ATLAS's Offline Software Framework(s)

Attila Krasznahorkay

...on behalf of all of the ATLAS software developers

Plural? 😕

- In this presentation I will be mostly talking about Athena, ATLAS's software framework for running simulation/digitisation/reconstruction
  - However, due to our data analysis model, we perform a lot of non-trivial corrections/calibrations as part of the analysis workflow. Most people do not use Athena for that, but a much more lightweight environment that we also provide centrally.
  - This setup, where physics analysers see a much simpler software infrastructure, has served us reasonably well over the years. But of course it has its own dark side as well…

Gaudi / Athena

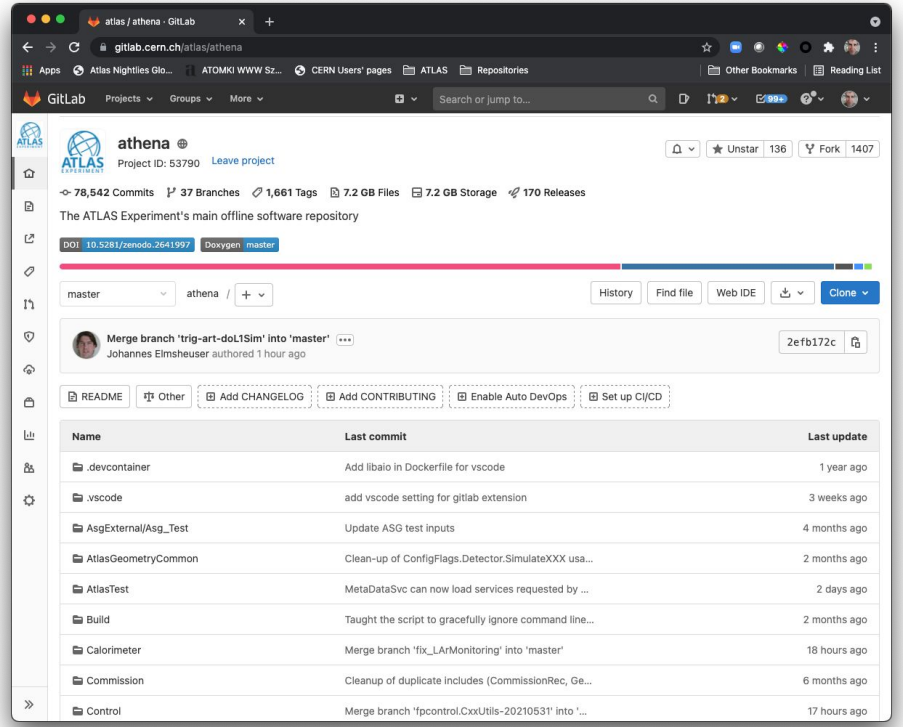
- We share a single “framework core” with LHCb: Gaudi
  - https://gitlab.cern.ch/gaudi/Gaudi
- Athena is ATLAS’s framework built on top of this common core
  - Actually “Athena” can mean a few different things for us…
    - The git repository where we store all of the code (https://gitlab.cern.ch/atlas/athena)
    - The collection of core framework “packages” that are responsible for the basic infrastructure of the framework (mostly held under: https://gitlab.cern.ch/atlas/athena/-/tree/master/Control)
    - The “main project” that builds almost everything from the repository (https://gitlab.cern.ch/atlas/athena/-/tree/master/Projects/Athena)
  - I’ll be highlighting some of its relevant features in the following

Athena Highlights

- (Optionally) Multi-process / multi-threaded application, started by an elaborate BASH/python script (athena.py)
- Jobs are configured through python scripts, which then determine what libraries to load, what objects to instantiate, etc.
  - Python allowing us to use elaborate code with branching/loops/etc. already for just configuring our jobs
- Multi-process jobs can be used to save on memory through Linux's Copy-on-Write behaviour
  - Was used very heavily during LHC's Run-2 so that we could fit into our LHC grid resources
- Multi-threaded jobs are orchestrated using TBB, using an elaborate set of data-flow and control-flow rules set up between the framework components
  - With Interval of Validity (IoV) transitions not requiring any explicit synchronisation points during the event processing

Code Organisation

- The code is managed through a single “mega repository”, which can be built into different projects
  - Smaller projects like AthSimulation, AthGeneration, etc. build just a subset of the code
- The build is managed using CMake, with a custom organisation that uses the concept of “packages” as project building blocks
  - With “projects” mainly differing in which “packages” they build together



Components

- Every “component” of the framework is initialised and finalised in a job very similar to each other, with only minor differences.
  - The main difference between components is what they do during the event processing.
- The framework defines the following types of components:
  - **Algorithms:** Classes that have an “execute function”, which is called by the framework once for every event
  - **Services:** Thread-safe components that are instantiated in a small number of global instances, and used by possibly many clients through their public (pure virtual) interface
  - **Tools:** Components associated to specific algorithms/services, providing some functionality through their public (pure virtual) interface
    - Reality is a bit more messy for tools, but that’s not for this presentation…
  - **Converters:** Components that we like less and less design-wise, and have been moving away from for some time 😛
- All components can declare “properties” on themselves, which in turn can be set/adjusted from the Python configuration of a job

Data Management

- All data goes through global object stores
  - While services and tools can receive data objects through their custom interfaces, algorithms can only exchange data between each other through these data stores
- A single service (StoreGateSvc) is instantiated into a few objects (for event-/conditions-/meta-data management)
  - These services provide some generic functions for saving/retrieving objects and querying the contents of the service
  - Direct access to the service is not recommended anymore though, as you will see in the next slide
- A combination of services/converters/algorithms are used to read/write “ByteStream” and ROOT files with objects to/from the data store(s)
  - The framework would allow support for any other file type as well, but in practice we only ever used these two

Data Flow / Control Flow

- The framework only “executes” algorithms
  - Services/tools/converters are “only” there to serve algorithms (at least to the first degree…)
- Data dependencies between algorithms are expressed through special properties
  - Declaring the C++ type and string identifier of all inputs and outputs of an algorithm. The string keys of which can be adjusted at configuration time.
  - Special objects are then used to retrieve/record data objects using these property objects. Without accessing StoreGateSvc directly.
- Special algorithms (“sequences”) can be used to group other algorithms together
  - “Control flow” rules can then be set up to establish relationships between such sequences
- A combination of data-flow and control-flow rules are used to set up the non-trivial relationships between components in our High Level Trigger as well
- The algorithm scheduler also has basic support for “blocking” (GPU…) algorithms, but work on that will still need to happen

An Algorithm Example

```
22  namespace xAODMaker {
23
24     /**
25      *  @short Algorithm for creating xAOD::EventInfo objects for the xAOD
26      *
27      *         This algorithm can be used to translate an EventInfo object
28      *         from an input (AOD) POOL file into an xAOD::EventInfo object.
29      *
30      * @author Attila Krasznahorkay <Attila.Krasznahorkay@cern.ch>
31      */
32     class EventInfoCnvAlg : public AthReentrantAlgorithm {
33
34     public:
35        /// Regular Algorithm constructor
36        EventInfoCnvAlg( const std::string& name, ISvcLocator* svcLoc );
37
38        /// Function initialising the algorithm
39        virtual StatusCode initialize() override;
40        /// Function executing the algorithm
41        virtual StatusCode execute (const EventContext& ctx) const override;
42
43     private:
44        /// Key for the input object
45        /// If blank, we do a keyless retrieve from SG instead!
46        SG::ReadHandleKey<EventInfo> m_aodKey;
47        /// Key for the output object
48        SG::WriteHandleKey<xAOD::EventInfo> m_xaodKey;
49
50        /// For pileup.
51        SG::WriteHandleKey<xAOD::EventInfoContainer> m_pileupKey;
52
53        /// Handle to the converter tool
54        ToolHandle< IEventInfoCnvTool > m_cnvTool;
55
56     }; // class EventInfoCnvAlg
57
58  } // namespace xAODMaker
```

- Putting this all together, a simple “reentrant” algorithm would look something like this
  - Declaring its input data dependencies and output data products, and setting up access to a tool through a pure virtual interface
- As always, things can get more complicated very quickly…

A Job Example

- Something like a full-blown reconstruction job becomes much more complicated of course
  - Requiring the setup of hundreds of algorithms and services, and thousands of tools
- Leading to the necessity of using some non-trivial Python code for setting up such a thing…

```
# Set up the reading of a file:
FNAME = "AOD.pool.root"
import AthenaPoolCnvSvc.ReadAthenaPool
ServiceMgr.EventSelector.InputCollections = [ FNAME ]

# Access the algorithm sequence:
from AthenaCommon.AlgSequence import AlgSequence
theJob = AlgSequence()

# Create a POOL output file with the StoreGate contents:
from OutputStreamAthenaPool.MultipleStreamManager import MSMgr
xaodStream = MSMgr.NewPoolRootStream( "StreamXAOD", "xAOD.pool.root" )

# Create the xAOD EventInfo object:
from xAODEventInfoCnv.xAODEventInfoCreator import xAODEventInfoCreator
xAODEventInfoCreator( outkey = "EventInfoTest" )

# Check what happened to the stream:
xaodStream.Print()

# Do some additional tweaking:
from AthenaCommon.AppMgr import theApp
theApp.EvtMax = 10
ServiceMgr.MessageSvc.OutputLevel = INFO
ServiceMgr.MessageSvc.defaultLimit = 1000000
```

“Production Transforms”

- The athena.py script is great for running jobs by hand, debugging them, etc.
  - However it doesn’t easily allow us to schedule jobs in our production system without teaching the production system how to write Python configuration files itself
- For this reason, an additional layer was developed, what we call the production transforms
  - These allow us to launch complicated jobs in a way that’s more friendly towards our production system

```
export ATHENA_CORE_NUMBER=8
Reco_tf.py --AMI=q431 --multithreaded=True --steering='doRAWtoALL' \
  --preExec 'all:DQMonFlags.doMonitoring=True; DQMonFlags.doNewMonitoring=True' \
  --imf=False --maxEvents=500 --outputAODFile=myAOD.pool.root \
  --outputHISTFile=myHIST.root
```

Not an actually functional command 😛

EventLoop - The ATLAS Analysis Framework

- Most analysis code uses a much lighter-weight framework to run calibrations / corrections in
  - https://gitlab.cern.ch/atlas/athena/-/tree/master/PhysicsAnalysis/D3PDTools/EventLoop
  - https://gitlab.cern.ch/atlas/athena/-/tree/master/PhysicsAnalysis/D3PDTools/AnaAlgorithm
- Because it's much more lightweight, it can be compiled on practically any POSIX platform
  - While Athena is tied to whatever platform LCG releases are available on
- We've set up all "analysis tools" for LHC's Run-2 to be usable in both EventLoop, and Athena (what we called "dual use" tools 😉)
  - And they have been! Both in analysis, and in the HLT and offline reconstruction.
- For Run-3 we have introduced "dual use" algorithms as well, which will hopefully be equally successful

Summary

- Our code organisation has served us very well
  - Especially how we share non-trivial pieces of code between different projects that are used separately for the HLT, reconstruction, simulation and analysis
- Gaudi/Athena is under constant development to allow us to make use of many-core (>32) CPUs and non-CPU accelerators in the future
  - With R&D going on for multi-node job executions as well

http://home.cern