Adaptable framework LDRD update

https://indico.fnal.gov/event/52666/contributions/231769/attachments/153101/198580/SCDProjects_LDRD_Knoepfel.pdf

Kyle J. Knoepfel

DUNE/LDRD monthly meeting

9 November 2022

Framework-supported algorithm constructs

Framework-agnostic constructs:

- **Transform** (producer) – creates data products from existing data of the same processing level
- **Reduction** (producer) – creates data products based on accumulations of data at a more granular processing level (e.g. endSubRun)
- **Monitor** (analyzer) – consumes data products and does not produce any new data
- **Filter** – supports processing a subset of data based on satisfying Boolean criteria

Framework-supported algorithm constructs

Framework-agnostic constructs:

- **Transform** (producer) – creates data products from existing data of the same processing level
- **Reduction** (producer) – creates data products based on accumulations of data at a more granular processing level (e.g. endSubRun)
- **Monitor** (analyzer) – consumes data products and does not produce any new data
- **Filter** – supports processing a subset of data based on satisfying Boolean criteria

Framework-aware constructs:

- **Source** – creates product stores that provide data products
- **Splitter** – splits existing product stores into smaller ones for downstream processing
- **Output** – writes product stores to an output file, stream, etc.

Framework glue code

- **Naturally separates user code from framework assumptions**
  - The input arguments for each registered function are data products produced by upstream functions or provided by the input source.
  - The return value(s) of each registered function are registered as a data product.

Framework glue code

- **Naturally separates user code from framework assumptions**
  - The input arguments for each registered function are data products produced by upstream functions or provided by the input source.
  - The return value(s) of each registered function are registered as a data product.

```
constexpr int add(int i, int j) { return i + j; }
```

Framework glue code

- **Naturally separates user code from framework assumptions**
  - The input arguments for each registered function are data products produced by upstream functions or provided by the input source.
  - The return value(s) of each registered function are registered as a data product.

```
constexpr int add(int i, int j) { return i + j; }
```

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

DEFINE_MODULE(m) // pset can also be passed in
{
  m.declare_transform("add", add)
   .concurrency(unlimited)
   .input(”num", ”neg_num")
   .output("sum");
}
```

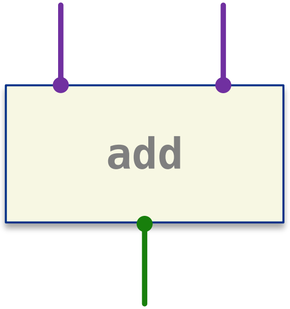
Framework glue code

- **Naturally separates user code from framework assumptions**
  - The input arguments for each registered function are data products produced by upstream functions or provided by the input source.
  - The return value(s) of each registered function are registered as a data product.

```
constexpr int add(int i, int j) { return i + j; }
```

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

DEFINE_MODULE(m) // pset can also be passed in
{
  m.declare_transform("add", add)
   .concurrency(unlimited)
   .input(”num", ”neg_num")
   .output("sum");
}
```

Generates graph node



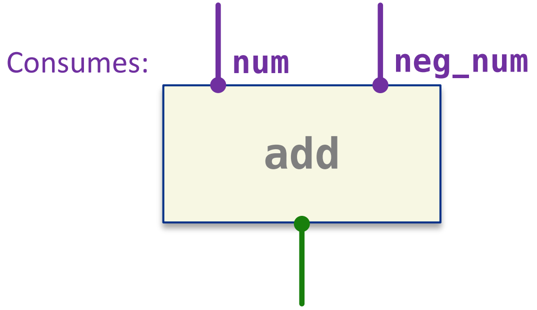
Framework glue code

- **Naturally separates user code from framework assumptions**
  - The input arguments for each registered function are data products produced by upstream functions or provided by the input source.
  - The return value(s) of each registered function are registered as a data product.

```
constexpr int add(int i, int j) { return i + j; }
```

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

DEFINE_MODULE(m) // pset can also be passed in
{
  m.declare_transform("add", add)
   .concurrency(unlimited)
   .input(”num", ”neg_num")
   .output("sum");
}
```

Generates graph node



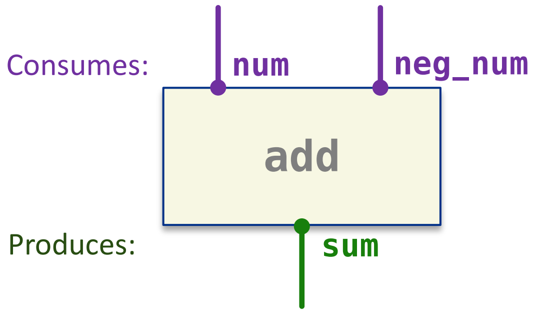
Framework glue code

- **Naturally separates user code from framework assumptions**
  - The input arguments for each registered function are data products produced by upstream functions or provided by the input source.
  - The return value(s) of each registered function are registered as a data product.

```
constexpr int add(int i, int j) { return i + j; }
```

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

DEFINE_MODULE(m) // pset can also be passed in
{
  m.declare_transform("add", add)
   .concurrency(unlimited)
   .input(”num", ”neg_num")
   .output("sum");
}
```

Generates graph node



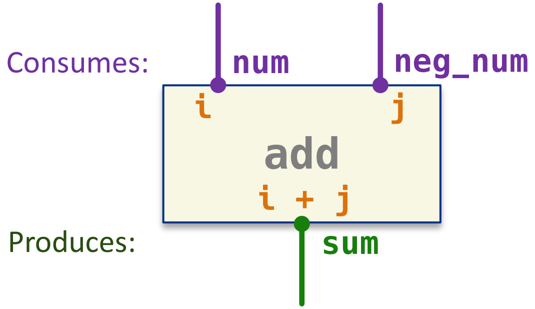
Framework glue code

- **Naturally separates user code from framework assumptions**
  - The input arguments for each registered function are data products produced by upstream functions or provided by the input source.
  - The return value(s) of each registered function are registered as a data product.

```
constexpr int add(int i, int j) { return i + j; }
```

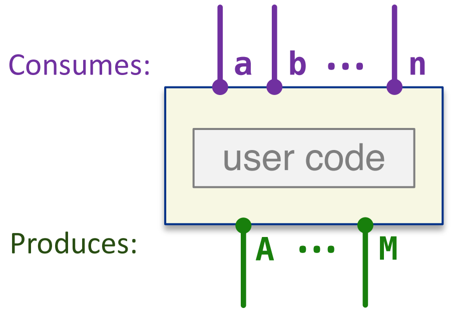
```
#include "meld/module.hpp"

DEFINE_MODULE(m) // pset can also be passed in
{
  m.declare_transform("add", add)
   .concurrency(unlimited)
   .input(”num", ”neg_num")
   .output("sum");
}
```

Generates graph node



Graph nodes

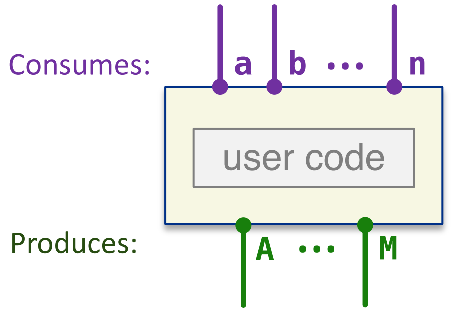
- For *framework-agnostic constructs*, framework details **do not need to be** accessed by the user within the node.

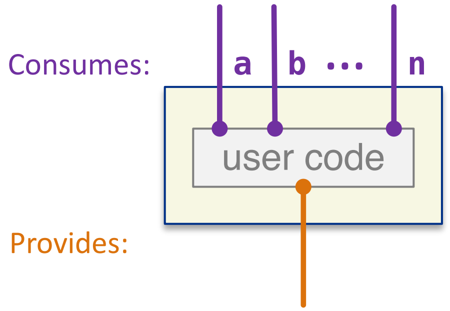


Graph nodes

- For *framework-agnostic constructs*, framework details **do not need to be** accessed by the user within the node.

- For *framework-aware constructs*, framework details **must be** accessed by the user within the node.





In the last month

- I have implemented filters that are specifiable via configuration:

  Allow all modules (except sources) to specify preceding filters

```
{
  source: {
    plugin: 'source_t',
    max_numbers: 10,
  },
  modules: {
    add: {
      plugin: 'module_t',
      filtered_by: ['only_evens', 'greater_than_5'], # logical AND of specified filters
    },
  },
}
```

Fermilab

Next steps

- Support non-product inputs to user functions
- Run performance tests against *art*
- Start looking at I/O
  - Includes eager writing of data products
  - Exploring HDF5 and other technologies
  - Considering a general dictionary system that can be used in addition to ROOT’s
- Explore paths and backwards compatibility
- Thoughts?