Algorithms and Tools

Edward Moyse

Introduction

You have already heard about Gaudi/Athena yesterday but as a quick reminder:

- Athena uses a ‘whiteboard’ model, where we read and write event data to a transient data store (StoreGate) and which implies separation of algorithms and data (i.e. we do not have smart data objects)
- An Athena job is made of Algorithms which can e.g. read/write data to StoreGate, and which can also call Services, AlgTools etc
- The order in which algorithms run is determined by **job configuration in Athena** or by the **scheduler in AthenaMT** (which looks at the declared data dependencies of Algorithms and AlgTools)
- AlgTools can be retrieved by Algorithms or other AlgTools, and are configurable, reusable framework components

Introduction (2)

In this presentation we will try to do two things:

- Show you how to write modern Athena Algorithms and Tools
- Understand a bit of the history (a lot of the code you look at will not look like this!)

Too much to cover explicitly, so we will try to give links to good (and bad) examples for you to look at later

The links will either be to our doxygen documentation or directly to gitlab (22.0.4 release):

- **Doxygen** is bit better formatted, and shows inherited methods (very useful!), but it struggles with the complexity of our code (e.g. sometimes links point to the wrong place)
- **Gitlab** is just our repository, but it shows you where the code actually lives, and also what edits have been done (can be useful to understand the history of a package)

Aside: AthenaBaseComps

When we talk about Algorithms, AlgTools and Services, we are talking about the Athena specific implementations of the Gaudi components, provided by the package AthenaBaseComps:

- i.e. **Ath(Reentrant)Algorithm**, **AthAlgTool**, **AthService**
  - You should **ONLY** inherit from these. Do **not** use Gaudi directly!

This package also provides messaging macros which avoid some performance pitfalls - please only use these (rather than the MsgStream directly):

```
ATH_MSG_VERBOSE("Only print this VERBOSE message if the job is configured for Verbose output")
```

(There is some more (slightly outdated) information on the twiki page [link])

Basic coding rules

In ATLAS, we use doxygen to document our code … please do the same (details on this twiki)

It is absolutely **forbidden** to use `cout` / `cerr` etc in production code - Gaudi provides a messaging service (and we provide convenient wrapper macros, as mentioned)

There must be absolutely no INFO level output during the event loop (if you want debugging information use `ATH_MSG_VERBOSE` or `ATH_MSG_DEBUG`)

Include the standard copyright statement as the first line of python and C++ files:

- `Copyright (C) 2002-2019 CERN for the benefit of the ATLAS collaboration`

More complete coding rules are available here

Basic coding rules (2)

You have heard a lot about threading principle, pthread, std::thread, TBB etc and might be tempted to use these in your algorithms….

DO NOT DO THIS!

‘We’ write algorithms and tools - **which must not be thread hostile** - and the scheduler takes care of threading for us

You can run some automated tests for thread safety by adding a file called `ATLAS_CHECK_THREAD_SAFETY` to root (or better, include directory if it exists) of a package.

- more here

ATLAS Package layout

A typical (modern) package will look like this:

```
/path/to/MyPackage/
```

- **`src/`** — **Mandatory**. Contains cxx (source) files and all **non-public** headers
- `MyPackage/` — Optional. Only create this if you have **public** headers (and only put these headers there)
- **`CMakeLists.txt`** — **Mandatory**. Configures the package. See this twiki for more.
- `python/` — Optional. Contains all python modules

(You might see more, like `test/` `share/` etc - we won't cover these today

Main methods of Algorithm

An algorithm is the main unit of work for Athena. Typically we have one algorithm of each type per job (but we equally can have many differently configured instances of the same algorithm)

When writing an Algorithm there are three main methods to override if needed:

- **`initialize()`** called once-per-job, before event processing starts
- Execute methods are called by the framework once per event.
  - **`execute()`** for Algorithms which inherit from `AthAlgorithm`,
  - **`execute(const EventContext& ctx)`** for those which inherit from `AthRentrantAlgorithm`
- **`finalize()`** - called once-per-job, when event processing stops

(There are more methods e.g. `configure()`, `sysStart()`, `stop()`, `terminate()` etc but they are not regularly used in Athena)

In addition, you **may** need to implement a constructor, and you **will** need to declare the alg to Athena (see later)

Helper methods of an Algorithm

An AthAlgorithm also provides some helper methods (which are less important these days, but which you might still see) e.g.:

- `evtStore()` - gives you access to the transient event store (which is cleared after every event)
- `detStore()` - gives you access to the 'detector store' (which lasts the entire job)
  - (These aren't typically needed any longer if you use handles … **and you should use handles**)

In very very very old code you might still see:

```
sc = service("StoreGateSvc",m_storeGate);
StatusCode sc =  m_storeGate->retrieve(object, m_objectLocation);
```

These instances should be fixed!

Reentrant algorithms

When running with >1 thread, the scheduler normally **clones** algorithms i.e. makes a different instance per thread.

However, with an `AthReentrantAlgorithm`we have **one instance** whose execute method is called **concurrently** on multiple threads.

In practical terms, this means that the `execute(...)` method must be const, and gets passed an `EventContext`

Being `const` means an `AthReentrantAlgorithm`must not alter its class members i.e. must not have any state (& this applies also to any tools which are called!)

Please write reentrant algos from now on! (And if you cannot because of legacy tools, fix them so you can!)

Declaring properties

We need to tell Gaudi about parameters of Algorithms (and AlgTools) that can be configured e.g.

- simple types (*int*, *float* …), containers (*std::string*, *std::vector*) etc

This has recently changed in Gaudi. Before we would:

- Create a property (e.g. a pT cut) in the header file
- Declare it in the constructor

```
std::vector<double> m_minRadius;   /** Converted photon reconstructed v
double              m_minPt;       /** Pt of the two participating trac
double              m_maxdR;       /** Distance of first track hit- rec
```

```
declareProperty("MinRadius",        m_minRadius   );
declareProperty("MinPt",            m_minPt       );
declareProperty("MaxdR",            m_maxdR       );
```

Now you can do it in one step in the header file:

```
118  /** @brief Tool to perform track matching*/
119  ToolHandle<IEMTrackMatchBuilder> m_trackMatchBuilder {this,
120      "TrackMatchBuilderTool", "EMTrackMatchBuilder",
121      "Tool that matches tracks to egammaRecs"};
```

Property name

Property variable

link

Declaring Properties (2)

In order to declare basic types this way, need to use Gaudi::Property e.g.

```
199  // @brief Maximum transverse energy to accept topo-seeded clusters
200  Gaudi::Property<float> m_maxEtTopo {this, "maxEtTopo", 8 * Gaudi::Units::GeV,
201      "Maximum transverse energy to accept topo-seeded clusters"};
202
203  // others:
204  Gaudi::Property<bool> m_dump {this, "Dump", false,
205      "Boolean to dump content of each object"};
```

Data access

In the past, we used to explicitly retrieve and record objects from StoreGate e.g.

```
ATH_CHECK( evtStore()->retrieve( taus, m_sgKey ) );
```

However for AthenaMT the scheduler needs to know about the data dependencies, so Algorithms (and Tools) MUST declare them using Read/Write Handles (which are fully backwards compatible with serial Athena)

Data access (2)

More on twiki but basically :

- Create keys for read/write locations
- Initialise in `initialize()`
- Create Handle from key inside `execute`
  - (For reentrant algorithms you also should pass the EventContext to the handles)

```
/** @brief Name of the topo-seeded cluster collection */
SG::ReadHandleKey<xAOD::CaloClusterContainer> m_topoSeededClusterContainerKey {this,
    "TopoSeededClusterContainerName", "EMTopoCluster430",
    "Input topo-seeded cluster container for egamma objects"};

/** @brief Name of egammaRec container */
SG::WriteHandleKey<EgammaRecContainer> m_egammaRecContainerKey {this,
    "egammaRecContainer", "egammaRecCollection",
    "Output container for egammaRec objects"};
```

```
ATH_CHECK(m_topoSeededClusterContainerKey.initialize(m_doTopoSeededPhotons));
ATH_CHECK(m_egammaRecContainerKey.initialize());
```

```
SG::WriteHandle<EgammaRecContainer> egammaRecs(m_egammaRecContainerKey);
ATH_CHECK(egammaRecs.record(std::make_unique<EgammaRecContainer>()));
```

Declaring the Components to Athena

Athena needs to do some extra work once you have written your algorithm or Tool e.g. creating the python configuration modules

So you need to let Athena know about your components:

- create `src/components/<package>_entries.cxx` which uses `DECLARE_COMPONENT` for each algorithm (and tool)
- make sure that your `CMakeLists.txt` includes `atlas_add_component` with `src/components/*.cxx`

```
#include "TrkRIO_OnTrackCreator/RIO_OnTrackCreator.h"
#include "../RIO_OnTrackErrorScalingCondAlg.h"
#include "../RIO_OnTrackErrorScalingDbOverrideCondAlg.h"

DECLARE_COMPONENT( Trk::RIO_OnTrackCreator )
DECLARE_COMPONENT( RIO_OnTrackErrorScalingCondAlg )
DECLARE_COMPONENT( RIO_OnTrackErrorScalingDbOverrideCondAlg )
```

link

```
# Component(s) in the package:
atlas_add_component( TrkRIO_OnTrackCreator
                     src/*.cxx
                     src/components/*.cxx
                     LINK_LIBRARIES AthenaBaseComps AthenaKernel EventPrimitives Gau
```

link

Example

```
class CaloClusterCellLinksUpdater
  : public ::AthReentrantAlgorithm
{

  ////////////////////////////////////////////////////////////////////
  // Public methods:
  ////////////////////////////////////////////////////////////////////
 public:

  /// Constructor with parameters:
  using AthReentrantAlgorithm::AthReentrantAlgorithm;

  /// Destructor:
  virtual ~CaloClusterCellLinksUpdater() = default;

  // Athena algorithm's Hooks
  /// Standard algorithm initialize
  virtual StatusCode  initialize() override;
  /// Standard algorithm execute
  virtual StatusCode  execute(const EventContext& ctx) const override;

 private:

  /// Name of the CaloCellContainer the links should point to(jobProperty)
  SG::ReadHandleKey<CaloCellContainer> m_caloCellContainerKey{this,
      "NewCaloCellName", "AODCellContainer", "Name of the CaloCellContainer the links should point to"};

  /// CaloClusters to be considered (jobProperty)
  SG::ReadHandleKeyArray<xAOD::CaloClusterContainer> m_clusterKeys{this,
      "CaloClusterNames", {}, "CaloClusters to be considered"};

};
```

https://gitlab.cern.ch/atlas/athena/blob/master/Calorimeter/CaloRec/src/CaloClusterCellLinksUpdater.h

AlgTools

Historically we have had two types of AlgTools, **public** (which means it is owned by the ToolSvc and one instance can be used by multiple clients) and **private**, which means it is owned by (and only used by) its parent, and is denoted by adding a 'this' to its construction

```
m_sctClusCor        ("InDet::SCT_ClusterOnTrackTool/SCT_ClusterOnTrackTool", this),
```

link to old-style (this class has public tools too)

```
ToolHandle<IEMTrackMatchBuilder> m_trackMatchBuilder {this,
    "TrackMatchBuilderTool", "EMTrackMatchBuilder",
    "Tool that matches tracks to egammaRecs"};
```

new-style link

Public tools can be problematic, as they can have state (either derived information, or pure event data), and they are conceptually similar to Services

So please, do not write new public tools!

AlgTools must also, like Algorithms, declare data dependencies (i.e. use Read/Write handles) - the scheduler will treat this as being a data dependency of the top level algorithm

Main methods of an AlgTool

When writing an Algorithm there are only two main methods to override:

`initialize()` - called once-per-job, before event processing starts

`finalize()` - called once-per-job, when event processing stops

In order to do anything useful, you will also need to implement some methods to do useful work. These are not defined by the framework, but by you

Typically an **AlgTool** will have an **IAlgTool** interface class, which specifies exactly what these types of Tools look like (and also removes the need to have a direct dependency on the specific tools)

Interfaces and AlgTools

We can have several implementations of the same interface

For example, the IRIO_OnTrackCreator interface declares the following pure virtual function:

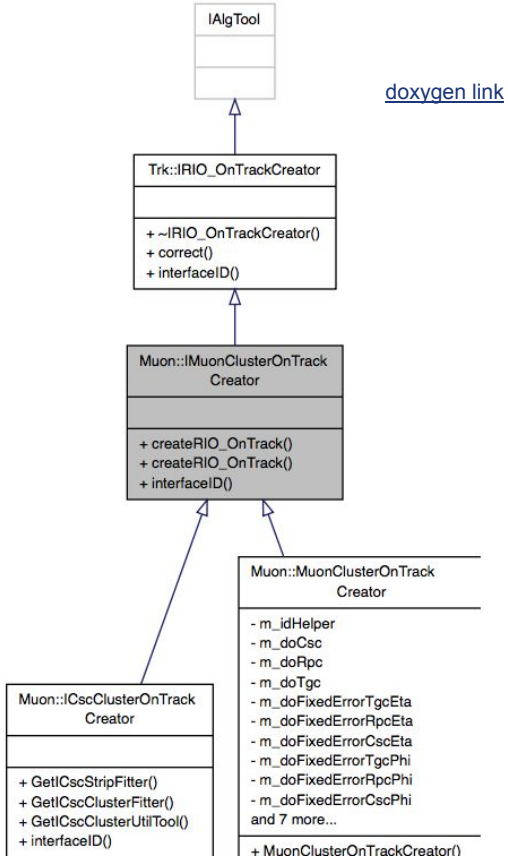
```
virtual const RIO_OnTrack*  correct (const PrepRawData& hit, const TrackParameters& trk)const =0;
```

And other tools (e.g. MuonClusterOnTrackCreator) implement it. A client can then own a ToolHandle to this interface:

```
ToolHandle< Trk::IRIO_OnTrackCreator >
m_ROTcreator{this,"ROTcreatorTool","Trk::IRIO_OnTrackCreator/FTK_ROTcreatorTool","Tool to
create RIO On Track"};
```

And be configured (in python) to use any concrete implementation

```
myTool.ROTcreatorTool = mySpecialROTCreatorInstance
```



doxygen link

Services

Not likely to be something that most of you will deal with, BUT in order to get rid of public tools, some code might need to be ported to services instead

Must be thread safe!

- can be accessed concurrently by clients in different threads from the **same** event
- can be accessed concurrently by clients in different threads from the **different** events

More details are available on this twiki

Conclusion

Hopefully you have now learnt a bit about how to write Algorithms and Tools.

As always, the best way to learn is by trying … which brings us to the practical. Information is available here