Physics-level Job Configuration

W. Liebig, CERN, Geneva, Switzerland; D. Rousseau, LAL, Orsay, France;
P. Calafiura, W. Lavrijsen,[*] LBNL, Berkeley, USA;
P. Loch, University of Arizona, Tucson, USA; A. Salzburger, University of Innsbruck, Austria

Abstract

The offline and high-level trigger software for the Atlas experiment has now fully migrated to a scheme which allows large tasks to be broken down into many functionally independent components. These components can focus, for example, on conditions or physics data access, on purely mathematical or combinatorial algorithms or on providing detector-specific geometry and calibration information. In addition to other advantages, the software components can be heavily re-used at different levels (subdetector tasks, event reconstruction, physics analysis) and on different running conditions (LHC data, trigger regions, cosmics data) with only little adaptations. A default setting therefore has to be provided for each component allowing these adaptations to be made. End-user jobs contain many of these small components, most of which the end-user is totally unaware. There is therefore a big semantic discrepancy between how the end-user thinks about a specific job's configuration and how the configuration is packaged with the individual components making up the job. This paper presents a partly automated system which allows component developers and aggregators to build a configuration ranging over all the above levels, such that e.g. component developers can use a low-level configuration, sub-detector coordinates work with functional sequences and the end user can think in physics processes. This system of python-based job configurations is flexible but easy to keep internally consistent and avoids possible clashes when a component is re-used in a different context. The paper also presents a working system used to configure the new Atlas track reconstruction software.

ATLAS SOFTWARE CONFIGURATION

The offline and high-level trigger software for the Atlas experiment use the ATHENA/GAUDI component architecture [1], which has allowed for the development of a large selection of functionally independent components. There are three main kinds of components as far as the framework is concerned: *Algorithms*, which implement a particular strategy to process event data, for example "find all jets in the calorimeter." *Services*, which provide facilities to Algorithms in performing their tasks, for example "write this object to file." And *AlgTools*, which are close in purpose to Services, but can be made private to a specific Algorithm. Functionally, however, there are as many kinds of components as there are tasks to perform.

[*] WLavrijsen@lbl.gov

All users of the framework, from physicists to component developers, utilize a python [2] user interface to assemble an ATHENA job by choosing from hundreds of components, defining their intercomponent relations, and setting their attributes (referred to as *properties* in GAUDI). Properties are a run-time construct, are developer-defined, and are generally not guaranteed by the interfaces that a component implements.

Athena Job Configuration

Components typically come with two kinds of configuration: the defaults as set in the source code of the component and compiled into it; and predefined, self-consistent *job options* fragments, written in python, that are bundled in a package with the component. Note that it isn't always possible to set proper defaults in the compiled code, without knowledge of the context in which the component is going to be used. In those cases, usage of the component is always done through selection of a context-specific job options fragment, and it is up to the user to find and load the right one.

Usually, such python fragments consist of settings that are collected into a proxy, to be applied to the component at the time of its initialization. For example:

```
ktJets = Algorithm("JetAlgorithm/KtJets")
ktJets.OutputLevel = VERBOSE
ktJets.AlgTools += ["JetKtFinder/KtFinder"]
ktJets.KtFinder.BeamType = "PP"
```

Note that this fragment contains both simple property settings as well the setup of an intercomponent relation between `KtJets` and its private tool `KtFinder`.

The usage of python as the job configuration language has many advantages: its rich, standardized, and well-documented syntax allows for simple, expressive fragments like the one above, and component developers can exploit python's logical constructs to deal with settings given a job context. In effect, this allows the combination of several configuration fragments into one file.

There are, however, three immediate problems: since the properties are defined for a component that is not yet loaded, they can not be verified for syntax or value errors; the names of the components are hardwired in this fragment, making it impossible to be re-used by derived classes or in jobs involving multiple instances of the algorithm; and the component-centric model leaves it up in the air how to either find the right fragment to load, or how to provide the proper logic settings for a combined fragment, when constructing a complex job.[1]

User-centric Job Configuration

It is less than two years before data-taking, and the Atlas software system is now used more by physicists building their analysis codes in anticipation of collisions, than by developers constructing new algorithms. The software has to make a transition from a system that mainly served developers, to one that serves both developers and end-users well. This is particularly evident in the job configuration.

The organization of job configuration in file fragments has followed the package structure of the components. While this makes sense from the developer point of view, it has the wrong granularity for an end-user of the component: localizing the right fragment for the job, if one exists, in the code repository can be rather challenging. More complex tasks that involve mixing, matching, and/or including in the right order, fragments originating from different contexts (e.g. detector commissioning and reconstruction of simulated data) do seldom work out, because the original fragment authors did not foresee the resulting combinatorics.

The end result is that the user has to do a lot of guessing as to what will work, and why, as well as what tweaks are necessary and how to apply those. The goal of an improved configuration from the point of view of the end-user, is to take that guesswork out of the equation. Doing so is a straightforward three step process:

1. Provide smarter, low-level building blocks that are *automatically* generated for all components, and while they fulfill the role of proxies, they should contain the component defaults as compiled in, and carry sufficient information to verify property settings.

2. Provide structuring support to standardize the setup of intercomponent relations, to ensure that they can be modified or even undone, and to remove custom logic constructs from job options fragments.

3. Build higher level structures, following the direction of the physics and developer communities, that correspond to elements in the same terms as the end-user likes to think (e.g. physics processes).

These steps can be summarized as: automate, reduce entropy, and improve the user interface.

CONFIGURABLES

A *Configurable class* is a python class that corresponds one-to-one to a GAUDI component class and carries the complete information of property names, types, documentation, and default values. A *Configurable instance* is an instance of a Configurable class, and corresponds with a GAUDI component instance that *may* exist at some point in the execution of the ATHENA job. It has access to the component property information from its class, and in addition keeps the actual values as set in the configuration.[2] A schematic example, based on the “HelloWorld” Algorithm, is provided in Fig. 1.

A Configurable is identified by a name, which is the class name by default. This immediately solves the multiple instance problem mentioned above: the defaults are now associated at the class level, rather than with any particular instance, and will be set for all instances that are selected for the job. Thus, instead of loading a fragment that explicitly uses a name in the `Algorithm` declaration, the user loads a module with a class and decides herself on the name, when instantiating the class.

Configurable classes are generated automatically by querying each component for its properties, their default values (which automatically yields the types), and the optional[3] documentation string. Note that if the defaults as set in the generated code are not complete, the developer can implement a derived class from the generated class and override the conventional `setUserDefaults()` member function, to take care of any left-over details, or to package the same component with different defaults for different uses. This is particularly useful for AlgTools.

The availability of the name and type information of all properties, allows ATHENA to check as early as during configuration for attribute and type errors in the python fragments. The alternative is to wait until much later in the program, when the libraries are loaded and the matching C++ component is created and initialized. The major problem with the latter approach is that in a given job, the component may never be loaded during development/testing, and hence its configuration would never be checked.

In addition, the availability of the documentation and the history of values set, allows for new debugging tools: it is now possible to browse, build, validate, and store a configuration without having to start ATHENA.

Finally, since the Configurable is really the proverbial extra intermediate layer that buys flexibility, it is also possible for developers to customize access and tracking for their specific components. For example, the default property descriptors can be replaced by ones that flag unintentional overwrites, or provide verification of certain semantical aspects of property settings: property boundaries and constraints, validity of a file identifier, etc.

Property Repository

The python modules containing Configurable classes are automatically generated during the build process of an Atlas software release. They can be collected and the ensem-

[1] A typical Atlas reconstruction job uses hundreds of components, configured in more than 10K lines of python distributed over hundreds of file fragments.

[2] The actual “value” is actually a history list of changes as applied to the Configurable instance.

[3] Often, the use of a property is obvious and doesn’t require any additional documentation: e.g. a property such as “minPt” for a track reconstruction algorithm is clear enough.

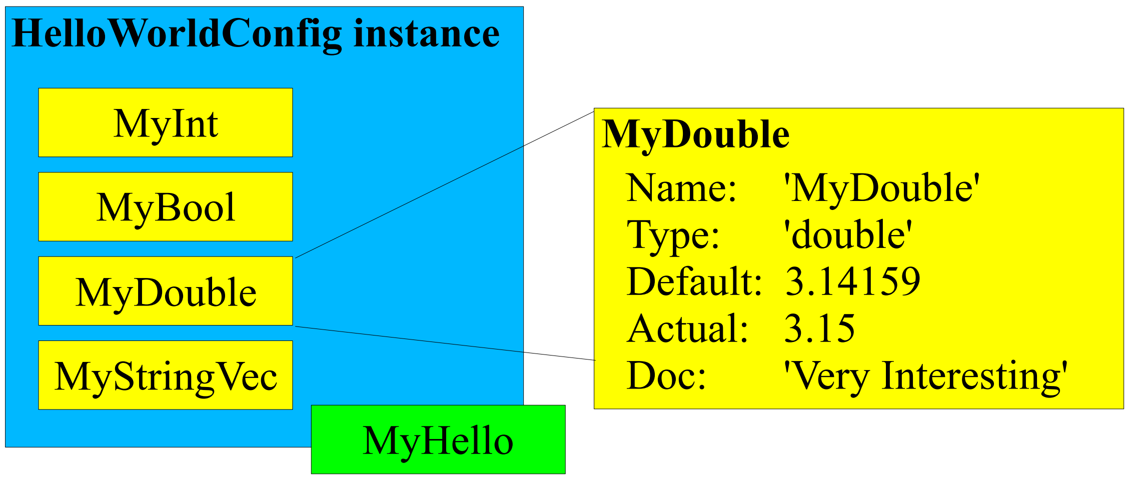

Figure 1: Schematic example of a Configurable: an instance of a generated Configurable class, labeled with name "MyHello", corresponds to an instance of the GAUDI component "HelloWorld/MyHello." It carries all the information that is available about the component from its class (name, type, default, and documentation of each of its properties) as well as the actual values from settings in the job configuration.

ble of these modules defines the *Property Repository*. The repository can also be used by a variety of job configuration tools (editors, browsers, documentation generators) to extract the property settings of any given component. The high level trigger community is interested in using this to fill a database, allowing easy and effective logging of job configurations used in the various trigger menus.

Structuring Job Configuration

Configurable instances can be chained as Composites [3]: they can contain a sequence of other Configurables, etc. The typical use case is an Algorithm using a number of AlgTools: the AlgTools, and therefore their configuration, do not have full meaning until the Algorithm requests their instantiation in a certain context. For example, particle propagation codes are different, for different magnetic fields. The field in the Atlas inner detector can be approximated with a solenoid, or a field map can be used. Based on the event structure and required precision, a tracking algorithm can favor one propagator tool over another, and the properties of that tool, such as minimum step size and particle cut-off energy.

Taking advantage of subclassing and sequences, the job configuration fragment given above can now be rewritten:

```
class KtJets(JetAlgorithmConfig):
  def __init__(self, name, finder=None):
    JetAlgorithmConfig.__init__(self, name)
    if not finder:
      self += JetKtFinderConfig("KtFinder")
    else:
       self.KtFinder = finder

  def setUserDefaults( self, ktJets ):
    ktJets.OutputLevel = VERBOSE
    ktJets.KtFinder.BeamType = "PP"
```

which is then subsequently used by the end-user (or in any another fragment) like so:

```
topSeq = AlgSequence( "myJob" )
topSeq += KtJets("KtJets")
```

Notice how `JetAlgorithm` and `JetKtFinder` tool are now instances of the auto-generated concrete Configurable classes, rather than generic Algorithm and AlgTool handles. Notice also how the `JetAlgorithm` Configurable instance now manages directly a sequence of "private" tools, nicely packaged in a class written by the component developer. Finally, notice that the sequences can be modified even after the fact: the user can remove/add complete sequences at a granularity that makes sense from the job configuration perspective. For example, removing `KtJets` from the top sequence will also immediately discard the settings for the finder tool.

Thus, sequences of Configurables are the foundation for task-oriented rather than component-oriented job configuration: they immediately provide a higher level structure that hides details in an easy to find location. Furthermore, they also help to handle a job configuration issue known as "key propagation," which is particularly hard to nail in the increasingly complex reconstruction jobs: most every Algorithm result in ATHENA is identified using a unique "well-known" name.[4] These well-known names are usually set as string properties of an Algorithm or an AlgTool, which allows control at the job configuration stage, as to which of several possible strategies to produce, say, a jet collection should be used. For all its flexibility, this string-based implementation of the Strategy pattern [3] creates long-range couplings between properties of several ATHENA components. These names can now be associated with sequences, or the selection of certain sequences, and

[4]Plus a numeric type identifier.

these can propagate the names to their children, preventing ordering problems and settings in multiple locations, while allowing value checking.

Physics-level Job Configuration

What constitutes an appropriate sequence granularity depends on the user: it is different for a subdetector developer, than for a reconstruction coordinator, than for a member of the heavy Higgs analysis group. But the ingredients are always the same, and since different sequences can peacefully coexistent (sequence instantiations do not cause their configuration to be used: only those selected by adding them to a top sequence are considered), it is now possible to write sequences for each developer-user relation, including the three mentioned above.

To support these developer-user relations, any properties that are set directly to the Configurable instance, e.g.:

```
topSeq.KtJets.OutputLevel = ERROR
```

will always, regardless of ordering,[5] take precedence over those that are set in `setUserDefaults()`, which in turn will always take precedence over the generated defaults. This way, a user can start experimenting with different settings, and once satisfied that a block of settings deserves its own fragment, collect them all in a new derived class.

All the tools are now in place, the physics level job configuration is now steered by their consumers: the developers and physicists doing the actual work. Two, more or less independent, implementations are underway: in the Atlas tracker software, which makes heavy use of AlgTool selection based on, among others, the subdetector that it is used in; and in the full reconstruction, which makes heavy use of Algorithm selection, based on the physics in the provided events. The initial prototypes are very promising, do considerably cut down turnaround time for users, and will be used in the next major Atlas software release.

ACKNOWLEDGEMENTS

We gratefully acknowledge the contribution of all members of the Atlas Users and Usability Task Forces. Special thanks to Pere Mato from CERN who initiated the autogeneration of concrete Configurable classes starting from the information in ATHENA/GAUDI component libraries.

REFERENCES

[1] C. Leggett, et al., "The Athena Control Framework in Production, New Developments and Lessons Learned" CHEP'04, Conference Proceedings, Interlaken, Switzerland, Sep. 2004

[2] http://www.python.org.

[3] E. Gamma, et al., "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, Jan. 1995

[5] User settings among themselves do follow normal flow ordering.