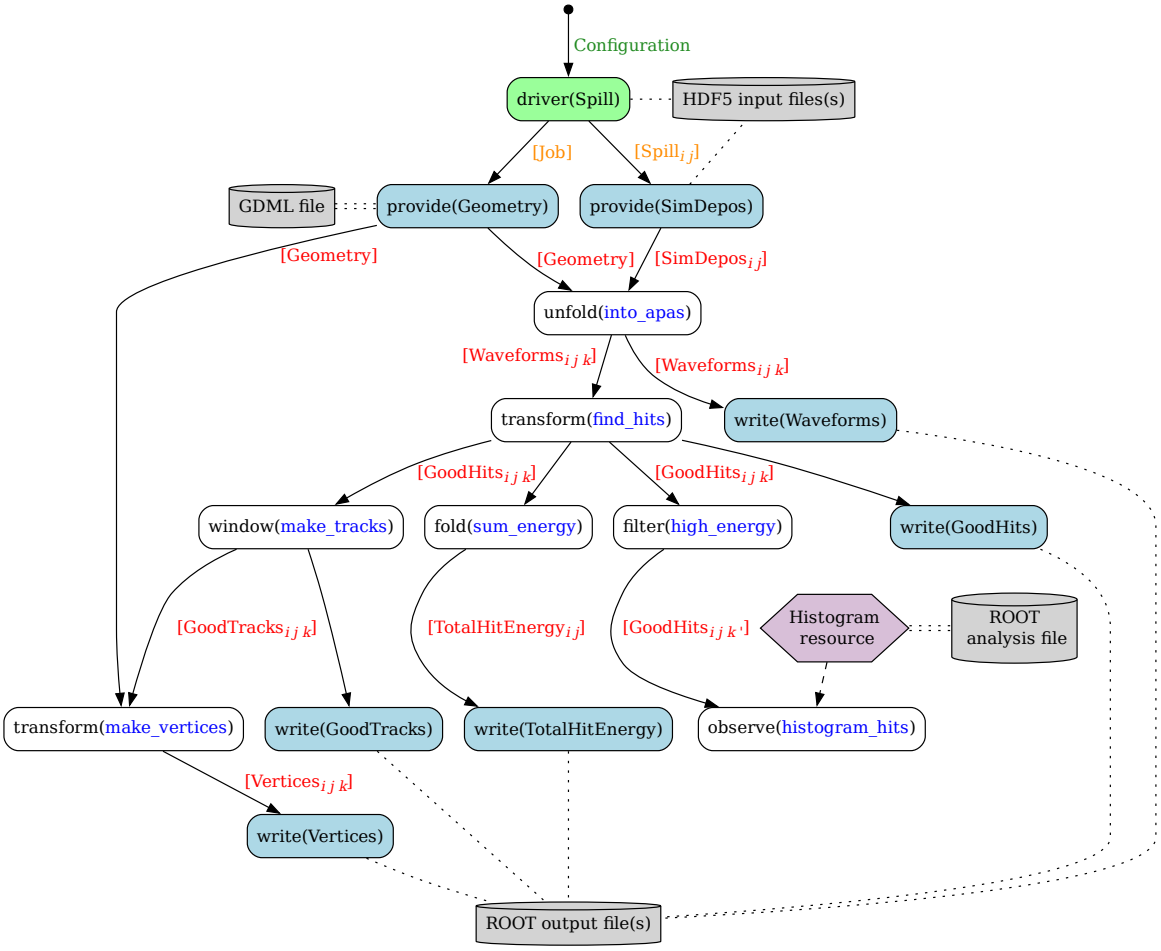


Fig. 3.1: A fictitious workflow showing how HOFs are used in a Phlex program. Each unshaded node represent a HOF bound to a user-defined algorithm, whose name is shaded in blue. Each user-defined algorithm operates on arguments received from the incoming arrows to the node: data products are passed along solid arrows; objects that provide access to resources are passed along dashed arrows. Whereas single-dotted lines indicate communication of data through the framework's IO system, double-dotted lines denote communication of data with entities not directly related to the framework. See text for workflow details.

Each input file is able to report which data products it contains, and each algorithm-registeration statement (see Section 3.4) declares which data products are required [DUNE 65] and which data products are created [DUNE 156] by the algorithm. These data-product dependencies, along with the dependencies implied by the resources, are sufficient to establish the data-flow graph.

3.2 Data Organization

This section provides a conceptual overview of *data products*, *data cells*, *data layers*, and *data-layer hierarchies*, as defined in [DUNE 85] [DUNE 86] [DUNE 87] [DUNE 88]. In addition, we discuss *data-product families* and *data-cell families*. This section aims to establish a mental model for how all of these concepts facilitate scientific workflows without delving into implementation specifics.

Data products represent things like raw detector readouts, calibration information, and derived physics quantities. [DUNE 40]. We call these kinds of things represented by data products *conceptual data products*. Data product types are the programming language representations of conceptual data products. A data layer is an experiment-defined level of aggregation of data products. Some example data layers are *Run*, *Subrun*, *Spill*, and an interval of validity for some flavor of calibration. Each Phlex job includes a *Job* data layer at the top of the data-layer hierarchy. A data cell is a collection of data products, associated with a data layer. A data-cell family is a family of data cells that are in the same data layer. The *Job* layer always includes a single data cell. Fig. 3.2 illustrates the relationships between all of these.

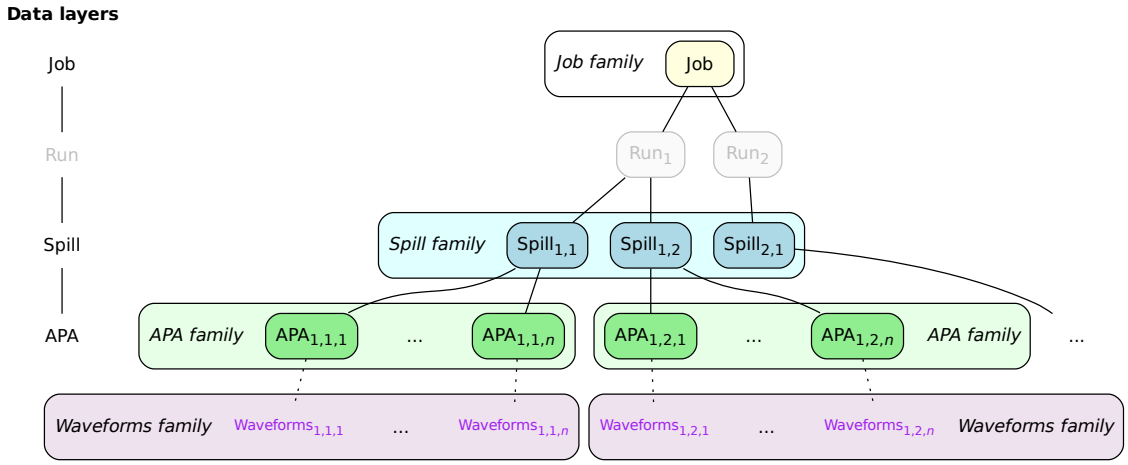


Fig. 3.2: The data organization corresponding to part of Fig. 3.1. The framework-provided *Job* data layer and three different user-defined (not special to the Phlex framework) data layers are shown: *Run*, *Spill*, and *APA*. Rectangles with labels Run_i, $\mathsf{Spill}_{i,j}$, and $\mathsf{APA}_{i,j,k}$ represent data cells. The pale green rectangles show two data-cell families; these are identified as families because they are the result of executing the *unfold(into_apas)* node shown in Fig. 3.1. A solid line from one data cell to another data cell represents a logical association between the two data cells. The bottom rectangle shows that $\mathsf{Waveforms}_{1,1,1}$ is in the data cell $\mathsf{APA}_{1,1,1}$, etc. Each pale purple rectangle indicates the data-product family created by unfolding each *SimDepos* object as shown in Fig. 3.1.

In Fig. 3.2, the *Run* data layer exists, but as no algorithm in Fig. 3.1 requires any data products from a *Run* data cell, the framework does not create any data-cell families corresponding to the *Run*.

3.2.1 Data Products

Data products are entities that encapsulate processed or raw data, of all kinds, separate from the algorithms that create them [DUNE 110]. They serve as the primary medium for communication between algorithms, ensuring seamless data exchange across processing steps [DUNE 111]. They are associated with (rather than containing) metadata and provenance information that describe how the data products were created [DUNE 121]. They are not tied to specific hardware or algorithm implementations, ensuring independence and reproducibility [DUNE 63]. They are also not tied to any specific IO back end, but must support reading and writing with both ROOT [DUNE 74] and HDF5 [DUNE 141]. They enable the framework to present data produced by one algorithm to subsequent algorithms, supporting iterative and chained processing workflows [DUNE 20].

3.2.1.1 Structure and Representation

The in-memory layout of a data product is determined by its type in the specified programming language. Phlex does not require that the in-memory representation of a data product shall be the same as its persistent representation [DUNE 2]. In general, a single conceptual data product can be represented by multiple programming language types. This includes representing a single conceptual data product in multiple supported programming languages.

The framework provides the ability to determine the memory footprint of each data product [DUNE 154].

3.2.1.2 Defining Data Product Types

Data product types are not defined by the framework. Framework users are expected to define their own data product types [DUNE 85].

3.2.2 Data Layers, Data Cells, and Families

As illustrated in Fig. 3.2, data products are organized into user-defined data cells, families, layers, and hierarchies, supporting varying levels of granularity [DUNE 86] [DUNE 87] [DUNE 88]. They can be unfolded into finer-grained units, enabling detailed analysis or reprocessing at different scales [DUNE 43]. This provides the ability to process data too large to fit into memory at one time [DUNE 25].

3.2.3 Data Product Management

Management of the data products returned by an algorithm is taken over by the framework. Read-only access to input data products is provided to algorithms [DUNE 121] [DUNE 130]. Read-only access to a data product must not mutate it. Data products that are intended to be written out are sent to the IO system as soon as they are created [DUNE 142]. Data products are removed from memory as soon as they are no longer needed for writing or as input to another algorithm [DUNE 142].

3.2.4 Data Product Identification

Each data product is associated with a specific set of metadata describing the algorithms and configurations used in their creation. These metadata allow that creation to be reproducible [DUNE 122]. The metadata are stored along with the data in the framework output file, and the IO interface allows access to the metadata [DUNE 121].

The data products created by an algorithm are associated with metadata that identify the algorithm that created them. Such metadata include:

- the *creator*, the name of the algorithm that created the data product
- an identifier for the *data cells* with which the data product is associated (e.g. *Spill*, *Run*, *Calibration Interval*, or other experiment-defined layer)
- the *processing phase*, an identifier for the job in which the data product was created
- an individual *name* for the data product (which may be empty), to distinguish between multiple products of the same type created by the same algorithm.

In addition to these metadata, a data product is also specified by its *type*.

The metadata are stored in the framework output file, and the IO interface allows access to the metadata [DUNE 121].

The metadata are also used in data product lookup, to specify which data products are to be provided as inputs to an algorithm. The algorithms are configured to identify the inputs in which they are interested by selecting on any of the metadata defined above, as well as by the programming language types of their inputs.

3.3 Algorithms

As mentioned in Section 2.5, an algorithm is registered with the framework as an operator to a higher-order function (HOF). In general, Phlex supports the registration of C++ algorithms with function signatures like (see Section 3.5 for a list of supported HOFs):

```
return_type function_name(P1, Pn..., Rm...) [quals];
```

where the types `P1, Pn...` denote types of data products and the types `Rm...` indicate *resources*. The bracketed `[quals]` term indicates that Phlex allows for class member functions that have trailing qualifiers (e.g. `const`). Each registered function must accept at least one data product.

The signature of a Python algorithm needs to be available through reflection, either because the function is JITed (e.g. with *Numba*), bound (e.g. with *ctypes*), or annotated. The latter is good practice regardless and commonly required by Python coding conventions:

```
def function_name(p1: P1, pn: Pn..., rm: Rm...) -> return_type:
```

We will first discuss the data-product and resource types in Section 3.3.1, followed by the return types in Section 3.3.2, and then the function name and optional qualifers in Section 3.3.3.

3.3.1 Input Parameters

A data product of type `P` may be presented to a C++ algorithm if the corresponding input parameter (i.e. the relevant `P1, ..., PN` type) is one of the following:

- `P const&` — read-only access to a data product provided through a reference
- `P const*` — read-only access to a data product provided through a pointer
- `P` — the data product is copied into an object (assumes data product is copyable)[13]
- `phlex::handle<P>` — a lightweight object that provides read-only access to a data product as well as any metadata associated with it

For each of these cases, the data product itself remains immutable. A Python algorithm can receive a *phlex::handle* or a direct reference to the data product. There is no equivalent language support for read-only access, but it will be enforced where possible.

Whereas data products may be copied, resources of type `R` may not. The following types are therefore supported:

- `R const&` — read-only access to a resource provided through a reference
- `R const*` — read-only access to a resource provided through a pointer
- `R&` — read-and-write access to a resource provided through a reference (if supported by resource)
- `R*` — read-and-write access to a resource provided through a pointer (if supported by resource)

Resources are described in more detail in Section 3.9.

[13] In C++, the function signature corresponds to the function *declaration* [Cpp-Function], for which the type `P` and `P const` are treated identically by the compiler. However, for the function implementation or *definition*, algorithm authors are encouraged to use `P const` to help guarantee the immutability of data.

3.3.2 Return Types

The meaning of an algorithm's return type depends on the HOF and is discussed in the Section 3.5. However, to simplify the discussion we introduce to concept of the *created data-product type*. For Phlex to appropriately schedule the execution of algorithms and manage the lifetimes of data products, the framework itself must retain ownership of the data products. This means that the data products created by algorithms must have types that connote unique ownership. An algorithm's returned object must therefore model a created data-product type, which can be:

- a *value* of type `T`, or
- a `std::unique_ptr<T>`, where the created object is non-null.

For Python, this means that an algorithm should not retain any external hard references to a returned object.

The following types (or their equivalents) are forbidden as created data-product types because they do not imply unambiguous ownership:

- *bare pointer types*, such as `T*` or `T const*`
- *reference types*, such as `T&` or `T const&`

3.3.3 Function Names and Qualifiers

The `function_name` in Section 3.3 above may be any function name supported by the C++ language. Code authors should aim to implement algorithms as free functions. However, in some cases it may be necessary for class member functions to be used instead. When member functions are required, the qualifier `const` should be specified to indicate that the class instance remains immutable during the execution of the member function[14].

3.4 Framework Registration

Consider the following C++ classes and function:

```
class hits { ... };
class waveforms { ... };

hits find_hits(waveforms const& ws) { ... }
```

where the implementations of `waveforms`, `hits`, and `find_hits` are unspecified. Suppose a physicist would like to use the function `find_hits` to transform a data product labeled `"Waveforms"` to one labeled `"GoodHits"` for each spill with unlimited concurrency. This can be achieved by in terms of the C++ *registration stanza*:

```
PHLEX_REGISTER_ALGORITHMS()   // <== Registration opener (w/o configuration object)
{
  products("GoodHits") =      // 1. Specification of output data product from find_hits
    transform(                // 2. Higher-order function
      "hit_finder",           // 3. Name assigned to HOF
      find_hits,              // 4. Algorithm/HOF operation
      concurrency::unlimited  // 5. Allowed CPU concurrency
    )
    .family(
      "Waveforms"_in("APA")   // 6. Specification of input data-product family (see text)
    );
}
```

[14] Phlex permits the registration of member functions that do not use the `const` qualifier. However, using such member functions is highly discouraged as it indicates that a class instance is modifiable during member-function execution, thus placing significant demands on the code author to ensure thread-safe code execution (see Section 2.4.1).

The registration stanza is included in a C++ file that is compiled into a *module*, a compiled library that is dynamically loadable by Phlex.

A Python algorithm can be registered with its own companion C++ module or through the Python import helpers that make use of a pre-built, configurable, Phlex module. For the sake of consistency and ease of understaning, the helpers have the same naming and follow the same conventions as the C++ registration.

The stanza is introduced by an *opener*—e.g. `PHLEX_REGISTER_ALGORITHMS()`—followed by a *registration block*, a block of code between two curly braces that contains one or more *registration statements*. A registration statement is a programming statement that closely follows the equation described in Section 3.5 and is used to register an algorithm with the framework.

$$[b_i]_{i\in\mathcal{I}_{\text{output}}} = \text{HOF}(f_1,\ f_2,\ \dots)\ [a_i]_{i\in\mathcal{I}_{\text{input}}}$$

Specifically, in the registration stanza above, we have the following:

`products(...)`

1. This is the equivalent of the output family $[b_i]_{i\in\mathcal{I}_{\text{output}}}$, which is formed from specification(s) of the data product(s) created by the algorithm [DUNE 156]. One of the fields of the data-product specification is the data layer to which the data products will belong [DUNE 90]. Phlex does not require the output and input data layers to be the same.

`transform(...)`

Fully specifying the mathematical expression $\text{HOF}(f_1,\ f_2,\ \dots)$ requires several items:

2. The HOF to be used,
3. The name to assign to the configured HOF,
4. The algorithm/HOF operator(s) to be used (i.e. $f_1,\ f_2,\ \dots$), and
5. The maximum number of CPU threads the framework can use when invoking the algorithm [DUNE 152].

`family(...)`

6. The specification of the input family $[a_i]_{i\in\mathcal{I}_{\text{input}}}$ requires (a) the specification of data products that serve as input family elements [DUNE 65], and (b) the label of the data layer in which the input data products are found. In the registration code above, this is achieved by providing the expression `"Waveforms"_in("APA")`, which instructs the framework to create a family of waveforms that reside in *APA*s[15].

The set of information required by the framework for registering an algorithm largely depends on the HOF being used (see the Section 3.5 for specific interface). However, in general, the registration code will specify which data products are required/produced by the algorithm [DUNE 111] and the hardware resources required by the algorithm [DUNE 9]. Note that the input and output data-product specifications are matched with the corresponding types of the registered algorithm's function signature. In other words:

- `"Waveforms"` specifies a data product whose C++ type is that of the first (and, in this case, only) input parameter to `find_hits` (i.e. `waveforms`).
- `"GoodHits"` specifies a data product whose C++ type is the `hits` return type of `find_hits`.

When executed, the above code creates a *configured higher-order function*, which serves as a node in the function-centric data-flow graph.

The registration block may contain any code supported by C++. The block, however, must contain a registration statement to execute an algorithm.

[15] The token `_in` is a suffix that is part of a user-defined literal [Cpp-UserLiteral], which permits an expression like `"Waveforms"_in("APA")`. The type returned by the expression is implementation-defined and has no public interface needed by the user.

Important

A module must contain only one registration stanza. Note that multiple registration statements may be made in each stanza.

3.4.1 Algorithms with Multiple Input Data Products

The registration example given above in Section 3.4 creates an output family by applying a one-parameter algorithm `find_hits` to each element of the input family, as specified by `family("Waveforms"_in("APA"))`. In many cases, however, the algorithm will require more than one data product. Consider another algorithm `find_hits_subtract_pedestals`, which forms hits by first subtracting pedestal values from the waveforms, both of which are presented to the algorithm as data products from the *APA*. The interface of the algorithm and its registration would look like:

```
class hits { ... };
class waveforms { ... };
class pedestals { ... };
hits find_hits_subtract_pedestals(waveforms const&, pedestals const&) {...}

PHLEX_REGISTER_ALGORITHMS(config)
{
  products("GoodHits") =
    transform("find_hits", find_hits_subtract_pedestals, concurrency::unlimited)
    .family("Waveforms"_in("APA"), "Pedestals"_in("APA"));
}
```

The elements of the input family are thus pairs of the data products labeled `"Waveforms"` and `"Pedestals"` in each APA.[16] In this case, the data cell for both data products is the same—i.e. for a given invocation of `find_hits_subtract_pedestals`, both data products will be associated with the same APA.

There are cases, however, where an algorithm needs to operate on data products from *different* data cells [DUNE 89].

Note

The number of arguments presented to the `family(...)` clause must match the number of input parameters to the registered algorithm. The order of the `family(...)` arguments also corresponds to the order of the algorithm's input parameters.

3.4.1.1 Data Products from Different Data Layers

Consider the operator *make_vertices* in Fig. 3.1 that requires two arguments: the *GoodTracks* collection for each APA (data layer *APA*), and the detector *Geometry* that applies for the entire job (data layer *Job*)[17]. This would be expressed in C++ as:

```
vertices make_vertices(tracks const&, geometry const&) { ... }

PHLEX_REGISTER_ALGORITHMS(config)
{
  products("Vertices") =
```

(continues on next page)

[16] The operation that forms the family $[(Waveforms_i, Pedestals_i)]_{i \in \mathcal{I}_{\text{APA}}}$ from the separate families $[Waveforms_i]_{i \in \mathcal{I}_{\text{APA}}}$ and $[Pedestals_i]_{i \in \mathcal{I}_{\text{APA}}}$ is called *zip*.

[17] As shown in Fig. 3.2, there is a *Job* data layer, to which job-level data products may belong.

(continued from previous page)

```
    transform("vertex_maker", make_vertices, concurrency::unlimited)
    .family("GoodHits"_in("APA"), "Geometry"_in("Job"));
}
```

where the data layers are explicit in the family statement.

Phlex supports such uses cases [DUNE 113], even if the specified data layers are unrelated to each other. For example, suppose an algorithm needed to access a data product from a *Spill*, and it also required a calibration offset provided from an external database table [DUNE 35]. Instead of providing a separate mechanism for handling calibration constants, a separate layer could be invented (e.g. *Calibration*) whose data cells corresponded to intervals of validity. So long as a relation can be defined between specific *Spill* data cells and specific *Calibration* data cells, the framework can use that relation to form the input family of *Spill-Calibration* data-product pairs that are presented to the algorithm. How the relation between data cells is defined is referred to as *data marshaling*, and it is described further in the technical design *(under preparation)*.

3.4.1.2 Data Products from Adjacent Data Cells

In some cases, it may be necessary to simultaneously access data products from adjacent data-products sets [DUNE 91], where *adjacency* is defined by the user [DUNE 92]. The notion of adjacency can be critical for (e.g.) time-windowed processing (see Section 3.5.7), where the details of the "next" time bin are needed to accurately calculate properties of the "current" time bin.

Supporting the processing of adjacent data cells is described further in the technical design *(under preparation)*.

3.4.2 Accessing Configuration Information

Instead of hard-coding all pieces of registration information, it is desirable to specify a subset of such information through a program's run-time configuration. To do this, an additional argument (e.g. `config`) is passed to the registration opener:

```
PHLEX_REGISTER_ALGORITHMS(config)
{
  auto selected_data_layer = config.get<std::string>("data_layer");

  products("GoodHits") =
    transform("hit_finder", find_hits, concurrency::unlimited)
    .family("Waveforms"_in(selected_data_layer));
}
```

> **Note**
>
> As discussed in the technical design *(under preparation)*, the registration code will have access only to the configuration relevant to the algorithm being registered, and to certain framework-level configuration such as debug level, verbosity, or parallelization options.

Except for the specification of `find_hits` as the algorithm to be invoked, and `transform` as the HOF, all other pieces of information may be provided through the configuration.

3.4.3 Framework Dependence in Registration Code

Usually, classes like `waveforms` and `hits` and algorithms like `find_hits` are framework-independent (see Section 1.4). There may be scenarios, however, where dependence on framework interface is required, especially if framework-specific metadata types are used by the algorithm. In such cases, it is strongly encouraged to keep framework dependence within the module itself and, more specifically, within the registration stanza. This can be often achieved by registering closure objects that are generated by lambda expressions.

For example, suppose a physicist would like to create an algorithm `find_hits_debug` that reports a spill number when making tracks. By specifying a lambda expression that takes a `phlex::handle<waveforms>` object, the data product can be passed to the `find_hits_debug` function, along with the spill number from the metadata accessed from the handle:

```
hits find_hits_debug(waveforms const& ws, std::size_t apa_number) { ... }

PHLEX_REGISTER_ALGORITHMS(m)
{
  products("GoodHits") =
    transform(
      "hit_finder",
      [](phlex::handle<waveforms> ws) { return find_hits_debug(*ws, ws.id().number()); },
      concurrency::unlimited
    )
    .family("Waveforms"_in("APA"));
}
```

The lambda expression *does* depend on framework interface; the `find_hits_debug` function, however, retains its framework independence.

3.4.4 Member Functions of Classes

In some cases, it may be necessary to register a class and its member functions with the framework. This is done by first creating an instance of the class by invoking `make<T>(args...)`, where `T` is the user-defined type, and `args...` are the arguments presented to `T`'s constructor. For example, the `find_hits` algorithm author could have instead created a `hit_finder` class, whose constructor takes a parameter called `sigma_threshold`:

```
class hit_finder {
public:
  hit_finder(float sigma_threshold);
  hits find(waveforms const& ws) const;
  ...
};

PHLEX_REGISTER_ALGORITHMS(config)
{
  auto sigma_threshold = config.get<float>("sigma_threshold");
  auto selected_data_layer = config.get<std::string>("data_layer");

  products("GoodHits") =
    make<hit_finder>(sigma_threshold)  // <= Make framework-owned instance of hit_finder
      .transform("hit_finder", &hit_finder::find, concurrency::unlimited)
      .family("Waveforms"_in(selected_data_scope));
}
```

Note that the `hit_finder` instance created in the code above is *owned by the framework*. The `hit_finder::find`

member function's address is registered in the `transform(...)` clause, thus instructing the framework to invoke `find`, bound to the framework-owned `hit_finder` instance.

> **Note**
>
> Algorithm authors should first attempt to implement algorithms as free functions (see Section 2.4.1). Registering class instances and their member functions with the framework should only be considered when:
>
> - multiple processing steps must work together, relying on shared internal data, or
> - supporting legacy code that relies on object-oriented design.

3.4.5 Overloaded Functions

Phlex performs a substantial amount of type deduction through the `transform(...)` clause. This works well except in cases where the registered algorithms are overloaded functions. For example, suppose one wants to register C++'s overloaded `std::sqrt(...)` function with the framework. Simply specifying `transform(..., std::sqrt)` will fail at compile time as the compiler will not be able to determine which overload is desired.

Instead, the code author can use the following[18]:

```
transform(..., [](double x){ return std::sqrt(x); }, ...);
```

where the desired overload is selected based on the `double` argument to the lambda expression.

3.5 Supported Higher-Order Functions

In Phlex, HOFs transform one family to another:

$$[b_i]_{i\in\mathcal{I}_{\text{output}}} = \text{HOF}(f_1,\ f_2,\ \dots)\ [a_i]_{i\in\mathcal{I}_{\text{input}}}$$

where the functions $f_1, f_2, \dots$ are *operators* required by the HOF. Note that the output index set $\mathcal{I}_{\text{output}}$ is not necessarily the same as the input index set $\mathcal{I}_{\text{input}}$.

In what follows, a family $[a_i]_{i\in\mathcal{I}_a}$ will often be represented by a single variable a. Whether a variable name (e.g. a) represents a family or an operation to a higher-order function will be apparent based on context.

Each HOF below also supports the *label* $: \mathbb{1} \to L$ operator, where the output data layer is user-specifiable. The *label* operator is explicitly mentioned only for those HOFs that require it–i.e. unfold and window[19].

3.5.1 Transforms

Transform	Operator	Output family length
$b = \text{transform}(f)\ a$	$f : A \to B$	$\lvert b\rvert = \lvert a\rvert$

The transform is the simplest HOF whose algorithms create data products. Specifically, the algorithm f is applied to each element of the input family a, creating a corresponding data product in the output family b:

$$b = [b_i]_{i\in\mathcal{I}_a} = [f\ a_i]_{i\in\mathcal{I}_a} = \text{transform}(f)\ [a_i]_{i\in\mathcal{I}_a}$$

where $b_i = f\ a_i$. Note that the index set of the output family is the same as the index set of the input family.

[18] Equivalently, one can use the obscure syntax `transform(..., static_cast<double(*)(double)>(std::sqrt), ...)`, where `std::sqrt` is cast to the desired overload.

[19] The specific rules by which the *label* operator can be used are given in the technical design *(under preparation)*.

3.5.1.1 Operator Signature

Operator	Allowed signature
f	`return_type function_name(P1, Pn..., Rm...) [quals];`

The `return_type` must model the created data-product type described in Section 3.3.2. An algorithm may also create multiple data products by returning a `std::tuple<T1, ..., Tn>` where each of the types `T1, ..., Tn` models a created data-product type.

3.5.1.2 Registration Interface

To illustrate the different ways a transform's algorithm can be registered with Phlex, we use the following classes and functions, which are presumably defined in some experiment libraries.

```
class geometry { ... };
class hits { ... };
class tracks { ... };
class vertices { ... };
class waveforms { ... };

hits find_hits(waveforms const&) { ... }
vertices make_vertices(geometry const&, tracks const&) { ... }

std::tuple<int, int> count_good_hits(hits const&) { ... }
// Return type: first number = number of good hits
//              second number = number of all hits
```

Transform with one argument (default output product name)

```
PHLEX_REGISTER_ALGORITHMS(config)
{
  transform("hit_finder", find_hits, concurrency::unlimited)
    .family("Waveforms"_in("APA"));
}
```

Transform with one argument (user-specified output product name)

As shown in Fig. 3.1 *and described in* Section 3.4

```
PHLEX_REGISTER_ALGORITHMS(config)
{
  products("GoodHits") =
    transform("hit_finder", find_hits, concurrency::unlimited)
    .family("Waveforms"_in("APA"));
}
```

Transform with two arguments (default output product name)

As shown in Fig. 3.1 *and described in* Section 3.4.1.1

```
PHLEX_REGISTER_ALGORITHMS(config)
{
  products("Vertices") =
    transform("vertex_maker", make_vertices, concurrency::unlimited)
```

(continues on next page)

(continued from previous page)

```
    .family("Geometry"_in("Job"), "GoodTracks"_in("APA"));
}
```

Transform creating two data products (user-specified output product names)

```
PHLEX_REGISTER_ALGORITHMS(config)
{
  products("NumGoodHits", "NumAllHits") =  // <= One name per tuple slot of return type
    transform("hit_counter", count_good_hits, concurrency::unlimited)
    .family("GoodHits"_in("APA"));
}
```

3.5.2 Observers

Observer	Operator	Output family length
$[\ ] = \text{observe}(f)\ a$	$f : A \to \mathbb{1}$	0

There are cases where a user may wish to inspect a data product without adjusting the data flow of the program. This is done by creating an algorithm called an *observer*, which may access a data product but create no data products. An example of this is writing ROOT histograms or trees that are not intended to be used in another framework program.

Note that, in a purely functional approach, it is unnecessary to invoke an observer as no data will be produced by it[20]. Phlex, however, supports observers as physicists rely on the ability to induce side effects to analyze physics data.

Unlike filters and predicates, observers (by definition) are allowed to be the most downstream algorithms of the graph.

3.5.2.1 Operator Signature

Operator	Allowed signature
f	`void function_name(P1, Pn..., Rm...) [quals];`

3.5.2.2 Registration Interface

The below shows how the *histogram_hits* operator in Fig. 3.1 would be registered in C++[21]. It uses the `resource<histogramming>` interface to provide access to a putative histogramming resource (see Section 3.9).

```
class hits { ... };
void histogram_hits(hits const&, TH1F&) { ... }

PHLEX_REGISTER_ALGORITHMS(m, config)
{
  auto h_resource = m.resource<histogramming>();

  observe(histogram_hits, concurrency::serial)
    .family("GoodHits"_in("APA"), h_resource->make<TH1F>(...));
}
```

[20] An observer is a special case of a filter that rejects all data presented to it.

[21] For this example, we ignore the need to filter `"GoodHits"` using the `high_energy` predicate. This is addressed in Section 3.5.4.

Note that the number of arguments presented to the `family(...)` call matches the number of input parameters of the registered algorithm `histogram_hits`. This indicates that each invocation of `histogram_hits` will be presented with one `"GoodHits"` data product and the `TH1F` resource.

3.5.3 Predicates

Predicate	Operator	Output family length
$\tilde{b} = \text{predicate}(f)\ a$	$f : A \to \mathbb{B}$	$\lvert\tilde{b}\rvert = \lvert a\rvert$

The predicate HOF is a transform (see Section 3.5.1) whose operator returns Boolean *true* or *false*. However, instead of the framework interpreting the Boolean result as a data product, the return value is used to short-circuit the processing of the data-flow graph for data products that do not meet the specified criteria. This short-circuiting behavior is known as *filtering* and is described in Section 3.5.4.

Note that the output family generated by the predicate is the same length as the input family but with values that are either Boolean *true* or *false*. It is not until the predicate results are used by the filter that an input family is potentially reduced in length.

> **Note**
>
> Phlex will schedule a predicate HOF for execution only if it is included in a predicate expression (see Section 3.5.4).

3.5.3.1 Operator Signature

Operator	Allowed signature
f	`bool function_name(P1, Pn..., Rm...) [quals];`

3.5.3.2 Registration Interface

The workflow in Fig. 3.1 demonstrates a use of a predicate in the *filter(high_energy)* node, where the predicate is *high_energy* that operates on each *GoodHits* data product. Although Fig. 3.1 does not include an explicit node for the *high_energy* predicate (for reasons of exposition), the predicate HOF *does* have its own node, which is then bound to one or more filters via predicate expressions. The registration for the predicate node in Fig. 3.1 would look like:

```
class hits { ... };
bool high_energy(hits const& hs) { ... }

PHLEX_REGISTER_ALGORITHMS(config)
{
  predicate("high_energy", high_energy, concurrency::unlimited)
    .family("GoodHits"_in("APA"));
}
```

3.5.4 Filtering

Filter	Operator	Output family length
$a' = \text{filter}(\phi)\ a$	$\phi : \mathbb{B}^n \to \mathbb{B}$	$\lvert a'\rvert \leq \lvert a\rvert$

As mentioned in Section 3.5.3, the execution of workflow graph can be short-circuited if data products do not meet specified criteria. This process, known as *filtering*, effectively shortens the input family a by retaining only those entries that satisfy a predicate ϕ, thus creating a different family a' composed of elements from a.

Filtering is of interest only when there is a downstream node that can receive the filtered family. Therefore, Phlex will not schedule a filter for execution if the only nodes downstream of it are other filters or predicates.

Filtering can be applied to the input data-product families of any HOFs without explicitly registering a filter HOF. This is done through a *predicate expression*, which is a stringized form of the predicate ϕ that is applied to the input data-product family, retaining only the elements that satisfy the predicate.

3.5.4.1 Predicate Expression

The predicate ϕ is Boolean expression whose input arguments correspond to the Boolean results of *explicitly registered* predicates. For example, in Fig. 3.1, the predicate in *filter(high_energy)* is *high_energy*, which is an explicitly registered predicate, as presented in Section 3.5.3.2.

It is possible for one filter named `"only_high_energy"` to use as its predicate *high_energy*, whereas another filter named `"not_high_energy"` could use the negation *filter*($\neg$*high_energy*). In this case, the predicate *high_energy* is executed only once, but its value can be used in different ways in the predicate expression.

A predicate expression can be evaluated on a higher-level data cell than the data-product family in question. For example, suppose none of the *GoodHits* data products in a given *Spill* were suitable for processing. It is possible to create a filter that would reject all *GoodHits* data products from that *Spill* even though the predicate itself interrogated only the *Spill* information and not the lower-level good-hits information from the *APA*.

The supported grammar of the predicate expression is discussed in the technical design *(under preparation)*.

3.5.4.2 Registration interface

The following example shows the complete registration for histogramming the filtered *GoodHits* data products shown in Fig. 3.1.

```
class hits { ... };
void histogram_hits(hits const&, TH1F&) { ... }

PHLEX_REGISTER_ALGORITHMS(m, config)
{
  auto h_resource = m.resource<histogramming>();

  observe(histogram_hits, concurrency::serial)
    .family("GoodHits"_in("APA"), h_resource->make<TH1F>(...))
    .when("high_energy");  // <= predicate expression within the when(...) call
}
```

In practice, it is convenient to specify a predicate expression as part of a Phlex program's run-time configuration instead of hard-coded into a compiled library. Phlex allows users to specify predicate expressions in a program configuration:

```
{
  # Speculative configurations for two observer nodes that use the same module library
  histogram_high_energy_hits: {
    plugin: "histogram_hits.so",
  },
  histogram_low_energy_hits: {
    plugin: "histogram_hits.so",
    when: "!high_energy"  # Negate filter, overriding compiled when(...) clause
```

(continues on next page)

(continued from previous page)

```
  },
}
```

Note that specifying a predicate expression via the `when` configuration parameter overrides whatever predicate expression may have been hard-coded into the compiled module.

3.5.5 Partitioned Folds

Partitioned fold	Operators	Output family length
$d = \text{fold}(f,\ \textit{init},\ \textit{part})\ c$	$f : D \times C \to D$ $\textit{init} : \text{Opt}(\mathcal{I}_d) \to D$ $\textit{part} : \{\mathcal{I}_c\} \to \mathbb{P}(\mathcal{I}_c)$	$\lvert d\rvert \leq \lvert c\rvert$

As mentioned in Section 2.5, a *fold* can be defined as a transformation of a family of data to a single value:

$$d = \text{fold}(f,\ \textit{init})\ [c_i]_{i \in \mathcal{I}_c}$$

where the user-defined operation f is applied repeatedly between an accumulated value (initialized by $init$) and each element of the input family.

In a framework context, however, multiple fold results are often desired in the same program for the same kind of computation. Consider the workflow in Fig. 3.1, which processes *Spill*s, identified by the index j or, more specifically, the tuple $(S\ j)$. Each *Spill* is unfolded into a family of *APA*s, which are identified by the pair of indices jk or, more specifically, the tuple $(S\ j, A\ k)$. The energies of the *GoodHits* data products in Fig. 3.1 are summed across *APA*s per *Spill* using the *fold(sum_energy)* node.

Instead of creating one fold result, we thus use a *partitioned fold* to create one summed energy data-product per *Spill*:

$$\begin{aligned}&[E_{(S\ 1)},\ \ldots,\ E_{(S\ n)}]\\ &\qquad = \text{fold}(\textit{sum\_energy},\ \textit{init},\ \textit{into\_spills})\ [hs_{(S\ 1,\ A\ 1)},\ hs_{(S\ 1,\ A\ 2)},\ \ldots,\ hs_{(S\ n,\ A\ 1)},\ hs_{(S\ n,\ A\ 2)},\ \ldots]\end{aligned}$$

where $E_{(S\ j)}$ denotes the *TotalHitEnergy* data product for *Spill* j, and $hs_{(S\ j,\ A\ k)}$ is the *GoodHits* data product for *APA* k in *Spill* j.

The above equation can be expressed more succinctly as:

$$[E_j]_{j \in \mathcal{I}_{\text{out}}} = \text{fold}(\textit{sum\_energy},\ \textit{init},\ \textit{into\_runs})\ [hs_i]_{i \in \mathcal{I}_{\text{in}}}$$

where

$$\begin{aligned}\mathcal{I}_{\text{in}} &= \{(S\ 1,\ A\ 1),\ (S\ 1,\ A\ 2),\ \ldots,\ (S\ n,\ A\ 1),\ (S\ n,\ A\ 2), \ldots\}, \text{and}\\ \mathcal{I}_{\text{out}} &= \{(S\ 1),\ \ldots, (S\ n)\}\ .\end{aligned}$$

3.5.5.1 Partitions

Factorizing a set of data into non-overlapping subsets that collectively span the entire set is called creating a set *partition* [Wiki-Partition]. Each subset of the partition is called a *cell*[22]. In the above example, the role of the *into_spills* operation is to partition the input family into *Spill*s so that there is one fold result per *Spill*. In general, however, the partitioning function is of the form $\textit{part} : \{\mathcal{I}_c\} \to \mathbb{P}(\mathcal{I}_c)$, where:

- the domain is the singleton set that contains only the index set $\mathcal{I}_c$ (i.e. *part* can only be invoked on $\mathcal{I}_c$), and
- the codomain is the set of partitions of $\mathcal{I}_c$ or $\mathbb{P}(\mathcal{I}_c)$; note that the output index set $\mathcal{I}_d \in \mathbb{P}(\mathcal{I}_c)$.

[22] The term *data cell* used elsewhere in this document is intended to closely reflect the concept of the partition cell.

The function *part* also establishes an equivalence relationship on the index set $\mathcal{I}_c$, where each element of the index set is mapped to a cell of the partition. The number of elements in the output family d corresponds to the number of partition cells.

As of this writing, the only partitions supported are those that correspond to the names of data layers. The partition *into_spills* can thus be represented by the string `"Spill"`, which denotes that there is one partition spell per *Spill*.

3.5.5.2 Initializing the Accumulator

A crucial ingredient of the fold is the *accumulator*, which stores the fold result while it is being formed. Each accumulator is initialized by invoking a user-defined operation *init* : Opt($\mathcal{I}_d$) $\rightarrow D$, which returns an object that has the same type D as the fold result[23]. The Opt($\mathcal{I}_d$) domain means that:

1. *init* can receive an argument corresponding to the identifier of a cell, which is a member of the output index set $\mathcal{I}_d$. In the example above, the relevant identifier would be that of the *Spill*–i.e. $(S\ j)$.
2. *init* can be invoked with no arguments, thus producing the same value each time the accumulator is initialized. This is equivalent to initializing the accumulator with a constant value.

The implementation of *init* for the total good-hits energy fold results is to return the constant 0.

3.5.5.3 Fold Operation

A cell's fold result is obtained by repeatedly applying a fold operation to the cell's accumulator and each element of that cell's input family. The fold operation has the signature $f : D \times C \rightarrow D$, where D represents the type of the accumulator/fold result, and C is the type of each element of the input family.

In the above example, the function *sum_energy* receives a floating-point number $E_{(S\ i)}$, representing the accumulated good-hits energy for *Spill* j and "combines" it with the good-hits object $hs_{(S\ j,\ A\ k)}$ that belongs to *APA* k in spill j. This combination involves calculating the energy represented by the *GoodHits* data product $hs_{(S\ j,\ A\ k)}$ and adding that to the accumulated value. This "combined" value is then returned by *sum_energy* as the updated value of the accumulator[24]. The function *sum_energy* is repeatedly invoked to update the accumulator with the *GoodHits* data product. Once all *GoodHits* data products in *Spill* j have been processed by *sum_energy*, the accumulator's value becomes the fold result for that *Spill*.

3.5.5.4 Operator Signatures

Operator	Allowed signature
f	`void function_name(result_type&, P1, Pn..., Rm...) [quals];`
init	*as constant:* `result_type{...}`
	as function: `result_type function_name() [quals];`
	as function: `result_type function_name( <cell identifier> ) [quals];`
part	*Name of data layer for output data product*

The fold's `result_type` must model the created data-product type described in Section 3.3.2. A fold algorithm may also create multiple data products by using a `result_type` of `std::tuple<T1, ..., Tn>` where each of the types `T1, ..., Tn` models a created data-product type.

[23] It is acceptable for *init* to return a type that is convertible to the accumulator's type.

[24] Returning an updated accumulated value is generally not the most memory-efficient approach as it requires at least two copies of an accumulated value to be in memory at one time. The approach adopted by Phlex is to include a reference to the accumulated value as part of the fold operator's signature. The accumulator can then be updated in place, thus avoiding the extra copies of the data.

3.5.5.5 Registration Interface

The *fold(sum_energies)* node in Fig. 3.1 would be represented in C++ as:

```
void sum_energy(double& total_hit_energy, hits const& hs) { ... }

PHLEX_REGISTER_ALGORITHMS(config)
{
  products("TotalHitEnergy") =
    fold(
      "sum_hit_energy",       // <= Node name for framework
      sum_energy,             // <= Fold operation
      0.,                     // <= Initializer for each fold result
      "Spill",                // <= Partition level (one fold result per Spill)
      concurrency::unlimited  // <= Allowed concurrency
    )
    .family("GoodHits"_in("APA"));
}
```

In order for the user-defined algorithm `sum_energy` algorithm to be safely executed concurrently, protections must be in place to avoid data races when updating the `total_hit_energy` result object from multiple threads. Possible solutions include using `std::atomic_ref<double>`[25], placing a lock around the operation that updates `total_hit_energy` (less desirable due to inefficiencies), or perhaps using `std::atomic<double>`[26] instead of `double` to represent the data product.

3.5.6 Partitioned Unfolds

Partitioned unfold	Operators	Output family length
$c = \text{unfold}(p,\ gen,\ label)\ d$	$p : N \to \mathbb{B}$	$\lvert c\rvert \geq \lvert d\rvert$
	$gen : N \to N \times C$	
	$label : \mathbb{1} \to L$	

As discussed in Section 2.5, the opposite of a fold is an *unfold*, where a family of objects is generated from a single object. The example given in Section 2.6.1 is iota, which generates a sequence of contiguous integers given one input number:

$$c = [1,\ 2,\ 3,\ \ldots,\ n] = \text{iota}\ n = \text{unfold}(greater\_than\_zero,\ decrement)\ n$$

where iota has been expressed in terms of an unfold HOF that receives the predicate $greater\_than\_zero$ and a generator called $decrement$.

The unfold operation is repeatedly called until the predicate returns *false*, whereby it emits an empty list $[\,]$:

$$\begin{aligned} c &= \text{unfold}(greater\_than\_zero,\ decrement)\ n \\ &= \text{unfold}(greater\_than\_zero,\ decrement)\ n-1\ +\ [n] \\ &\ \ \vdots \\ &= \text{unfold}(greater\_than\_zero,\ decrement)\ 0\ +\ [1, 2, \ldots, n-1, n] \\ &= [\,]\ +\ [1, 2, \ldots, n-1, n] \end{aligned}$$

where $+$ in this example denotes an operator that concatenates two lists.

Heuristically, this can be thought of as executing the function:

[25] https://en.cppreference.com/w/cpp/atomic/atomic_ref.html
[26] https://en.cppreference.com/w/cpp/atomic/atomic.html

```
def unfold(predicate, generator, n):
    result = []
    next_value = n
    while predicate(next_value):
        # generator returns a new value for next_value
        next_value, family_element = generator(next_value)
        result.prepend(family_element)
    return result
```

where the user supplies the `predicate` (p) and `generator` (*gen*) algorithms.

Phlex expands the concept of an unfold by allowing it to operate on a family of data products corresponding to a set partition [DUNE 33]. This *partitioned unfold* is shown in Fig. 3.1, where the *unfold(into_apas)* node transforms a flat family of *SimDepos* data products (each of which belong to a cell within the *Spill* partition) into a family of families, with each nested family containing the *Waveforms* data products for all *APA*s within a given *Spill*.

Unfolding in this way can be used for parallelizing the processing of a data product in smaller chunks. Breaking up the processing of a data product can also be an important ingredient in controlling the memory use of a Phlex program.

> **Note**
>
> Phlex requires the use of the *label* operator in unfolds to avoid collisions with already-existing data products and to reflect the more granular data-processing that occurs as a result of the unfold.

3.5.6.1 Next Type

The signatures for the operators p and *gen* have the curious type N, which seems unrelated to the input family d, whose elements are of type D, or the output family c, whose elements are of type C. The type N refers to the type of the *next* value on which the unfold operates. In the iota example above, the type N is the same as the input argument n, which is an integer, and it is the same as that of the output family elements, which are also integers.

The unfold in Fig. 3.1, however, demonstrates an example where N is equal to neither D nor C. Whereas the input type D corresponds to the *SimDepos* data product in each *Spill*, the output type C represents the *Waveforms* data products produced for each *APA*. Assuming *SimDepos* is represented as a `std::vector<SimDepo>` object, a reasonable type for N might be `std::vector<SimDepo>::const_iterator`, thus permitting the comparison of iterators in the predicate p and using it in the generator *gen* for processing portions of the initial data product. The generator would thus return a pair with an advanced iterator and a *Waveforms* object corresponding to one *APA*.

The choice of the next type N thus depends on the use case and is not prescribed by Phlex.

3.5.6.2 Operator Signatures

Operator	Allowed signature
p	`bool function_name(next_type) [quals];`
gen	`std::pair<next_type, product_type> function_name(next_type, Rm...) [quals];`
label	*Name of data layer of output data products*

The unfold's `product_type` must model the created data-product type described in Section 3.3.2. An unfold's *gen* algorithm may also create multiple data products by returning an object of type `std::tuple<next_type, T1, ..., Tn>`, where each of the types `T1, ..., Tn` models a created data-product type.

3.5.6.3 Registration Interface

As unfolds require coordination between the predicate p and the generator *gen*, they are supported by implementing classes with member functions that are registered with the framework.

For the *unfold(to_apas)* node in Fig. 3.1, the C++ code for the *experiment* algorithm would be:

```
class sim_depos { ... };
class waveforms { ... };

class to_apas {
  using next_type = sim_depos::const_iterator;
  next_type advance(next_type) { ... }
  next_type end_;

public:
  explicit to_apas(sim_depos const& sds)  // Constructed with input data-product
    : end_{sds.end()}
  {}

  bool keep_going(next_type next) const { return next != end_; }

  std::pair<next_type, waveforms> make_waveforms(next_type next) const
  {
    // Create waveforms object 'ws' using 'next',
    // ... and then move into result
    return std::make_pair(advance(next), std::move(ws));
  }
};
```

The definition of `advance(...)` would advance the next iterator according to some desired chunk size, or it would return an end iterator when all elements of the `"SimDepos"` data product have been processed. The class is then registered with Phlex via:

```
PHLEX_REGISTER_ALGORITHMS(config)
{
  products("Waveforms") =
    unfold<to_apas>(
      "to_apas",                 // <= Node name for framework
      &to_apas::keep_going,      // <= Unfold predicate
      &to_apas::make_waveforms,  // <= Unfold generator
      "APA",                     // <= Data layer for output data products
      concurrency::unlimited     // <= Allowed concurrency
    )
    .family("SimDepos"_in("Spill"));
}
```

Note that the template argument in `unfold<to_apas>` is an indication that the framework will create an object of type `to_apas` each time it receives a `"SimDepos"` data product. The framework ensures that all data products remain in memory for as long as they are required, and once they are no longer needed, they (as well as any unneeded `to_apas` objects) are evicted from memory as soon as possible [DUNE 142].

3.5.7 Windows

Window	Operators	Output family length
$y = \text{window}(f,\ adj,\ label)\ x$	$f : X \times \text{Opt}(X) \to Y$ $adj : \mathcal{I}_x \times \mathcal{I}_x \to \mathbb{B}$ $label : \mathbb{1} \to L$	$\lvert y\rvert = \lvert x\rvert$

One of the unique capabilities of Phlex is to execute an algorithm on data products that belong to adjacent data cells (see Section 3.4.1.2). The workflow in Fig. 3.1 shows a such a node *window(make_tracks)*, which is presented with pairs of *GoodHits* data products, with each data product in the pair belonging to adjacent *APA*s. It is the user-provided adj function which determines whether two data cells are adjacent.

For simplicity, imagine that each *APA* identifier (i.e. member of the set $\mathcal{I}_{\text{APA}}$) can be represented as an integer. A straightforward adj implementation might be to group the *GoodHits* data products from *APA*s with consecutive numbers:

$$[hs_i]_{i\in\mathcal{I}_{\text{APA}}} = [\overbrace{hs_1}^{a}, \underbrace{\overbrace{hs_2}^{b},\ \overbrace{hs_3}^{c}}_{b},\ hs_4,\ \ldots,\ \underbrace{hs_{n-1},\ hs_n}_{m}\ ]$$

The data products corresponding to windows a through m are grouped into pairs and presented to an algorithm $make\_tracks'$, which has the signature Hits $\times$ Hits $\to$ Tracks. There are, at most, $n-1$ unique pairs that can be presented to the function $make\_tracks'$ such that:

$$[ts_i]_{i\in\mathcal{I}'_{\text{APA}}} = [make\_tracks'(hs_i, hs_{i+1})]_{i\in\mathcal{I}'_{\text{APA}}} = \text{window}(make\_tracks',\ adj,\ \text{APA})\,[hs_i]_{i\in\mathcal{I}_{\text{APA}}}$$

where the index set $\mathcal{I}'_{\text{APA}}$ is $\mathcal{I}_{\text{APA}}$ without the last identifier n[27]. In this example, the identifier of the first hs object in the pair is used to identify the tracks collection ts. But Phlex does not mandate this choice, and a different data layer could be specified by the $label$ operator for the data products of the output family.

3.5.7.1 Operator Signatures

One limitation of the above formulation is that index sets of the input and output families are not the same. To address this infelicity, the function signature of $make\_tracks'$ can be adjusted such that the second argument receives an optional type. We call this new algorithm $make\_tracks$:

$$make\_tracks : \text{Hits} \times \text{Opt(Hits)} \to \text{Tracks}$$

thus permitting symmetry between the input and output data-product families:

$$\begin{aligned}[ts_i]_{i\in\mathcal{I}_{\text{APA}}} &= \text{window}(make\_tracks,\ adj,\ label)\,[hs_i]_{i\in\mathcal{I}_{\text{APA}}}\\ &= [make\_tracks(hs_i, hs_{i+1})]_{i\in\mathcal{I}'_{\text{APA}}} + [make\_tracks(hs_n, ())]\end{aligned}$$

where $label$ returns the value of *APA*, $+$ is the list-concatenation operator, and $()$ is the null value. Phlex supports the function signature whose second argument is an optional type $\text{Opt}(X)$.

Operator	Allowed signature
f	`return_type function_name(P1, Opt<P2>, Rm...) [quals];`
adj	`bool function_name(<P1 identifier>, <P2 identifier>) [quals];`
$label$	*Name of data layer of output data products*

The `return_type` must model the created data-product type described in Section 3.3.2. The algorithm f may also create multiple data products by returning a `std::tuple<T1, ..., Tn>` where each of the types `T1, ..., Tn` models a created data-product type.

[27] The expression $f(hs_n, hs_{n+1})$ is ill-formed as there are only n elements in the set $\mathcal{I}_{\text{APA}}$.

The second argument `Opt<P2>` indicates that an optional type is passed to the algorithm. It is permitted to use resources (i.e. `Rm...`) in the function f. The data cell identifers of `P1` and `P2` are used to determine whether two data-products reside in adjacent data cells.

3.5.7.2 Registration Interface

The *window(make_tracks)* node in Fig. 3.1 would be represented in C++ as:

```
class hits { ... };
class tracks { ... };
class id { ... };
tracks make_tracks(tracks const& ts, tracks const* next_ts) { ... }
bool are_adjacent(id const& left, id const& right) { ... }

PHLEX_REGISTER_ALGORITHMS(config)
{
  products("GoodTracks") =
    window(
      "track_maker",          // <= Node name for framework
      make_tracks,            // <= Window algorithm (f)
      are_adjacent            // <= Adjacency criterion
      "APA",                  // <= Output data layer
      concurrency::unlimited  // <= Allowed concurrency
    )
    .family("GoodHits"_in("APA"));
}
```

Note that the second input parameter for `make_tracks` is an optional type. The type `id` is a metadata type (possibly defined by the experiment) that enables the comparison of data-product identifiers for establishing adjacency.

3.6 Framework Driver

In imperative programming languages, control flow structures (such as loops) are provided by the language. Users of the language are not able to create new control flow facilities. Previous frameworks contained framework-defined *event loops* that provided the control flow for a framework program. Users of these frameworks were not able to create alternate control flow structures.

In the functional programming paradigm, control flow is provided by higher-order functions. Users of such languages are able to create new control flow structures by writing new higher-order functions. In Phlex, a framework *driver* is a higher-order function that provides the control flow for a framework program. Phlex provides a variety of framework drivers and also allows users to register their own drivers with the framework, to support different processing needs, e.g. [DUNE 21] and [DUNE 40].

The workflow shown in Fig. 3.1 is driven by a *driver(Spill)* node that is configured to process all the *Spill*s in the specified HDF5 input files. The *driver(Spill)* communicates with the IO system to determine what *Spill*s are available for processing. All driver algorithms know about the single *Job*-level data cell that corresponds to the entirety of the framework program execution (the job). The driver emits the single *Job* object, and a family of *Spill* objects, which cause the *provide* algorithms to create their products, thus starting the data flow through the graph.

Other driver algorithms support different processing workflows. In one example workflow, a single *Spill* could be reconstructed multiple times with different calibration objects used in the reconstruction, thus allowing a calibration study for a single spill to be performed in one framework execution. In another example a different driver algorithm could be used to drive the framework to process the Cartesian product of a set of *Spill*s and a set of *Calibration* data cells, to perform a different calibration study.

3.7 Data-Product Providers

Providers communicate with the IO system through a specified API that is implemented by each IO back end. The IO API provides the ability to read (and write) data to the IO system. Providers communicate with the IO system only through this API. This allows the framework to support multiple IO back ends, including ROOT [DUNE 74] and HDF5 [DUNE 141], and ensure that new back ends can be added without modifying the framework code [DUNE 73].

Providers are responsible for being able to read data written by earlier code versions, subject to policy decisions made by the experiment [DUNE 76].

Providers are also responsible for be able to read some types of data (such as calibration data, or geometry descriptions) from sources other than files written by the IO system. The workflow shown in Fig. 3.1 shows an example of two such *provide* algorithms. The first one is responsible for reading data *Spill*s from the IO system, and the second one is responsible for reading the data from GDML files.

3.8 Data-Product Writers

Writers write both (user-defined) data products and (framework- and user-defined) metadata to persistent output [DUNE 17]. Files written to by writers can be read using providers, through the same IO back end [DUNE 19]. The metadata written by writers is sufficient to allow the reproduction of the processing that created the written data [DUNE 28]. This means that, for example, the full configuration of the framework executable is included in the written metdata.

Writers use the same IO API as providers, ensuring that the only communication with the IO system is through the IO API. This allows the framework to support multiple IO back ends, including ROOT [DUNE 74] and HDF5 [DUNE 141], and ensure that new back ends can be added without modifying the framework code [DUNE 73]. The workflow shown in Fig. 3.1 shows an example with the *write* algorithm writing data products to a ROOT back end.

3.9 Resources

As mentioned in Section 3.3, the typical form of a C++ function that can be registered with the framework is:

```
return_type function_name(P1, Pn..., Rm...) [quals];
```

where the types `P1, Pn...` represent data products, and the types `Rm...` represent resources.

> **Note**
>
> Phlex calls *resources* the objects that can be used by an algorithm to notify the framework that the algorithm requires access to a shared entity that is not semantically related to the data-layer hierarchy. Note that in the requirements that the word "resource" is sometimes used in a more general sense. In this section we are only referring to the specific kind of resource described above.

An example registration of an algorithm that requires both a data product and a resource is found in Section 3.5.2. The details of the registration code express to the framework which arguments correspond to data products and which correspond to resources [DUNE 52]. They may be stateless objects (e.g. a resource that denotes that an algorithm requires the use of a specific thread-unsafe library) or stateful objects (e.g. a resource that denotes access to a GPU, when the platform on which the framework program is running contains several GPUs). Neither of these examples contain mutable state. Resources (unlike data products) may have mutable state accessible to the algorithm (e.g. a histogram instance that could be shared across multiple algorithms). For resources that are thread-unsafe, the framework ensures that two algorithms are not interacting with the resource at the same time. The framework is responsible for efficiently scheduling algorithms based, in part, upon the availability of resources [DUNE 50]. To facilitate efficient scheduling of work, the resources needed by the program are specified via configuration or in the algorithm-registration code [DUNE

47][28].

Examples of resources include:

- GPUs
- Network connections
- Thread-unsafe utilities
- Inference servers
- Database handles [DUNE 35], [DUNE 40]

Whereas data products have provenance associated with them, resources do not.

3.9.1 Limited Resources

Some resources are used to indicate that an algorithm requires sole use of some program entity. One example of such an entity is a thread-unsafe library, where the framework must ensure that only one algorithm is interacting with that library at any time [DUNE 45], [DUNE 145]. A second example of a limited resource is a fixed number of GPUs present on a particular platform, where Phlex must ensure that each algorithm requiring the use of a GPU has sole access to the GPU it is running on for the duration of the algorithm's execution. A third example of a limited resource could be an algorithm's declaration that it requires spawning some number of threads for its execution (rather than using the framework's task-based execution model). Such an algorithm could declare the need for the reservation of some number of threads by requiing that number of thread resources [DUNE 152]. The framework would then ensure that only as many threads as the configuration has provided can be used by algoirthms at any one time. These resources are called *limited resources*, because the framework is responsible for limiting the access to the resource to one algorithm at a time.

An algorithm to be used by Phlex indicates that it requires a limited resource by requiring an argument that denotes such a resource.

3.9.2 GPUs

In order to allow algorithms to make use of GPUs, and to allow the composition of workflows that involve both CPU-based and GPU-based algorithms, Phlex provides a mechanism for an algorithm that requires access to a GPU to declare that fact [DUNE 54]. This is done by making the algorithm accept a resource that denotes the GPU. Phlex can support running on platforms that provide access to more than one GPU while ensuring that a given algorithm has sole access to the GPU it requires while it is executing. Phlex also provides, through the same mechanism, the ability for an algorithm to specify that it requires remote access to a GPU.

3.9.3 Random Number Resource

The generation of pseudorandom numbers (hereafter just *random numbers*) is a critical aspect of obtaining physics results, especially when simulating data. Although multiple random-number generation techniques exists, the vast majority of random-number implementations used in HEP rely on stateful random-number engines, which are subject to data races in multi-threaded contexts. To ensure reproducible data and to ensure thread-safe access to stateful random-number engines, HEP frameworks impose idiosyncratic constraints on their use.

DUNE has similar requirements on reproducibility of random numbers in a concurrent context [DUNE 36]. However, instead of working around the limitations of stateful random-number engines, Phlex supports a random-number generation technique specifically designed to reproduce random-numbers in a concurrent program. Counter-based random number generators (CBRNGs) [Wiki-CBRNG] provide such capabilities, and Phlex will provide an interface algorithm authors can use to take advantage of them.

[28] Details for specifying program resources are described in the technical design *(under preparation)*.

3.9.4 User-defined Resources

While Phlex will provide some commonly-used types to represent resources, it will also be possible for users to create new types to represent a new type of resouce, with no modifications to the Phlex framework code [DUNE 149]. Such resource types have no dependency on Phlex, so that a user algorithm employing such a resource does not thereby incur any dependency on the framework.

3.10 Program Configuration

As described in Section 3.1, a Phlex program starts by ingesting a configuration, which informs how the data-flow graph (e.g. see Fig. 3.1) should be established.

To support reproducibility of data, a Phlex program is configured by *only one* configuration document [DUNE 69], although that document may be compiled from multiple distinct sources [DUNE 133] (such as included or imported files). These documents may be written in the following languages: Jsonnet, YAML, JSON (as a strict subset of Jsonnet), or FHiCL [DUNE 72]. A given configuration can be written in any of these languages, but no configuration document may be written in more than one language.

The other details and rules by which a Phlex program is configured largely do not inform the conceptual design and are described in the technical design *(under preparation)*.

APPENDIX

A

DEFINITIONS

Algorithm
A user-defined function registered for execution by the framework.

Algorithms often serve as operators in a higher-order function.

APA
Anode **P**lane **A**ssembly, a physical far-detector unit comprising thousands of wires as a planar slice.

Backward compatibility (persisted data)
The ability to construct the in-memory representation of data from persisted information derived from an earlier in-memory representation of those data.

CHOF
See Configured higher-order function

Configured higher-order function
An entity created when registering an algorithm with the framework.

The registration includes the algorithm's input/output data product requirements, the algorithm's resource requirements, and the higher-order function to which the algorithm serves as an operator.

Data cell
A grouping of data products that is identifiable by the framework[29].

Each data cell has has a data layer label, which indicates the type of the data cell. All the data cells of a given data layer are of the same type. All the data cells of a given data layer are identifiable by the same type of index, and each is identified by a unique index value. In *art*, individual *Run* objects, *Subrun* objects and *Event* objects are examples of data cells.

Data layer
A group of data cells each of which has the same data layer label.

A data layer is one node in a data-layer hierarchy. The top layer of the graph is always the *Job*. All other layers are defined by the combination of the configuration of the framework job and the layers defined in the input data. By contrast, in *art* the data layers are *Run*, *Subrun*, and *Event*.

Data product
An object managed by the framework and for which provenance information is recorded.

A data product is produced by an algorithm (or is recovered from storage by a provider) and can be passed as an input to other algorithms. Data products determine the flow of execution of the graph of CHOFs configured in a framework program.

Data-layer hierarchy
A hierarchy of data layers.

[29] In earlier documentation, the term *data product set* was used for this concept.

A data-layer hierarchy is an acyclic graph of relationships of logical containment. The top layer of the hierarchy is always the *Job*. All other layers in the hierarchy are defined by the combination of the configuration of the framework job and the layers defined in the input data. In *art*, the data-layer hierarchy is *Run-Subrun-Event*.

Framework ecosystem
The ensemble of software delivered with the application framework.

Index set
A mathematical set that provides the indexes of an indexed family.

For the family $a = [a_1, a_2, \ldots, a_n] = [a_i]_{i \in \mathcal{I}}$, the index set $\mathcal{I}$ is the set $\{1,\ 2,\ \ldots,\ n\}$.

Indexed family
A collection of elements identified by an index from an index set.

The term family is often used as a shorthand for indexed family. In *art* there was no direct representation of families. However, the sequence of *Events* processed by a given job is an example of a family of data cells. In a single *art* module, the sequence of event data products looked up by the module across all events is an example of a family of data products.

Metadata
Data that is ancillary to physics data.

Module
A compiled library that is dynamically loadable by the framework.

Provenance
A description of how the data were produced.

Examples include product parentage, job configuration, and library versions.

Reproducible
Identical inputs produce identical results.

Resource
A non-data software or hardware component managed by the program that can be used by algorithms.

Examples include:

- CPU cores
- CPU memory
- GPUs
- Network
- Thread-unsafe utilities
- Inference servers
- Databases

APPENDIX

B

FRAMEWORK REQUIREMENTS

Requirements norm: Baseline 1 (created March 03, 2025)

Important

All stakeholder requirements approved by DUNE are listed here for convenience. However, the requirements recorded in the DUNE framework Jama Connect project are authoritative and take precedence over any unintentional variations below.

B.1 Conceptual Requirements

Requirement: **Algorithm Decomposability** *DUNE 1*

status: Approved
tags: General, Original
jama: DUNE-DUNE_STKH-1*;*
notes: This is ID #01 from the original DUNE document.

The framework shall allow the execution of multiple algorithms.

- See Chapter 3

Requirement: **Data Product Representation** *DUNE 2*

status: Approved
tags: Original, General, Accelerators
jama: DUNE-DUNE_STKH-2*;*
notes: This is ID #02 from the original DUNE document.

The framework shall separate the persistent representation of data products from their in-memory representations as seen by algorithms.

- See Section 3.2.1.1

Requirement: **Full utilization of DUNE computing resources** *DUNE 8*

status: Approved
tags: General, Original, Reproducibility
jama: DUNE-DUNE_STKH-8*;*
notes: This is ID #05 from the original DUNE document.

The framework shall run on widely-used scientific computing systems in order to fully utilize DUNE computing resources.

- See Section 1.2.2

Requirement: **Algorithm hardware requirements** *DUNE 9*

status: Approved
tags: General, Original, Reproducibility
jama: DUNE-DUNE_STKH-9*;*
notes: This is ID #06 from the original DUNE document.

The framework shall provide an API that allows users to express hardware requirements of the algorithms.

- See Section 3.4

Requirement: **Algorithms can use a GPU** *DUNE 11*

status: Approved
tags: General, Accelerators, Reproducibility
jama: DUNE-DUNE_STKH-11*;*

The framework shall support running algorithms that require a GPU.

- See Section 1.2.2

Requirement: **Support for multiple programming languages** *DUNE 14*

status: Approved
tags: Original, General
jama: DUNE-DUNE_STKH-14*;*
notes: This is ID #07 from the original DUNE document. If DUNE decides that additional languages should be supported in the future, a specific requirement can be added for that language as a sub-requirement.

The framework shall support the invocation of algorithms written in multiple programming languages.

- See Section 1.3

Requirement: **Persist user-defined metadata** *DUNE 17*

status: Approved
tags: General, Original
jama: DUNE-DUNE_STKH-17*;*
notes: This is ID #08 from the original DUNE document.

The framework shall provide user-accessible persistence of user-defined metadata.

- See Section 3.8

Requirement: **Framework shall read its own output files** *DUNE 19*

status: Approved
tags: Physics Analysis, Original
jama: DUNE-DUNE_STKH-19*;*
notes: This is ID #10 from the original DUNE document.

The framework shall provide the ability to read a framework-produced file as input to a subsequent framework job so that the physics data are equivalent to the physics data obtained from a single execution of the combined job.

- See Section 3.8

Requirement: **Presenting data to subsequent algorithms** *DUNE 20*

status: Approved
tags: Physics Analysis
jama: DUNE-DUNE_STKH-20*;*

The framework shall present data produced by an already executed algorithm to each subsequent, requesting algorithm.

- See Section 3.1, Section 3.2.1

Requirement: **Mix input streams** *DUNE 21*

status: Approved
tags: Physics Analysis, Original
jama: DUNE-DUNE_STKH-21*;*
notes: This is ID #11 from the original DUNE document. This document uses "data cells" rather than "data sets".

The framework shall support the creation of data sets composed of data products derived from data originating from disparate input sources.

- See Section 3.6.

Requirement: **Flexible data units** *DUNE 22*

status: Approved
tags: Original, Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-22*;*
notes: This is ID #12 from the original DUNE document.

The framework shall support flexibly defined, context-aware processing units to address the varying granularity necessary for processing different kinds of data.

- See Section 1.1, Section 1.2.1

Requirement: **Process collections of unconstrained size** *DUNE 25*

status: Approved
tags: Original, Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-25*;*
notes: This originates from ID #16 from the original DUNE document.

The framework shall support processing of collections that are too large to fit into memory at one time.

- See Section 3.2.2

Requirement: **Framework recording of metadata for reproduction of output data** *DUNE 28*

status: Approved
tags: Original, Reproducibility, Provenance
jama: DUNE-DUNE_STKH-28*;*
notes: This is ID #18 from the original DUNE document.

The framework shall record metadata to output enabling the reproduction of the processing steps used to produce the data recorded in that output.

- See Section 3.8

Requirement: **Unfolding data products** *DUNE 33*

status: Approved
tags: Memory management, Original, Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-33*;*
notes: This is ID #58 from the original DUNE document.

The framework shall allow the unfolding of data products into a sequence of finer-grained data products.

- See Section 3.5.6

Requirement: **Access to external data sources** *DUNE 35*

status: Approved
tags: Original, Services
jama: DUNE-DUNE_STKH-35*;*
notes: This is ID #47 from the original DUNE document. By "external data sources," we mean "data sources **other than** framework-readable data files containing detector readout or simulated physics data."

The framework shall support access to external data sources.

- See Section 3.4.1.1, Section 3.9

Requirement: **Reproducibility with pseudo-random numbers** *DUNE 36*

status: Approved
tags: Original, Reproducibility, Provenance
jama: DUNE-DUNE_STKH-36*;*
notes: This is ID #22 from the original DUNE document.

The framework shall provide a facility to produce random numbers enabling algorithms to create reproducible data in concurrent contexts.

- See Section 3.1, Section 3.2.1, Section 3.4

Requirement: **Calibration database algorithms** *DUNE 40*

status: Approved
tags: Services
jama: DUNE-DUNE_STKH-40*;*
notes: This is ID #68 as proposed to DUNE.

The framework shall support algorithms that provide data from calibration databases.

- See Section 3.2, Section 3.6, Section 3.9.

Requirement: **Algorithms independent of framework interface** *DUNE 43*

status: Approved
tags: Services, Original
jama: DUNE-DUNE_STKH-43*;*
notes: This is ID #48 from the original DUNE document.

The framework shall support the registration of algorithms that are independent of framework interface.

- See Section 1.2.3, Section 1.4, Section 3.2.2, Section 3.3

Requirement: **Safely executing thread-safe and non-thread-safe algorithms** *DUNE 45*

status: Approved
tags: Original, Concurrency and multithreading
jama: DUNE-DUNE_STKH-45*;*
notes: This is ID #26 from the original DUNE document.

The framework shall safely execute user algorithms declared to be non-thread-safe along with those declared to be thread-safe.

- See Section 3.9.1

Requirement: **Resource specification for the program** *DUNE 47*

status: Approved
tags: Original, Concurrency and multithreading, Resource management
jama: DUNE-DUNE_STKH-47*;*
notes: This is ID #28 from the original DUNE document.

The framework shall enable the specification of resources required by the program.

- See Section 3.9

Requirement: **Resource-based algorithm concurrency** *DUNE 50*

status: Approved
tags: Original, Concurrency and multithreading, Resource management
jama: DUNE-DUNE_STKH-50*;*
notes: This is ID #31 from the original DUNE document.

The framework shall dynamically schedule algorithms to execute efficiently according to the availability of each algorithm's required resources.

- See Section 3.9

Requirement: **Resource specification for algorithms** *DUNE 52*

status: Approved
tags: Original, Concurrency and multithreading, Resource management
jama: DUNE-DUNE_STKH-52*;*
notes: This is ID #33 from the original DUNE document.

The framework shall enable the specification of resources required by each algorithm.

- See Section 3.9

Requirement: **Composable workflows using GPU algorithms and CPU algorithms** *DUNE 54*

status: Approved
tags: Original, Concurrency and multithreading, Resource management
jama: DUNE-DUNE_STKH-54*;*
notes: This is ID #36 from the original DUNE document.

The framework shall support composable workflows that use GPU algorithms along with CPU algorithms.

- See Section 3.9.2

Requirement: **Specification of data products required by an algorithm** *DUNE 65*

status: Approved
tags: Registration
jama: DUNE-DUNE_STKH-65*;*
notes: This is ID #63 as proposed to DUNE.

The framework shall support the specification of data products required as input by an algorithm.

- See Section 3.1, Section 3.4

Requirement: **One configuration per framework execution** *DUNE 69*

status: Approved
tags: Original, Configuration
jama: DUNE-DUNE_STKH-69*;*
notes: This is ID #44 from the original DUNE document.

The framework shall accept exactly one configuration per program execution.

- See Section 3.10

Requirement: **Framework configuration language** *DUNE 72*

status: Approved
tags: Configuration
jama: DUNE-DUNE_STKH-72*;*
notes: This is ID #60 as proposed to DUNE.

The framework shall provide the ability to configure the execution of a framework program at runtime using a human-readable language.

- See Section 3.10

Requirement: **I/O plugins** *DUNE 73*

status: Approved
tags: Data I/O layer, Original
jama: DUNE-DUNE_STKH-73*;*
notes: This is ID #50 from the original DUNE document. Data includes physics data and metadata (both user-provided and framework metadata). The goal is to enable non-framework developers to implement an IO backend without needing to modify the framework itself.

The framework shall provide a public API that enables the implementation of a concrete IO backend for a specific persistent storage format.

- See Section 3.7, Section 3.8

Requirement: **Data I/O backward compatibility** *DUNE 76*

status: Approved
tags: Original, Data I/O layer, Backwards compatibility
jama: DUNE-DUNE_STKH-76*;*
notes: This is ID #54 from the original DUNE document. Backward compatibility means that new code is able to read data produced by older versions of the framework.

The framework IO subsystem shall support backward compatibility across versions, subject to policy decisions on deprecation provided by DUNE.

- See Section 3.7

Requirement: **Support C++ algorithms** *DUNE 81*

status: Approved
tags: General
jama: DUNE-DUNE_STKH-81*;*

The framework shall support the invocation of algorithms written in C++.

- See Section 1.3

Requirement: **Support Python algorithms** *DUNE 82*

status: Approved
tags: General
jama: DUNE-DUNE_STKH-82*;*

The framework shall support the invocation of algorithms written in Python.

- See Section 1.3

Requirement: **Definition of data products** *DUNE 85*

status: Approved
tags: Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-85*;*

The framework shall provide the ability for user-level code to define data products.

- See Section 3.2, Section 3.2.1.2, Section 3.2.2

Requirement: **Creation of data sets** *DUNE 86*

status: Approved
tags: Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-86*;*
notes: This document uses "data cells" rather than "data sets".

The framework shall provide the ability for user-level code to create new data sets.

- See Section 3.2, Section 3.2.2

Requirement: **Definition of data families** *DUNE 87*

status: Approved
tags: Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-87*;*
notes: This document uses "data layers" rather than (this use) of "data families".

The framework shall provide the ability for user-level code to define data families.

- See Section 3.2, Section 3.2.2

Requirement: **Definition of data family hierarchies** *DUNE 88*

status: Approved
tags: Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-88*;*
notes: This document uses "data-layer hierarchies" rather than "data-family hierarchies".

The framework shall provide the ability for user-level code to define hierarchies of data families.

- See Section 3.2, Section 3.2.2

Requirement: **Algorithm invocation with data products from multiple data sets** *DUNE 89*

status: Approved
tags: Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-89*;*
notes: This document uses "data cells" rather than "data sets".

The framework shall allow a single invocation of an algorithm with data products from multiple data sets.

- See Section 3.4.1

Requirement: **Specification of algorithm output FPUs** *DUNE 90*

status: Approved
tags: Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-90*;*
notes: To implement this requirement, the algorithm should not know where its created data products are going--that is something that will be specified at the plugin level (perhaps by configuration). This document uses "data layer" rather than (this use) of "data family".

The framework shall support the user specification of which data family to place the data products created by an algorithm.

- See Section 3.4

Requirement: **Algorithm invocation with data products from adjacent data sets** *DUNE 91*

status: Approved
tags: Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-91*;*
notes: This document uses "data cells" rather than "data sets".

The framework shall support the invocation of an algorithm with data products belonging to adjacent data sets.

- See Section 3.4.1.2

Requirement: **User-defined adjacency** *DUNE 92*

status: Approved
tags: Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-92*;*
notes: This document uses "data cells" rather than "data sets".

The framework shall support user code that defines adjacency of data sets within a data family.

- See Section 3.4.1.2

Requirement: **Algorithm-Data Separability** *DUNE 110*

status: Approved
tags: General
jama: DUNE-DUNE_STKH-110*;*

The data objects exchanged among algorithms shall be separable from those algorithms.

- See Section 3.9.3

Requirement: **Algorithm Communication Via Data Products** *DUNE 111*

status: Approved
tags: General, Reproducibility, Provenance
jama: DUNE-DUNE_STKH-111*;*

The framework shall mediate communication between algorithms via data products.

- See Section 3.2.1

Requirement: **Algorithm invocation with data products from multiple data families** *DUNE 113*

status: Approved
tags: Flexible Processing Unit (FPU)
jama: DUNE-DUNE_STKH-113*;*
notes: This document uses "data layers" rather than (this use) of "data families".

The framework shall allow a single invocation of an algorithm with data products from multiple data families.

- See Section 3.4.1.1

Requirement: **Provenance discovery** *DUNE 121*

status: Approved
tags: Provenance
jama: DUNE-DUNE_STKH-121*;*

The framework shall enable users to discover the provenance of data products.

- See Section 1.2, Section 3.2.1, Section 3.2.3, Section 3.2.4.

Requirement: **Reproducibility of data products** *DUNE 122*

status: Approved
tags: Reproducibility, Provenance
jama: DUNE-DUNE_STKH-122*;*

The framework shall support the reproduction of data products from the provenance stored in the output.

- See Section 3.2.4.

Requirement: **Thread-safe design for algorithms** *DUNE 130*

status: Approved
tags: Concurrency and multithreading
jama: DUNE-DUNE_STKH-130*;*

The framework shall facilitate the development of thread-safe algorithms.

- See Section 2.4, Section 3.2.3

Requirement: **Composing configurations of framework components** *DUNE 133*

status: Approved
tags: Configuration
jama: DUNE-DUNE_STKH-133*;*

The framework shall support executing programs configured by composing configurations of separate components.

- See the technical design *(under preparation)*

Requirement: **Graceful shutdown of framework program** *DUNE 134*

status: Approved
tags: Error handling
jama: DUNE-DUNE_STKH-134*;*
notes: A graceful shutdown refers to a framework program that completes the processing of all in-flight data, safely closes all open input and output files, cleans up connections to external entities (e.g. databases), etc. before the program ends. This ensures that no resources are left in ill-defined states and that all output files are readable and valid. An important example of this is when a batch job exceeds a time limit and the grid system sends a signal to shutdown the job, which should end gracefully.

The framework shall attempt a graceful shutdown by default.

- See Section 1.2.3

Requirement: **Optimize memory management for data products** *DUNE 142*

status: Approved
tags: Resource management
jama: DUNE-DUNE_STKH-142*;*
notes: Optimization means making the data products available for the shortest period of time possible for all algorithms that require them. The framework, however, may need to run in series multiple algorithms requiring those data products if those algorithms would run out of resources if run concurrently.

The framework shall optimize the memory management of data products.

- See Section 3.2.3, Section 3.5.6.3

Requirement: **Serial access to a thread-unsafe resource** *DUNE 145*

status: Approved
tags: Concurrency and multithreading, Original, Resource management
jama: DUNE-DUNE_STKH-145*;*

The framework shall permit algorithm authors to specify that the algorithm requires serial access to a thread-unsafe resource.

- See Section 3.9.1

Requirement: **Specification of user-defined resources** *DUNE 149*

status: Approved
tags: Resource management
jama: DUNE-DUNE_STKH-149*;*

The framework shall enable the specification of user-defined resources required by the program.

- See Section 3.9.4

Requirement: **Specification of algorithm's user-defined resources** *DUNE 155*

status: Approved
tags: Resource management
jama: DUNE-DUNE_STKH-155*;*

The framework shall enable the specification of user-defined resources required by the algorithm.

- See Section 3.9.4

Requirement: **Specification of data products created by an algorithm** *DUNE 156*

status: Approved
tags: Registration
jama: DUNE-DUNE_STKH-156*;*

The framework shall support the specification of data products created as output by an algorithm.

- See Section 3.1, Section 3.4

B.2 Supporting Requirements

Requirement: **Shut down upon unmet algorithm hardware requirements** *DUNE 13*

status: Approved
tags: General
jama: DUNE-DUNE_STKH-13*;*

The framework shall shut down if the platform fails to meet each specified hardware requirement.

Requirement: **Provide instructions for writing algorithms in supported languages** *DUNE 16*

status: Approved
tags: Documentation
jama: DUNE-DUNE_STKH-16*;*

The framework documentation shall provide instructions for writing framework-executable algorithms in supported languages.

Requirement: **Data product I/O independence** *DUNE 24*

status: Approved
tags: Original, Data I/O layer
jama: DUNE-DUNE_STKH-24*;*
notes: This is ID #14 from the original DUNE document.

The framework shall support reading from disk only the data products required by a given algorithm.

Requirement: **Framework configuration persistency** *DUNE 27*

status: Approved
tags: Original, Configuration, Reproducibility, Provenance
jama: DUNE-DUNE_STKH-27*;*
notes: This is ID #17 from the original DUNE document. This requirement is in support of documenting and reproducing previous results.

The framework shall provide an option to persist the configuration of each framework execution to the output of that execution.

- See the technical design *(under preparation)*

Requirement: **Record execution environment** *DUNE 30*

status: Approved
tags: Original, Reproducibility, Provenance
jama: DUNE-DUNE_STKH-30*;*
notes: This is ID #20 from the original DUNE document.

The framework shall record the job's execution environment.

Requirement: **Maximum memory usage** *DUNE 31*

status: Approved
tags: Original, Memory management, Resource management
jama: DUNE-DUNE_STKH-31*;*
notes: This is ID #59 from the original DUNE document. The maximum memory available is a static quantity that can apply to (a) a job using an entire node with all of its available RSS, and (b) a job using a specific grid slot with a limit on the RSS. It is assumed that the operating system and C++/Python runtimes are not already enforcing this limit.

The framework shall gracefully shut down if the program attempts to exceed a configured memory limit.

Requirement: **Local GPU algorithm support** *DUNE 41*

status: Approved
tags: Services
jama: DUNE-DUNE_STKH-41*;*
notes: This is ID #69 as proposed to DUNE.

The framework shall support algorithms that perform calculations using a local GPU.

Requirement: **Remote GPU algorithm support** *DUNE 42*

status: Approved
tags: Services
jama: DUNE-DUNE_STKH-42*;*
notes: This is ID #70 as proposed to DUNE.

The framework shall support algorithms that perform calculations using a remote GPU.

Requirement: **Intra-algorithm concurrency and multi-threading** *DUNE 46*

status: Approved
tags: Original, Concurrency and multithreading
jama: DUNE-DUNE_STKH-46*;*
notes: This is ID #27 from the original DUNE document. It is the responsibility of the algorithm author to ensure that any parallelism libraries used can work compatibly with those used by the framework itself.

The framework shall allow algorithms to use the same parallelism mechanisms the framework uses to schedule the execution of algorithms.

- See the technical design *(under preparation)*

Requirement: **Logging resource usage per algorithm invocation** *DUNE 48*

status: Approved
tags: Original, Concurrency and multithreading, Resource management
jama: DUNE-DUNE_STKH-48*;*
notes: This is ID #29 from the original DUNE document.

The framework shall support logging the usage of a specified resource for each algorithm using the resource.

Requirement: **Emit message stating algorithm resource requirements** *DUNE 56*

status: Approved
tags: Original, Concurrency and multithreading, Resource management
jama: DUNE-DUNE_STKH-56*;*
notes: This is ID #38 from the original DUNE document.

The framework shall have an option to emit a message stating the resources required by each algorithm of a configured program without executing the workflow.

Requirement: **Monitoring global memory use** *DUNE 59*

status: Approved
tags: General, Memory management, Resource management
jama: DUNE-DUNE_STKH-59*;*
notes: This is ID #67 as proposed to DUNE.

The framework shall be able to report the global memory use of the framework program at user-specified points in time.

- See the technical design *(under preparation)*

Requirement: **Elapsed time information** *DUNE 60*

status: Approved
tags: General, Resource management
jama: DUNE-DUNE_STKH-60*;*
notes: This is ID #66 as proposed to DUNE. This option is intended to capture wall-clock time and not CPU time. If more granular reporting of CPU vs. IO time is required, dedicated profiling tools like VTune or Linaro Forge should be used.

The framework shall have an option to provide elapsed time information for each algorithm executed in a framework program.

- See the technical design *(under preparation)*

Requirement: **Framework-independent message logging** *DUNE 61*

status: Approved
tags: General
jama: DUNE-DUNE_STKH-61*;*
notes: This is ID #65 as proposed to DUNE.

The framework shall support a logging solution that is usable in an algorithm without that algorithm explicitly relying on the framework.

- See the technical design *(under preparation)*

Requirement: **Independence from unique hardware characteristics** *DUNE 63*

status: Approved
tags: Reproducibility, Provenance
jama: DUNE-DUNE_STKH-63*;*
notes: This is ID #62 as proposed to DUNE.

The framework shall operate independently of unique characteristics of existing hardware.

- See Section 1.2.2, Section 3.2.1

Requirement: **Configuration validation** *DUNE 64*

status: Approved
tags: Original, Configuration
jama: DUNE-DUNE_STKH-64*;*
notes: This is ID #42 from the original DUNE document.

The framework shall validate an algorithm's configuration against specifications provided at registration time.

- See the technical design *(under preparation)*

Requirement: **Algorithm configuration schema availability** *DUNE 67*

status: Approved
tags: Original, Configuration
jama: DUNE-DUNE_STKH-67*;*
notes: This is ID #43 from the original DUNE document.

The framework shall have an option to emit an algorithm's configuration schema in human-readable form.

- See the technical design *(under preparation)*

Requirement: **Emit message describing data flow of program** *DUNE 68*

status: Approved
tags: Concurrency and multithreading
jama: DUNE-DUNE_STKH-68*;*
notes: This is ID #64 as proposed to DUNE.

The framework shall have an option to emit a description of the data flow of a configured program without executing the workflow.

Requirement: **Eager validation of algorithm configuration** *DUNE 70*

status: Approved
tags: Original, Configuration
jama: DUNE-DUNE_STKH-70*;*
notes: This is ID #45 from the original DUNE document. Validation includes any reading, parsing, canonicalizing, and checking against applicable schemata.

The framework shall validate the configuration of each algorithm before that algorithm processes data.

- See the technical design *(under preparation)*, the technical design *(under preparation)*.

Requirement: **I/O backend for ROOT** *DUNE 74*

status: Approved
tags: Data I/O layer, Original
jama: DUNE-DUNE_STKH-74*;*
notes: This is ID #51 from the original DUNE document.

The framework ecosystem shall support a ROOT IO backend.

- See Section 3.2.1.

Requirement: **Configurable data compression** *DUNE 77*

status: Approved
tags: Original, Data I/O layer
jama: DUNE-DUNE_STKH-77*;*
notes: This is ID #55 from the original DUNE document.

The framework IO subsystem shall allow user-configuration of compression settings for each concrete IO implementation.

Requirement: **User-configurable output file rollover** *DUNE 78*

status: Approved
tags: Original, Data I/O layer
jama: DUNE-DUNE_STKH-78*;*
notes: This is ID #56 from the original DUNE document.

The framework shall support user-configurable rollover of output files.

Requirement: **Configuration comparison** *DUNE 98*

status: Approved
tags: Configuration
jama: DUNE-DUNE_STKH-98*;*

The framework shall provide the ability to compare two configurations.

- See the technical design *(under preparation)*

Requirement: **User-selectable list of recordable execution environment components** *DUNE 100*

status: Approved
tags: Reproducibility, Provenance
jama: DUNE-DUNE_STKH-100*;*

The framework shall provide the list of recordable components of the execution environment.

Requirement: **Save user-selected execution environment components** *DUNE 101*

status: Approved
tags: Reproducibility, Provenance
jama: DUNE-DUNE_STKH-101*;*

The framework shall save each execution-environment description selected by the user from the framework-provided-list.

Requirement: **Data product backward compatibility** *DUNE 106*

status: Approved
tags: Data I/O layer, Backwards compatibility
jama: DUNE-DUNE_STKH-106*;*

The framework's IO subsystem shall support backward compatibility of data products.

Requirement: **Metadata backward compatibility** *DUNE 107*

status: Approved
tags: Data I/O layer, Backwards compatibility
jama: DUNE-DUNE_STKH-107*;*
notes: Metadata here can include user-visible (including user-provided) metadata, and framework metadata, which may not be user-visible but is necessary for framework operation.

The framework's IO subsystem shall support backward compatibility of metadata.

Requirement: **Output file rollover due to number of data sets in data family** *DUNE 109*

status: Approved
tags: Data I/O layer
jama: DUNE-DUNE_STKH-109*;*
notes: Some examples include limiting the output file to contain data for: (a) 1 subrun ("subrun" is the user-specified family) (b) 100 spills ("spill" is the user-specified family) (c) 10 blobs ("blob" is the user-specified family) This document uses "data cells" rather than "data sets", and "data layer" rather than (this use) of "data family".

The framework shall have an option to rollover output files according to a configurable limit on the number of data sets in a user-specified data family.

Requirement: **Emit diagnostic upon unmet algorithm hardware requirements** *DUNE 112*

status: Approved
tags: General
jama: DUNE-DUNE_STKH-112*;*

The framework shall emit a diagnostic message for each hardware requirement the platform fails to meet.

Requirement: **ProtoDUNE single-phase raw data** *DUNE 116*

status: Deferred
tags: Backwards compatibility, Data I/O layer
jama: DUNE-DUNE_STKH-116*;*
notes: ProtoDUNE single-phase was used in run 1. For this requirement, the framework *ecosystem* is responsible for processing run 1 data (e.g. the framework might not read the run 1 data directly, but a translation program might first prepare the run 1 data for reading within the framework).

The framework ecosystem shall support processing ProtoDUNE single-phase raw data.

Requirement: **ProtoDUNE dual-phase raw data** *DUNE 117*

status: Deferred
tags: Backwards compatibility, Data I/O layer
jama: DUNE-DUNE_STKH-117*;*
notes: ProtoDUNE dual-phase was used in run 1. For this requirement, the framework *ecosystem* is responsible for processing run 1 data (e.g. the framework might not read the run 1 data directly, but a translation program might first prepare the run 1 data for reading within the framework).

The framework ecosystem shall support processing ProtoDUNE dual-phase raw data.

Requirement: **ProtoDUNE II horizontal-drift raw data** *DUNE 118*

status: Approved
tags: Backwards compatibility, Data I/O layer
jama: DUNE-DUNE_STKH-118*;*

The framework ecosystem shall support processing ProtoDUNE II horizontal-drift raw data.

Requirement: **ProtoDUNE II vertical-drift raw data** *DUNE 119*

status: Approved
tags: Backwards compatibility, Data I/O layer
jama: DUNE-DUNE_STKH-119*;*

The framework ecosystem shall support processing ProtoDUNE II vertical-drift raw data.

Requirement: **Write collections of unconstrained size** *DUNE 120*

status: Approved
tags: Original, Memory management
jama: DUNE-DUNE_STKH-120*;*

The framework shall support the writing of collections too large to hold in memory.

Requirement: **Record user-selected items from the shell environment** *DUNE 123*

status: Approved
tags: Provenance
jama: DUNE-DUNE_STKH-123*;*

The framework shall record user-selected items from the shell environment.

Requirement: **User-provided execution environment information** *DUNE 124*

status: Approved
tags: Provenance
jama: DUNE-DUNE_STKH-124*;*

The framework shall record labelled execution environment information provided by the user.

Requirement: **Command line interface (CLI)** *DUNE 125*

status: Approved
tags: Configuration
jama: DUNE-DUNE_STKH-125*;*

The framework shall provide a command-line interface that allows the setting of configuration parameters.

- See the technical design *(under preparation)*

Requirement: **Support local configuration changes** *DUNE 126*

status: Approved
tags: Configuration
jama: DUNE-DUNE_STKH-126*;*

The framework shall support the use of local configuration changes with respect to a separate complete configuration to modify the execution of a program.

- See the technical design *(under preparation)*

Requirement: **Configuration tracing** *DUNE 127*

status: Approved
tags: Configuration
jama: DUNE-DUNE_STKH-127*;*

The framework configuration system shall have an option to provide diagnostic information for an evaluated configuration, including origins of final parameter values.

- See the technical design *(under preparation)*

Requirement: **Configuration language single point of maintenance** *DUNE 128*

status: Approved
tags: Configuration
jama: DUNE-DUNE_STKH-128*;*
notes: This must be met by each configuration language.

The language used for configuring a framework program shall include features for maintaining hierarchical configurations from a single point of maintenance.

- See the technical design *(under preparation)*

Requirement: **Enable identification of data sets containing chunked data products** *DUNE 129*

status: Approved
tags: Provenance, Chunkification
jama: DUNE-DUNE_STKH-129*;*
notes: This document uses "data cells" rather than "data sets".

The framework shall record metadata identifying data sets where the framework took special measures to process data collections of unconstrained size.

Requirement: **Framework build flags** *DUNE 131*

status: Approved
tags: Debugging
jama: DUNE-DUNE_STKH-131*;*

The framework build system shall support options that enable debugging executed code.

Requirement: **Floating-point environment** *DUNE 132*

status: Approved
tags: Reproducibility, Error handling, Debugging
jama: DUNE-DUNE_STKH-132*;*

The framework shall allow the per-execution setting of the float-point environment to control the handling of IEEE floating-point exceptions.

Requirement: **Graceful shutdown for uncaught exception** *DUNE 135*

status: Approved
tags: Error handling
jama: DUNE-DUNE_STKH-135*;*

The framework shall by default attempt a graceful shutdown upon receiving an uncaught exception from user algorithms.

Requirement: **Graceful shutdown for received signal** *DUNE 136*

status: Approved
tags: Error handling
jama: DUNE-DUNE_STKH-136*;*

The framework shall by default attempt a graceful shutdown when receiving a signal.

Requirement: **Diagnostic message when exceeding memory limit** *DUNE 137*

status: Approved
tags: Memory management, Error handling
jama: DUNE-DUNE_STKH-137*;*

The framework shall emit a diagnostic message if the program attempts to exceed the configured maximum memory.

Requirement: **Output file rollover due to file size** *DUNE 138*

status: Approved
tags: Data I/O layer
jama: DUNE-DUNE_STKH-138*;*

The framework shall have an option to rollover output files according to a configurable limit on output-file size.

Requirement: **Output file rollover due to user-defined quantities** *DUNE 139*

status: Approved
tags: Data I/O layer
jama: DUNE-DUNE_STKH-139*;*
notes: An example of an aggregated value of a user-derived quantity is the number of protons on target (POTs).

The framework shall have an option to rollover output files according to a configurable limit on the aggregated value of a user-derived quantity.

Requirement: **Output file rollover due to file lifetime** *DUNE 140*

status: Approved
tags: Data I/O layer
jama: DUNE-DUNE_STKH-140*;*

The framework shall have an option to rollover output files according to a configurable limit on the time the file has been open.

Requirement: **I/O backend for HDF5** *DUNE 141*

status: Approved
tags: Data I/O layer, Original
jama: DUNE-DUNE_STKH-141*;*

The framework ecosystem shall support an HDF5 IO backend.

- See Section 3.2.1.

Requirement: **Optimize availability of external resources** *DUNE 143*

status: Approved
tags: Resource management
jama: DUNE-DUNE_STKH-143*;*
notes: Examples of external resources include local GPUs, remote inference servers, and databases. This requirement also serves as a replacement for a former requirement: "The framework MUST be able to broker/cache service replies that might be common to multiple instances of algorithms (reduce load on external service/disk/memory/network/...). E.g. a request for a calibration constant that is common among all FPUs in a run. Not every instance of an algorithm should trigger an actual request to the central database providing these."

The framework shall optimize the availability of external resources.

Requirement: **Efficient execution of algorithms requiring access to a network resource** *DUNE 144*

status: Approved
tags: Concurrency and multithreading, Original, Resource management
jama: DUNE-DUNE_STKH-144*;*
notes: An example of efficient execution is an algorithm requiring network resource does not occupy a core that can do other work while the algorithm "waits" for the resource to respond.

The framework shall efficiently execute a graph of algorithms where at least one algorithm requires access to a network resource.

Requirement: **Specification of maximum number of CPU threads** *DUNE 146*

status: Approved
tags: Resource management
jama: DUNE-DUNE_STKH-146*;*

The framework shall enable the specification of the maximum number of CPU threads permitted by the program.

Requirement: **Specification of maximum allowed CPU memory** *DUNE 147*

status: Approved
tags: Resource management, Memory management
jama: DUNE-DUNE_STKH-147*;*

The framework shall enable the specification of the maximum CPU memory allowed by the program.

Requirement: **Specification of GPU resources** *DUNE 148*

status: Approved
tags: Resource management
jama: DUNE-DUNE_STKH-148*;*

The framework shall enable the specification of GPU resources required by the program.

Requirement: **Efficient execution of algorithms with specific CPU memory requirements** *DUNE 150*

status: Approved
tags: Resource management, Memory management
jama: DUNE-DUNE_STKH-150*;*

The framework shall efficiently execute a graph of algorithms where at least one algorithm specifies a required amount of CPU memory.

Requirement: **Efficient execution of algorithms with specific GPU memory requirements** *DUNE 151*

status: Approved
tags: Resource management, Memory management
jama: DUNE-DUNE_STKH-151*;*

The framework shall efficiently execute a graph of algorithms where at least one algorithm specifies a required amount of GPU memory.

Requirement: **Specification of algorithm's maximum number of CPU threads** *DUNE 152*

status: Approved
tags: Resource management
jama: DUNE-DUNE_STKH-152*;*

The framework shall enable the specification of the maximum number of CPU threads permitted by the algorithm.

- See Section 3.4

Requirement: **Specification of algorithm's GPU resources** *DUNE 153*

status: Approved
tags: Resource management
jama: DUNE-DUNE_STKH-153*;*

The framework shall enable the specification of GPU resources required by the algorithm.

Requirement: **Specification of algorithm's CPU memory usage** *DUNE 154*

status: Approved
tags: Resource management, Memory management
jama: DUNE-DUNE_STKH-154*;*

The framework shall enable the specification of an algorithm’s expected CPU memory usage.

- See Section 3.2.1.1

BIBLIOGRAPHY

[Wiki-Framework] https://en.wikipedia.org/w/index.php?title=Software_framework&oldid=1285034658

[Jama-Connect] https://www.jamasoftware.com/platform/jama-connect/

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

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

[O2] 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)

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

[Meld] K. Knoepfel, EPJ Web of Conferences **295**, 05014 (2024)

[SPEC-0] https://scientific-python.org/specs/spec-0000/

[Wiki-Pure] https://en.wikipedia.org/wiki/Pure_function

[Wiki-HOF] https://en.wikipedia.org/wiki/Higher-order_function

[Bird] R. Bird, Introduction to Functional Programming using Haskell (2nd ed.), Prentice Hall (1988), pp. 131–132

[Cpp-Function] https://en.cppreference.com/w/cpp/language/function.html

[Cpp-UserLiteral] https://en.cppreference.com/w/cpp/language/user_literal.html

[Wiki-Partition] https://en.wikipedia.org/wiki/Partition_of_a_set

[Wiki-CBRNG] https://en.wikipedia.org/wiki/Counter-based_random_number_generator

INDEX