EPJ Web of Conferences **214**, 05015 (2019) https://doi.org/10.1051/epjconf/201921405015
CHEP 2018

A new approach for ATLAS Athena job configuration

Walter Lampl[1,*] on behalf of the ATLAS Collaboration

[1]University of Arizona, 1118 E 4th Street, Tucson AZ, 85721

Abstract. The offline software framework of the ATLAS experiment (Athena) consists of many small components of various types like Algorithm, Tool or Service. To assemble these components into an executable application for event processing, a dedicated configuration step is necessary. The configuration of a particular job depends on the work-flow (simulation, reconstruction, high-level trigger, overlay, calibration, analysis ...) and the input data (real or simulated data, beam-energy, ...).
The configuration step is done by executing python code. The resulting configuration depends on optionally pre-set flags as well as meta-data about the data to be processed. For the python configuration code, there is almost no structure enforced, leaving the full power of python to the user.
While this approach did work, it also proved to be error prone and complicated to use. It also leads to jobs containing more components that they actually need. For LHC Run 3 a more robust system is envisioned. It is still based on python but enforces a structure and emphasis modularity. This contribution briefly reports about the configuration system used during LHC Run 1 and Run 2 and details the prototype of an improved system to be used in Run 3 and beyond.

1 Introduction

The software framework of the ATLAS Experiment [1], Athena [2], consists of thousands of C++-written components. They are assembled at run-time into executable jobs. The individual components can define properties (variables) that can be set at run-time. This component model allows for a high degree of flexibility. The Athena framework is used in the high-level trigger, reconstruction, simulation, digitzation and to some extend also for analysis jobs. This article is about configuring Athena by defining a set of components, their properties and the relationship between then.

1.1 Types of components

The Athena framework knows the following types of components:

Algorithms are executed once per event.

Services are singletons (exactly one instance per job). Other framework components can call methods of Services. Examples include the white-board store that is used to pass data from one algorithm to the next (`StoreGateSvc`).

*e-mail: walter.lampl@cern.ch

AlgTools are used to factorize work of Service or Algorithms. So far, they can be either private to one Algorithm or Service or public, shared by several Algorithms or Services. With the online migration to a multi-threaded version of Athena [3], public AlgTools are deprecated.

At the time of writing, the standard real-data reconstruction uses about 750 public tools, 88 services and 198 event processing algorithms. Each sub-detector group of ATLAS writes the components necessary to reconstruct data from this sub-detector and to simulate the sub-detector. The same applies to areas like tracking and jet-reconstruction. The configuration of these components is also jointly done by many people. Therefore modularity is important.

2 A bit of history

In the early years of Athena, only a text-file based job-configuration system was used. This system had almost no programming-language capabilities, it was largely declarative. In 2003, ATLAS decided that such a system was too limiting and switched to python for job configuration. To ease the migration from the text-based configuration to python, some concepts were preserved, in particular the option to *include* other files. In the python world, that means executing them in the same name-space as the caller. This leads to an effectively global name-space where name-clashes are inevitable. With the full power of the python available, developers used that power to create overly complicated configuration "programs". Over the years, the configuration part of Athena grew into hundreds of thousands lines of hard-to-maintain python code.

2.1 Auto-Configuration

Auto-Configuration is a concept that was devised during the commissioning period of ATLAS, where the detector configuration was frequently changing. The configuration automatically adapts itself to data-taking circumstances. In practice, this means that during the configuration step, the input file is opened and various data-bases are contacted to determine for example if the magnets of the detector had been on or off when the data was taken or if the input file is real data or simulated data. While this feature was initially meant as a stopgap solution for the commissioning period it turned out to be very useful and is therefore maintained. It comes with a small performance penalty because of additional data-base traffic and the fact that the input file is opened and closed multiple times.

3 Basic principles of the new configuration system

The new configuration system described here avoids the global namespace and imposes more structure on the configuration code.

The configuration of pieces of a job is done by python methods (called configuration method) that take flags as parameter and return configured algorithms, tools, etc. The result should be self-consistent in the sense that it contains all auxiliary components (services) that are needed for a functional job. That implies that each configuration method yields an independently run-able set of components, as long as it contains at least one event-processing algorithm and the input of this algorithm can be read from an input file.

The result of these configuration methods can be merged together to obtain bigger jobs that do more work. This way the configuration of a full reconstruction job can be built from smaller units.

Since many components (for example basic services) are used by more than one algorithm, they will be declared multiple times if two self-consistent configurations are merged. To deal with this situation, a dedicated de-duplication step is required.

4 Elements of the new configuration system

The new configuration system uses the same python representation of Athena components that have been used for many years, with minor modifications. To store configurations, a new python class called `ComponentAccumulator` is introduced. It can hold the configuration of an entire job or a single component. This class has a merge method that can unify two instances of `ComponentAccumulator`, applying de-duplication if necessary.

4.1 De-Duplication

When two instances of `ComponentAccumulator` are unified or when more components are added to a `ComponentAccumulator`, de-duplication avoids multiple components with the same name. The following cases can be distinguished:

- If two components have the same type, same name and all their properties are identical, the second instance is silently dropped. The second component may have been produced by executing the same configuration method twice with the same parameter. This is not an error but rather a unavoidable side-effect of self-consistent configuration sets.
- If two components have the same type, but a different name we assume that they are meant to be different. Both instances are kept.
- If two components share the same name, same type but (some) of their properties are different one has to evaluate the property.
  - Properties that are lists of some kind and are explicitly white-listed as merge-able are merged. This covers cases like the `Folders` property of `IOVDbSvc` which is the list of conditions data-base folders that a job is supposed to read. Each configuration fragment that requires conditions data sets up data-base access and declares the folders it needs.
  - All other cases are considered a name-clash and will result in an error.

The de-duplication needs to be applied recursively to private tools and their properties.

4.2 Configuration flags

Configuration flags are used to steer the configuration process, like turning off a particular feature or setting properties of various components consistently. Some of the flags are set via the auto-configuration mechanism explained in section 2. The configuration flags used up to now live in a global name-space and can be altered in the course of the configuration process. That may lead to inconsistent configurations. Moreover, we observe a proliferation of flags: more than 3000 job configuration flags have been defined.

The new version of configuration flags has their inter-dependence and a locking mechanism built in. Each flag is defined with either a default value or a function that sets the default value based on other, previously defined flags. These functions are in most cases very simple and can be implemented as python lambda functions. These default-setter are invoked if a flag that has not been set explicitly is asked for its value. This works recursively: if the default-setter of flag A asks flag B for it's value, the default setter of flag B gets invoked.

The auto-configuration becomes part of the functions that set the default values, which may include opening files and connecting to databases. The following chain is typical: a particular correction tool applies only to real data, not to simulated data. There is a dedicated flag `SubDet.doMyCorr` that forces the correction on or off. The configuration asks for the value of this flag, but in general its value was not explicitly set. So the default-setter is invoked that queries the flag `input.isMC`. Since that depends on the input file, the default setter of `input.isMC` invokes as function that opens the input files (the path to the input file is itself a flag) to determines if it contains real or simulated data.[1] After that happened, the values of `input.isMC` and `SubDet.doMyCorr` are set. Subsequent calls do not trigger the default-setter.

The second important feature is that flags can be set only before they get used and can not be updated multiple times in the course of the configuration process. All possible configuration flags are defined (either directly or by importing) in one place (called `AllConfigFlags.py`). We expect we can do with much fewer than the 3000 flags we have now. The flag-container is the parameter to the configuration methods. When a configuration method calls another configuration method, the flag-container is passed on. This way the flag-container is passed through the call-stack from the main configuration method down to the small methods configuring just one component.

5 Integration

The main-section of a python configuration file typically first imports the flag-container from `AllConfigFlags.py`, sets values as appropriate (at least the input file) and locks it. Then it calls a configuration method using the flag-container as parameter. In fact, most of the new-style configuration python files have a short main-section that contains a few lines of boilerplate code to access data from an input file and then calls the configuration-method from the same file. This can serve as a unit-test of the algorithm in question. As an example, the main-section of the configuration file of the algorithm producing topological calorimeter clusters is shown in listing 1.

```
if __name__=="__main__":
    from AthenaConfiguration.AllConfigFlags import ConfigFlags
    ConfigFlags.Input.Files = ["myESD.pool.root"]
    ConfigFlags.lock()

    from AthenaConfiguration.MainServicesConfig import MainServicesSerialCfg
    from AthenaPoolCnvSvc.PoolReadConfig import PoolReadCfg
    cfg=MainServicesSerialCfg()
    cfg.merge(PoolReadCfg(ConfigFlags))

    topoAcc=CaloTopoClusterCfg(ConfigFlags)
    cfg.merge(topoAcc)

    cfg.run()
```

Listing 1. Example of a basic top-level configuration method

The flag-container is passed down the call-chain, as a parameter of the configuration methods. The instances of `ComponentAccumulator` they return get merged and returned up the call-chain to the top-level method. Along the way the duplicate components are eliminated or reconciled as explained in section 4. This is illustrated in figure 1.

[1] For performance reasons, the opening of the input files happens only once and the relevant meta-data is cached.

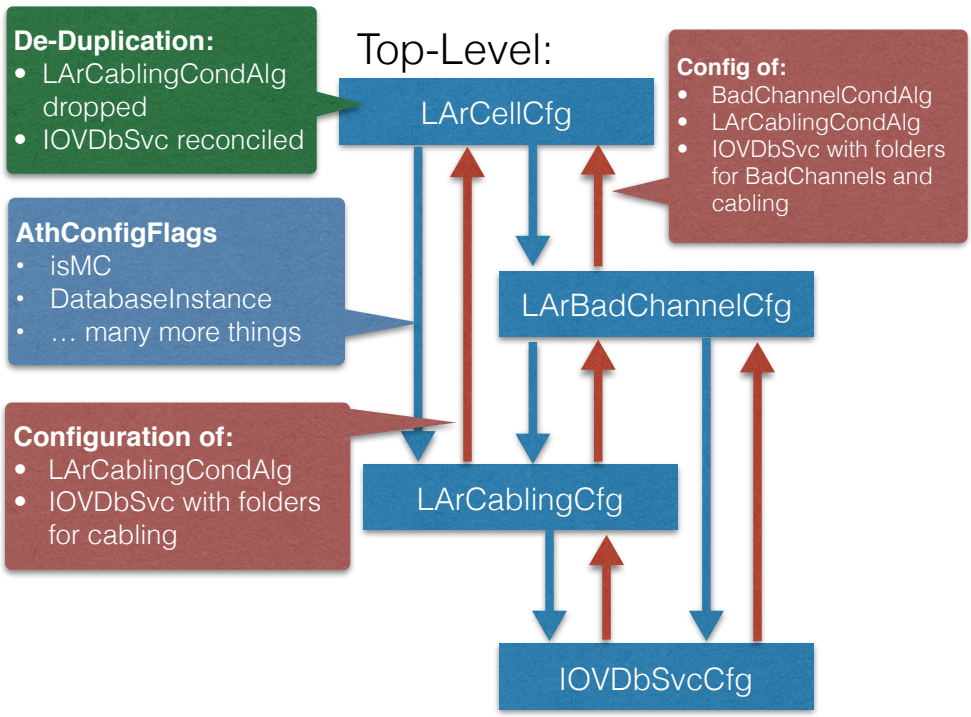

Figure 1. Example of a configuration work-flow. Blue arrows denote the container of flags passed down from caller to callee. Red arrows are instances of ComponentAccumulator returned by the configuration methods.

6 Current status and plans

The basic infrastructure of the new configuration system has been developed. The configuration of some of the basic services of Athena (like data-base access, detector geometry, input file reading) have been implemented using the `ComponentAccumulator`-approach. A number of demonstrators that configure a few event-processing algorithms with the tools and service they need exist as well. ATLAS plans to use the configuration system described here from run 3 onwards.

7 Conclusions

ATLAS has started to redesign the configuration system of its offline software. The new system aims to be more maintainable and easier to understand.

References

[1] The ATLAS Collaboration, Journal of Instrumentation **3** S08003 (2008)
[2] Calafiura P, Lavrijsen W, Leggett C, Marino M and Quarrie D, The Athena control framework in production, new developments and lessons learned, CHEP 2004 Conf. Proc. **C04-09-27** pp 456-458 (2005)
[3] Charles Leggett et al, J. Phys.: Conf. Ser. **898 042009** (2017)