TOWARDS A FRAMEWORK-INDEPENDENT ALGORITHM LIBRARY FOR EIC AND BEYOND

Argonne NATIONAL LABORATORY

algorithms

```
using ClusteringAlgorithm = Algorithm<
    Input<edm4eic::ProtoClusterCollection,
          std::optional<edm4hep::SimCalorimeterHitCollection>>,
    Output<edm4eic::ClusterCollection,
           std::optional<edm4eic::MCRecoClusterParticleAssociation
```

SYLVESTER JOOSTEN
sjoosten@anl.gov

WOUTER DECONINCK
wouter.deconinck@umanitoba.ca

This work is supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics, under contract DE-AC02-06CH11357.

Joint EIC & Key4hep Meeting
September 21, 2022

DESIGN GOALS AND CHALLENGES

Towards a first prototype for `algorithms`

DESIGN GOALS

- Enable algorithm sharing across experiments and even communities
- Framework agnostic algorithms
- Main dependencies: EDM4hep/ EDM4eic and DD4hep
- Showcase independence through both Gaudi and JANA2 integration
- Minimal boilerplate for integration
- Avoid duplication of definitions

CHALLENGES

- Service integration
- Data store interactions
- Properties
- Some definition duplication and manual glue code unavoidable
- Automatic testing in a no-framework context

CHALLENGE 1: SERVICE INTEGRATION

Towards a first prototype for `algorithms`

```
// Thread-safe lazy-evaluated minimal service system
// CRTP base class to add the instance method
// This could have been part of DEFINE_SERVICE macro, but I think it is better
// to keep the macro magic to a minimum to maximize transparency
template <class SvcType> class Service : public PropertyMixin, public NameMixin {
public:
  static SvcType& instance() {
    // This is guaranteed to be thread-safe from C++11 onwards.
    static SvcType svc;
    return svc;
  }
  // constructor for the service base class registers the service, except
  // for the ServiceSvc which is its own thing (avoid circularity)
  Service(std::string_view name) : NameMixin{name} { ServiceSvc::instance().add(name, this); }
};

} // namespace algorithms
```

```
// Note: the log action is responsible for dealing with concurrent calls
//       the default LogAction is a thread-safe example
class LogSvc : public Service<LogSvc> {
public:
  using LogAction = std::function<void(LogLevel, std::string_view, std::string_view)>;
  void defaultLevel(const LogLevel l) { m_level.set(l); }
  LogLevel defaultLevel() const { return m_level; }
  void action(LogAction a) { m_action = a; }
  void report(const LogLevel l, std::string_view caller, std::string_view msg) const {
    m_action(l, caller, msg);
  }

private:
  Property<LogLevel> m_level{this, "defaultLevel", LogLevel::kInfo};
  LogAction m_action = [](const LogLevel l, std::string_view caller, std::string_view msg) {
    static std::mutex m;
    std::lock_guard<std::mutex> lock(m);
    fmt::print("{} [{}] {}\n", logLevelName(l), caller, msg);
  };
  ALGORITHMS_DEFINE_SERVICE(LogSvc)
};
```

- Services as lazy-evaluated singletons
- Support standalone minimal interface
  - Interface has usable defaults for standalone operation
  - Standalone defaults are meant to be overridden by the framework
- Prototype currently implements `LogSvc` and `GeoSvc`
- Special `ServiceSvc` provides framework with all required services, so it can handle the bindings

EXAMPLE SERVICE INTEGRATION (JUGGLER)

Towards a first prototype for `algorithms`

```
auto& serviceSvc = algorithms::ServiceSvc::instance();
info() << "ServiceSvc declared " << serviceSvc.services().size() << " services" << endmsg;
// loop over all services and handle each properly
// Note: this code is kind of dangerous, as getting the types wrong will lead to
// undefined runtime behavior.
for (auto [name, svc] : serviceSvc.services()) {
  if (name == algorithms::LogSvc::kName) {
    auto* logger = static_cast<algorithms::LogSvc*>(svc);
    const algorithms::LogLevel level{
        static_cast<algorithms::LogLevel>(msgLevel() > 0 ? msgLevel() - 1 : 0)};
    info() << "Setting up algorithms::LogSvc with default level "
           << algorithms::logLevelName(level) << endmsg;
    logger->defaultLevel(level);
    logger->action(
        [this](const algorithms::LogLevel l, std::string_view caller, std::string_view msg) {
          const std::string text = fmt::format("[{}] {}", caller, msg);
          if (l == algorithms::LogLevel::kCritical) {
            this->fatal() << text << endmsg;
          } else if (l == algorithms::LogLevel::kError) {
            this->error() << text << endmsg;
          } else if (l == algorithms::LogLevel::kWarning) {
            this->warning() << text << endmsg;
          } else if (l == algorithms::LogLevel::kInfo) {
            this->info() << text << endmsg;
          } else if (l == algorithms::LogLevel::kDebug) {
            this->debug() << text << endmsg;
          } else if (l == algorithms::LogLevel::kTrace) {
            this->verbose() << text << endmsg;
          }
        });
    // set own log level to verbose so we actually display everything that is requested
    // (this overrides what was initally set through the OutputLevel property)
    updateMsgStreamOutputLevel(MSG::VERBOSE);
  } else if (name == algorithms::GeoSvc::kName) {
    // Setup geometry service
    m_geoSvc = service("GeoSvc");
    if (!m_geoSvc) {
      error() << "Unable to locate Geometry Service. "
              << "Make sure you have GeoSvc in the right order in the configuration." << endmsg;
      return StatusCode::FAILURE;
    }
    info() << "Setting up algorithms::GeoSvc" << endmsg;
    auto* geo = static_cast<algorithms::GeoSvc*>(svc);
    geo->init(m_geoSvc->detector());
  }
}
```

Juggler integration

What services does `algorithms` need?

Link the logger to the Gaudi logger

Link the DD4hep GeoSvc to the Juggler GeoSvc

CHALLENGE 2: DATA STORE INTERACTIONS

Towards a first prototype for `algorithms`

```
using ClusteringAlgorithm = Algorithm<
    Input<edm4eic::ProtoClusterCollection,
          std::optional<edm4hep::SimCalorimeterHitCollection>>,
    Output<edm4eic::ClusterCollection,
           std::optional<edm4eic::MCRecoClusterParticleAssociationCollection>>>;

class ClusterRecoCoG : public ClusteringAlgorithm {
public:
  using Input      = ClusteringAlgorithm::Input;
  using Output     = ClusteringAlgorithm::Output;
  using WeightFunc = std::function<double(double, double, double)>;

  ClusterRecoCoG(std::string_view name)
      : ClusteringAlgorithm{name,
                            {"inputProtoClusterCollection", "mcHits"},
                            {"outputClusterCollection", "outputAssociations"}} {}
```

```
void ClusterRecoCoG::process(const ClusterRecoCoG::Input& input,
                             const ClusterRecoCoG::Output& output) {
  const auto [proto, opt_simhits] = input;
  auto [clusters, opt_assoc]      = output;

  for (const auto& pcl : *proto) {
    auto cl = reconstruct(pcl);

    if (aboveDebugThreshold()) {
      debug() << cl.getNhits() << " hits: " << cl.getEnergy() / dd4hep::GeV <<
              << cl.getPosition().x / dd4hep::mm << ", " << cl.getPosition().y
              << cl.getPosition().z / dd4hep::mm << ")" << endmsg;
    }
    clusters->push_back(cl);
```

- Needed to choose between (1) providing `algorithms` with a framework allocator, (2) going with a purely functional approach, or (3) passing pointers to already existing objects
- Chose (3) (tuple of pointers) as it significantly simplifies interactions with the frameworks
- Algorithm definition takes an `Input` and an `Output` type to define the signature of the `::process` function
- Special cases for `std::vector<T>` (to handle multiple objects of the same type) and `std::optional<T>` (to handle optional data, e.g. MC truth info in reconstruction algorithms)

CHALLENGE 3: PROPERTIES

Towards a first prototype for `algorithms`

```
private:
  edm4eic::MutableCluster reconstruct(const edm4eic::ProtoCluster&) cor

  Property<double> m_sampFrac{this, "samplingFraction", 1.0};
  Property<double> m_logWeightBase{this, "logWeightBase", 3.6};
  Property<std::string> m_energyWeight{this, "energyWeight", "log"};
  Property<std::string> m_moduleDimZName{this, "moduleDimZName", ""};
  Property<bool> m_enableEtaBounds{this, "enableEtaBounds", true};

  WeightFunc m_weightFunc;

  const GeoSvc& m_geo = GeoSvc::instance();
};
} // namespace algorithms::calorimetry
```

```
public:
  ClusterRecoCoG(const std::string& name, ISvcLocator* svcLoc) : Al

  virtual StatusCode configure() {
    setAlgoProp("samplingFraction", m_sampFrac.value());
    setAlgoProp("logWeightBase", m_logWeightBase.value());
    setAlgoProp("energyWeight", m_energyWeight.value());
    setAlgoProp("moduleDimZName", m_moduleDimZName.value());
    setAlgoProp("enableEtaBounds", m_enableEtaBounds.value());
    return StatusCode::SUCCESS;
  }
```

Juggler integration

- Need a way to define properties for algorithms
- Ideally they should provide for a programatic way to deal with automatic initialization at the framework end (non-trivial)
- Currently choose a Gaudi-like `Property<T>` class that has run-time performance of a bare T, while providing an avenue for the framework to set the property
- Automatic handling may be possible in the future but outside the scope of this prototype implementation

EXAMPLE ALGORITHM INTEGRATION (JUGGLER)

Towards a first prototype for `algorithms`

```cpp
#include <JugAlgo/Algorithm.h>
#include <algorithms/calorimetry/ClusterRecoCoG.h>

#include "Gaudi/Property.h"

namespace Jug::Reco {

namespace {
  using AlgoBase = Jug::Algo::Algorithm<algorithms::calorimetry::ClusterRecoCoG>;
}

class ClusterRecoCoG : public AlgoBase {

public:
  ClusterRecoCoG(const std::string& name, ISvcLocator* svcLoc) : AlgoBase(name, svcLoc) {}

  virtual StatusCode configure() {
    setAlgoProp("samplingFraction", m_sampFrac.value());
    setAlgoProp("logWeightBase", m_logWeightBase.value());
    setAlgoProp("energyWeight", m_energyWeight.value());
    setAlgoProp("moduleDimZName", m_moduleDimZName.value());
    setAlgoProp("enableEtaBounds", m_enableEtaBounds.value());
    return StatusCode::SUCCESS;
  }

private:
  Gaudi::Property<double> m_sampFrac{this, "samplingFraction", 1.0};
  Gaudi::Property<double> m_logWeightBase{this, "logWeightBase", 3.6};
  Gaudi::Property<std::string> m_energyWeight{this, "energyWeight", "log"};
  Gaudi::Property<std::string> m_moduleDimZName{this, "moduleDimZName", ""};
  Gaudi::Property<bool> m_enableEtaBounds{this, "enableEtaBounds", false};
};

// NOLINTNEXTLINE(cppcoreguidelines-avoid-non-const-global-variables)
DECLARE_COMPONENT(ClusterRecoCoG)

} // namespace Jug::Reco
```

Juggler integration

Include the Juggler `algorithms` bindings and the actual algorithm implementation

Instantiate a Juggler algorithm based on the `algorithms` algorithm

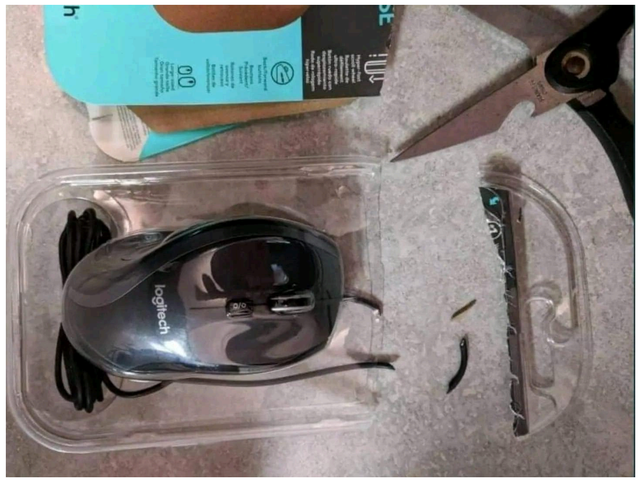
Minimal Gaudi boilerplate

Only real code: handle properties

CHALLENGE 4&5: DUPLICATION, AND TESTING

Towards a first prototype for `algorithms`

- Avoided duplication for:
  - Data store interaction without duplication (handled automagically by `JugAlgo::Algorithm` in prototype)
  - Service interactions without duplication (handled by `JugAlgo::AlgoServiceSvc`)
- So far did *not* avoid duplication in Property handling (possible source of errors, should be addressed in future)
- Need to define testing strategy. While integration tests with frameworks are useful, stand-alone unit tests of the library algorithms would be valuable



OUTLOOK

Towards a first prototype for `algorithms`

- ✓ Library infrastructure code ready
- 🚧 Gaudi bindings being tested
- 🚧 Once prototype testing complete (soon), will move to finish proof-of-concept by providing JANA2/EICRecon bindings (together with Dmitry Romanov)
- ✖Then can start migrating ongoing development work (e.g. on tracking with ACTS) into `algorithms`
- ✖Anticipate a first pre-release version on GitHub soon (~week)
- 🚧 Explore collaboration with Key4hep