Brookhaven™
National Laboratory

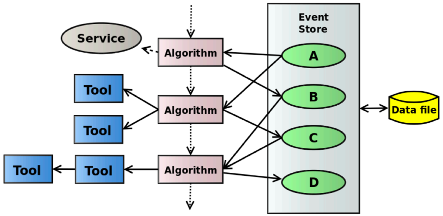
ATLAS Athena configuration

Johannes Elmsheuser

19 July 2023, ePIC software meeting

With material from Tadej Novak (DESY), Walter Lampl (Arizona), Scott Snyder (BNL), Vakho Tsulaia (LBNL)

Athena framework



- Athena is CERN ATLAS experiment Simulation, Reconstruction and partially Data Analysis software framework
- Originally designed more than 20 years ago for sequential execution

- LHC Run 1 (2010-2012): production jobs in multi-job mode
    - Run several independent instances of serial Athena on a compute node to utilize multiple CPU cores
- LHC Run 2 (2015-2018): switch to to multi-process AthenaMP
    - Initialize single process, then fork multiple sub-processes
    - Each sub-process still running serial Athena
- LHC Run 3 (2022-2025): transitioned to multithreaded AthenaMT
    - A major migration effort affecting practically all domains of the ATLAS software

Gaudi and Athena

- Athena is implemented on top of the **Gaudi** framework
- Gaudi contains components and interfaces for building event data processing frameworks for HEP experiments
    - Used by several experiments (e.g. ATLAS, LHCb), as well as the Future Circular Collider
- Gaudi implements a Component Model
    - Components implement an interface, and use other components through an interface
    - Components get compiled into Shared Object Libraries (DSO)
    - A dedicated Gaudi Plugin Service locates at run time which DSO contains requested component, loads the library, and creates an instance of the component in memory
- Gaudi components are implemented in C++
- For several key Gaudi components Athena implements Athena-specific extensions

Athena job configuration

- The ATLAS code contains thousands of Gaudi Components:
    - Algorithms, Tools, Services, Converters, ...
    - They are put together at run-time.
- To have a meaningful Athena job, one needs to define a meaningful set of components, their properties and their relationships.
- This is what we call **Athena job configuration**

Python as the configuration language

- A long time ago ATLAS chose to use **Python** as the configuration language.
    - That implies that the configuration step is itself a ”program” and can get arbitrarily complicated.
- For all components create Python equivalents at compile-time.
    - Python classes with the same name and properties as the C++ component.
    - Done automatically by the build system.
- Our configuration organically grew throughout the lifetime of Athena.
    - Many different iterations
    - Over the past 2 years moved all main workflows to a common, modern configuration infrastructure called **ComponentAccumulator** configuration system

Athena configuration system

The Athena configuration system in broad terms the configuration system works as follows:

1. Each C++ component defines a set of configurable parameters (*),
    - There is rich, yet restricted set of types of configurable parameters that are supported,
    - Typically reasonable default values are defined as well.
2. During compilation a database entry is made containing information about these properties.
3. During the configuration this database is queried for information about a component and as a result Python class is generated with class attributes corresponding to the properties defined in C++.
4. Set of Python script creates Python objects and manipulates them (set properties). (*)
5. Resulting set of Python objects produce a serialized/textual representation of the configuration.
6. The configuration is read in C++ program and populates the global dictionary with all settings.
7. C++ components during the initialization fetch the values and set configurable parameters (see 1st point) accordingly.

The steps marked with (*) are places where developer intervention is needed. That is to define configurable parameters and to prepare scripts setting them. Other steps are automatized.

One slide summary on how Athena configuration should look like

- No global namespace, no ”include”, proper Python.
- Information flow should be obvious and not hidden.
- Configuration fragments are standalone run-able and mergeable to yield larger configuration fragment.
    - A fragment should configure all the components it needs to work.
    - Implies a dedicated de-duplication step to drop components that are set up by multiple fragments.
    - Standalone run-able only if the fragment contains at least one event-algorithm and the input is readable from a file.
    - Often, a minimal wrapper to open the input file is necessary in unit tests.
- Modularity is still desirable:
    - Each service, tool or algorithm has a piece of Python code associated to configure it.
    - There might be cases where the autogenerated one is just enough, but this hides the dependency chain.

Types of configuration settings

1. Frequently changed settings (e.g. input file location)
2. Settings affecting multiple components (e.g. if the input file is MC or data).
3. Settings of main components (e.g. number of threads)
4. Settings specific to a single component changed only during debugging and in very special situations (a threshold for a single detector readout element)

- Settings 1-3 are treated as main Athena settings are called **flags**.
- They are defined in AthenaConfiguration.AllConfigFlags and are further categorised into domains like Input, Global, Calo, …

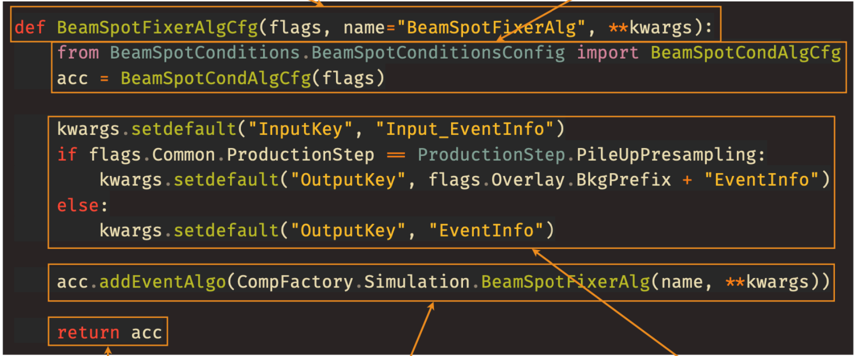
Component Accumulator

- Contains lists of:
    - Event algorithms per sequence,
    - Conditions algorithms (a special case of the above), services
    - Public tools (the use public tools is discouraged),
    - A handle to pass private tools on to the caller, parameters to be assigned to the AppMgr.
- The payload of these lists are auto-generated Python Configurables.
- It knows how to merge itself with another ComponentAccumulator instance.
- Has getter methods to access individual components.
- It is the return value of the methods that configure pieces of the job. May contain only one algorithm or the entire job!
- Supports running the job fully from a Python script.

How to write an configuration fragment: Algorithms

1. Define the fragment function
(first flags, then usually name and keyword arguments)

2. Initial ComponentAccumulator
(can be a new one or an existing one)



```python
def BeamSpotFixerAlgCfg(flags, name="BeamSpotFixerAlg", **kwargs):
    from BeamSpotConditions.BeamSpotConditionsConfig import BeamSpotCondAlgCfg
    acc = BeamSpotCondAlgCfg(flags)

    kwargs.setdefault("InputKey", "Input_EventInfo")
    if flags.Common.ProductionStep == ProductionStep.PileUpPresampling:
        kwargs.setdefault("OutputKey", flags.Overlay.BkgPrefix + "EventInfo")
    else:
        kwargs.setdefault("OutputKey", "EventInfo")

    acc.addEventAlgo(CompFactory.Simulation.BeamSpotFixerAlg(name, **kwargs))

    return acc
```

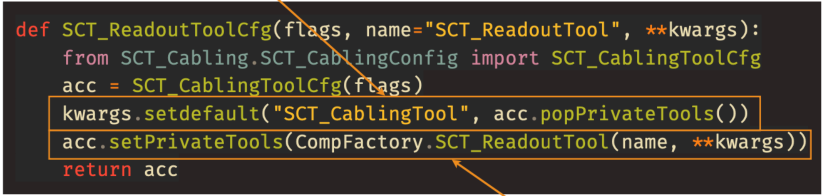
5. Return!

4. Initialise and add to the CA
(can be initialised in-place, no aliasing needed)

3. Define properties
(usually using keyword arguments)

How to write an configuration fragment: Tools

Imported private tools need to be popped
(can also use popToolsAndMerge if not using the base CA)



```
def SCT_ReadoutToolCfg(flags, name="SCT_ReadoutTool", **kwargs):
    from SCT_Cabling.SCT_CablingConfig import SCT_CablingToolCfg
    acc = SCT_CablingToolCfg(flags)
    kwargs.setdefault("SCT_CablingTool", acc.popPrivateTools())
    acc.setPrivateTools(CompFactory.SCT_ReadoutTool(name, **kwargs))
    return acc
```

Add a new private tool

Deduplication

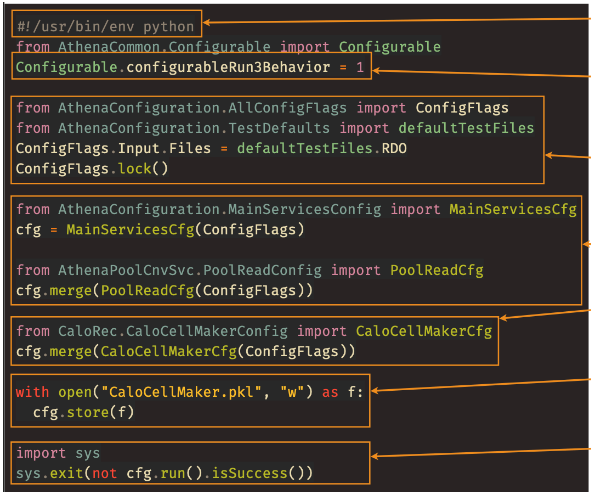
Some components are needed by more than one algorithm (e.g. GeoModelSvc or IOVDbSvc).

- Each of the self-consistent configuration fragments has to declare them, so merging sees multiple instances. $\rightarrow$ That's ok! Only a problem if the same component is requested with conflicting configuration.
- The legacy configuration scheme considers any duplication an error.

Strategy:

- If two components compare equal: Ignore the second instance.
- If they have a different name and have different configuration, they are probably meant to be different and keep both instances.
- Tricky case: same name but different configuration.
    - List-properties can have custom methods to unify (merge, concatenate, …) them.
    - Examples: IOVDbSvc.Folders, GeoModelSvc.DetectorTools.
    - All other cases are consider this a name-clash: raise an exception.

How to run a minimal jobs/script



```
#!/usr/bin/env python
from AthenaCommon.Configurable import Configurable
Configurable.configurableRun3Behavior = 1

from AthenaConfiguration.AllConfigFlags import ConfigFlags
from AthenaConfiguration.TestDefaults import defaultTestFiles
ConfigFlags.Input.Files = defaultTestFiles.RDO
ConfigFlags.lock()

from AthenaConfiguration.MainServicesConfig import MainServicesCfg
cfg = MainServicesCfg(ConfigFlags)

from AthenaPoolCnvSvc.PoolReadConfig import PoolReadCfg
cfg.merge(PoolReadCfg(ConfigFlags))

from CaloRec.CaloCellMakerConfig import CaloCellMakerCfg
cfg.merge(CaloCellMakerCfg(ConfigFlags))

with open("CaloCellMaker.pkl", "w") as f:
  cfg.store(f)

import sys
sys.exit(not cfg.run().isSuccess())
```

1. Declare to use python

2. Retrieve configurables in the right way

3. Setup flags
(Many are autoconfigured based on input. Do not forget to lock!)

4. Setup main services
(input is also not a direct dependency)

5. Setup algorithm(s)

Optional: Dump the pkl
(can be used to persistify and/or compare configurations)

6. Run
(needs to exit with the right code)

Steering Production Jobs

- Productions jobs used in PanDA are steered using transforms (e.g. Reco_tf.py, Sim_tf.py).
    - In production configured from AMI (ATLAS Metadata Interface).
    - AMI config can be used locally: Reco_tf.py –AMIConfig q442
- Often want to inject some configuration at the start of the job (i.e change some flags) or at the end of the job (i.e. add or modify algorithms).
    - preExec/preInclude: executed after flags ”autoconfiguration” but before ConfigFlags.lock(), the goal is to override specific flags with non-default values
    - postExec/postInclude: execute arbitrary code on the final CA or merge additional ones
- In production there should be no pre/postExecs/Includes (with some exceptions like campaign steering).

Technical details of steering production jobs - Rucio datasets

- Example of a production job command line (missing here `--AMI r14622`):

```
Overlay_tf.py \
  --CA \
  --inputHITSFile ${HITS_File} \
  --inputRDO_BKGFile ${RDO_BKG_File} \
  --outputRDOFile ${RDO_File} \
  --maxEvents $events \
  --conditionsTag OFLCOND-MC16-SDR-RUN2-09  \
  --geometryVersion ATLAS-R2-2016-01-00-01 \
  --preInclude 'all:Campaigns.MC20e' \
  --postInclude 'OverlayConfiguration.OverlayTestHelpers.OverlayJobOptsDumperCfg' \
  --postExec 'with open("ConfigOverlay.pkl", "wb") as f: cfg.store(f)'
```

- Rucio dataset example (AMI tags encoded in dataset name):

```
mc23_13p6TeV:mc23_13p6TeV.601237.PhPy8EG_A14_ttbar_hdamp258p75_allhad.deriv.DAOD_PHYS.
    e8514_e8528_s4111_s4114_r14622_r14663_p5737
```

Summary and Material

- Provided a very brief overview of ATLAS Athena job configuration based on material of the ATLAS software developer tutorial
- ATLAS software documentation is open at https://atlassoftwaredocs.web.cern.ch/
- Some links:
    - https://atlassoftwaredocs.web.cern.ch/guides/ca_configuration/
    - https://gaudi-framework.readthedocs.io/en/latest/
    - https://gitlab.cern.ch/atlas/athena
    - https://gitlab.cern.ch/gaudi/Gaudi