Configuration and scheduling of the LHCb trigger application

Rosen Matev[1,*], *Niklas* Nolte[1,2,**], and *Alex* Pearce[1,***]

[1]CERN, Geneva, Switzerland
[2]TU Dortmund University, Dortmund, Germany

Abstract. For Run 3 of the Large Hadron Collider, the final stage of the LHCb experiment's high-level trigger must process 100 GB/s of input data. This corresponds to an input rate of 1 MHz, and is an order of magnitude larger compared to Run 2. The trigger is responsible for selecting all physics signals that form part of the experiment's broad research programme, and as such defines thousands of analysis-specific selections that together comprise tens of thousands of algorithm instances. The configuration of such a system needs to be extremely flexible to be able to handle the large number of different studies it must accommodate. However, it must also be robust and easy to understand, allowing analysts to implement and understand their own selections without the possibility of error. A Python-based system for configuring the data and control flow of the Gaudi-based trigger application is presented. It is designed to be user-friendly by using functions for modularity and removing indirection layers employed previously in Run 2. Robustness is achieved by reducing global state and instead building the data flow graph in a functional manner, whilst keeping configurability of the full call stack.

1 Introduction

The LHCb experiment [1] is undergoing a major upgrade for the third run of the CERN LHC in 2021 [2, 3]. This involves a new tracking detector and the removal of the hardware trigger stage to gain versatility and efficiency for physics selections. Since we need to meet much higher computational performance requirements with respect to previous runs, the trigger software is currently being rewritten almost entirely. In the second, final high-level trigger stage, we plan to run $\mathcal{O}(10^4)$ unique selection and secondary vertexing algorithm instances to be able to efficiently identify the desired signatures for a large number of different physics analyses. The configuration of such a system needs to be extremely flexible due to the many different studies it must support. To keep an overview, such a configuration also needs to be readable, traceable and consistent throughout the system. A high-level layout of the data and control flow at LHCb will be discussed in Chapter 2. Next, algorithmic foundations of the LHCb software framework are introduced. Chapters 4 and 5 will show the key features that we have introduced to achieve a robust configuration setup.

[*]e-mail: rmatev@cern.ch
[**]e-mail: nnolte@cern.ch
[***]e-mail: alex.pearce@cern.ch

2 Data and control flow in the trigger application

A trigger application is a decision algorithm, albeit a complex one in reality. Based on reconstructed event properties, we either want to save or to discard an event. The LHCb detector will produce around 40 Tb of data per second, far larger than offline disk space permits. Hence, the trigger's job is to select interesting events, where "interesting" is a union of many and very different physics signatures, in order to reduce the experiment's output bandwidth. To achieve this in a manner which is efficient across the broad experimental programme, thousands of selection lines need to be carefully tailored to the desired physics signatures and then run as part of the trigger's control flow, illustrated in Figure 1.

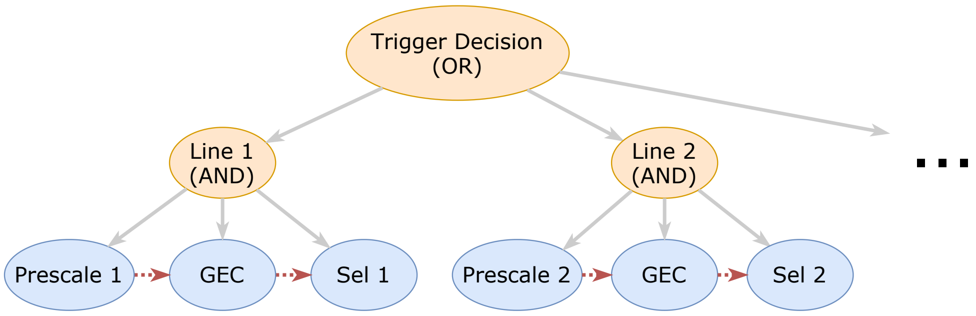

Figure 1. The control flow of a typical trigger application, with basic (blue) and composite (yellow) control flow nodes. Trigger lines can be prefixed by a 'prescale' to control their overall rate. A global event cut (GEC) can be used to remove the busiest events from the processing chain. Each line has a selection specific to the physics signature it tries to select, e.g. a specific particle decay like $D^0 \to \pi^+\pi^-$.

Most *basic* control flow nodes, algorithm instances, have data dependencies. The execution of a node requires the execution of the algorithms which satisfy its data dependencies. A detailed scheme of a line which selects muon candidates is shown in Figure 2. Managing thousands of these lines, as will be the case at LHCb in Run 3, requires a framework which reduces any per-selection maintenance burden and which allows the concepts of control and data flow to be expressed naturally.

3 Algorithms in LHCb software

The LHCb software is written in C++ and based on the Gaudi framework [4], which provides different templates for algorithms that can interact with an event-local storage. Almost all algorithms used in the LHCb Run 3 software derive from Gaudi's *functional* algorithm templates. Unlike other algorithm templates in Gaudi, these model pure functions. The evaluation of a pure function has no side effects and the return value depends only on the inputs, which are declared explicitly in the algorithm's API. The implied reentrancy and thread safety is vital for the trigger application, which will process events in parallel via multi-threading rather than via multiple processes. The different kinds of functional algorithm templates in Gaudi are visualized in Figure 3.
Inputs to functional algorithms are loaded from an event-local store by key lookup and outputs are pushed into the store after evaluation. Data pushed into the store is immutable, and can be used as input for any number of successive algorithms.

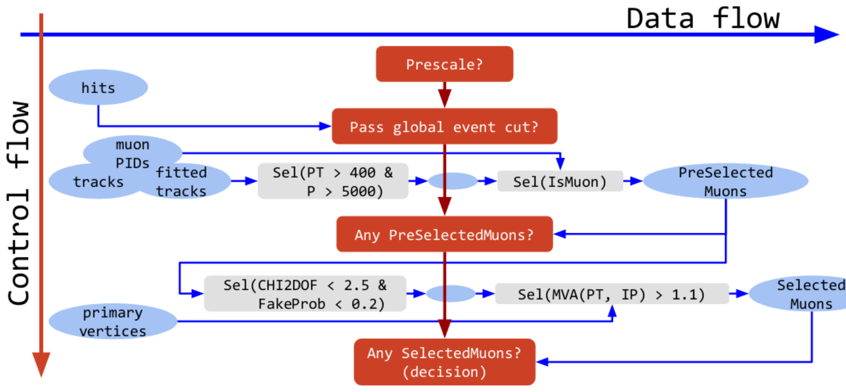

Figure 2. Schematic view of control and data flow defining a selection. Control flow nodes are shown in red, data dependencies in blue and selection algorithms in grey.

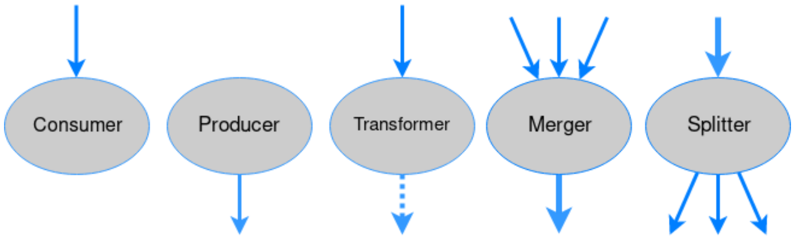

Figure 3. Different functional Algorithms and their input and output behaviour.

Algorithms can declare parameters which alter runtime behaviour, such as kinematic thresholds or feature flags. Input and output keys are also parameters, such that one is able to define data dependencies at configuration time. Parameter values can be set from a Python application, which runs once at the start of data processing, without having to recompile the software. Every Gaudi algorithm exports a Python class, a *configurable*, which holds all parameters as data members. To apply a non-default configuration, one instantiates a configurable and sets parameters within an *options file* that is an argument to the main application. Each configurable has a name by which it can uniquely be referred to. They are implemented as singletons up to their names:

```
from Configurables import Alg
# exported by defining struct Alg : public Transformer<...> {...}
x = Alg(name="myAlg")
Alg("myAlg").RapidityThreshold = 2 # retrieve the global singleton and set ↩
    its property
assert x is Alg("myAlg")
assert x.RapidityThreshold == 2
```

Implementing configurables as named singletons enables users to tweak many aspects of the configuration without having to change routines explicitly. They may apply parameter configuration in their options file, which overwrites any previously bound value to this parameter. This allows for ultimate flexibility, but creates a “who’s last wins” condition which can be difficult to reason about.

4 A new approach to data flow configuration

In practice, most of the configuration takes care of setting up data dependencies rather than thresholds or other parameters. It is then important that the definition of the data flow is readable and traceable to allow analysts to quickly gain an overview over how their data is produced. The following goals or guidelines are defined for configuring our trigger application:

1. Functions are to be kept small and only serve a single purpose. If the purpose is producing data, rather than a control flow node for example, its name starts with `make_`.
2. Functions need to be pure, interacting with minimal global mutable state.
3. When an algorithm configurable is instantiated (as part of a data producing function), the instance should be immutable.

These goals go hand-in-hand and yield a functional way of configuring data flow, mirroring the way we apply functional processing in the C++ algorithms. The following is an example of a data producing function:

```python
from Algorithms import LongTracking


@configurable
def make_long_tracks(make_tracks=make_default_upstream_tracks,
                     make_hits=make_default_scifi_hits,
                     minimum_pt=400 * MeV):
    """Makes long tracks from upstream tracks and scifi hits"""

    # prepare inputs
    hits = make_hits()
    upstream_tracks = make_tracks()

    # setup tracking algorithm
    tracking = LongTracking(InputHits=hits,
                            InputTracks=upstream_tracks,
                            MinimumPT=minimum_pt)

    # return the data
    return tracking.OutputTracks
```

The function defines reasonable defaults and configures one part of data flow. The `Algorithms` module is a wrapper module around the `Configurables` module provided by Gaudi. Importing an object from the `Algorithms` module returns the usual configurable but wrapped in the `Algorithm` class. This wrapper is immutable after instantiation and distinguishes input and output (I/O) parameters from other parameters (made possible since these parameters are declared as I/O in functional algorithms). A special 'data handle' object type is used when assigning values to these parameters. This allows for advantageous behaviour:

1. An `Algorithm` verifies that all input parameters are specified on instantiation.
2. The objects representing output parameters are automatically generated on instantiation, and these hold a reference back to the producer algorithm. These objects can then be used as input parameter values in subsequent algorithm instantiations. Thus, an algorithm has access to the entire data flow graph of each of its inputs. These graphs can be be plotted, as illustrated in Figure 4, which was generated from one of the first data producers in the trigger data flow with `make_velo_tracks().plot()`.

3. A hash identity is formed based on an `Algorithm` instance's inputs and other properties in order to allow memoization of instantiation. This implies de-duplication if all properties and all inputs match across instantiations, where input hashes depend on the hash of their producing algorithm. Multiple calls to the `make_long_tracks` example above with the same arguments will then always return the same object, removing the need to pass the data around everywhere it is needed.

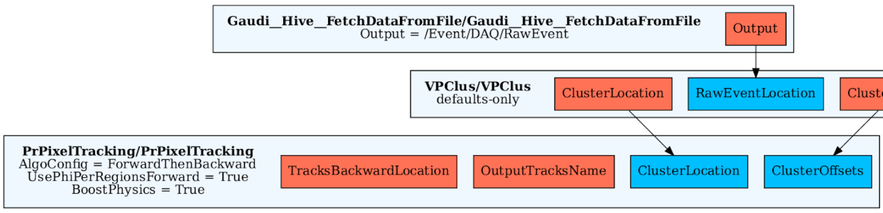
Gaudi__Hive__FetchDataFromFile/Gaudi__Hive__FetchDataFromFile
Output = /Event/DAQ/RawEvent
Output
VPClus/VPClus
defaults-only
ClusterLocation
RawEventLocation
ClusterOffsets
PrPixelTracking/PrPixelTracking
AlgoConfig = ForwardThenBackward
UsePhiPerRegionsForward = True
BoostPhysics = True
TracksBackwardLocation
OutputTracksName
ClusterLocation
ClusterOffsets

Figure 4. An example dataflow graph, generated by calling the `plot` method on the Python data representation

The reason for preferring small functions to define data flow components is twofold. Firstly, its easier to follow the single responsibility principle with small functions. Secondly, it increases modularity in the configuration. Being able to step in at almost every point in the configuration to get the relevant dependency sub-tree is feature that was not available before. A developer of configuration should be able to branch off of existing data flow configurations at as many places as possible to minimize code duplication and unnecessary execution.

Although a high level of modularity allows maximum flexibility, it is cumbersome to define small changes in the middle of the data flow, such as those that only differ in one aspect from the default configuration. One has to branch off at this point and duplicate the data flow up to the desired output, or change the default behaviour. Changing default behaviour involves modifying routines by hand, as one would otherwise have to modify every call site. Previously, the singleton-like behaviour of `Configurables` made it possible to change parameters by simply overwriting configuration: retrieve the global configurable instance by name and set the property as desired. This proved to be so useful for post-production use cases that we decided to implement something similar, but more powerful and explicit in this framework. The `@configurable` decorator can be attached to any function to make its arguments configurable within a Python `with` context. The effect can be seen in the following example:

```
@configurable
def filter_tracks_by_P(min_p=1 * GeV)
    print(f'keeping tracks with P > {min_p}')
    tracks = MomentumFilter(min_p)
    return tracks


def use_tracks():
    tracks = filter_tracks_by_P()
    do_something_with(tracks)


with filter_tracks_by_P.bind(min_p=2 * GeV):
    use_tracks() # -> 'keeping tracks with P > 2000'
```

After decoration with `@configurable`, the `bind` method can be used on the function object, enabling deep configurability only in the context of the `with` scope. The `bind` functionality comes with a debug mode, which prints every occasion of argument overwriting. Multiple overwrites can occur, but will always result in at least a warning, allowing developers to understand where configuration clashes are occuring. Using `bind` in production code is discouraged, to reduce cognitive overhead when reading the default configuration, as was experienced with the `Configurables` approach.

5 Control flow configuration

How data is assembled to create a selection line is best visualized with an example:

```
def dzero2Kpi_line(name='Hlt2D02KpiLine'):
    event_filter = GlobalEventCut(hit_threshold=3000)
    long_tracks = make_long_tracks()
    kaons = make_kaons(long_tracks, Cut=PID_K > 3)
    pions = make_pions(long_tracks, Cut=PID_K < 3)
    dzeros = make_dzeros(
        particles =[kaons, pions],
        decays = ['[D0 -> K- pi+]cc'])
    return HltLine(
        name = name,
        children = [event_filter, dzeros],
        prescale = 1
    )
```

The returned object `HltLine` represents a composite control flow node as previously described in Figure 1. The children argument defines the nodes it depends on. By default, an `HltLine` represents a lazy logical `AND`, meaning that its decision is the intersection of all boolean child values with short-circuiting enabled. If the event filter fails, the combiner algorithm will not run. Note that the data dependencies of the combiner algorithm do not need to be specified in the control flow. Because the data dependency graph of the algorithm can be inspected, the framework can deduce which algorithms must be run in order to evaluate the control flow result.

6 Conclusion

We have introduced a functional way to configure the LHCb trigger application for Run 3 of the LHC. The main design goals were reducing implicit behaviour and global state present in the previous generation of configuration code, whilst allowing deep configurability. We have received positive feedback from developers and analysts on the ease of comprehension and debugging features of the new framework, and have begun expanding its usage into an offline context. Around 150 selection lines have already been written, and we are confident that the functional approach to data and control flow configuration will scale to the thousands required for the full LHCb physics programme.

References

[1] The LHCb Collaboration, JINST **3**, S08005 (2008)
[2] The LHCb Collaboration, Tech. Rep. CERN-LHCC-2011-001. LHCC-I-018 (2011)
[3] The LHCb Collaboration, Tech. Rep. CERN-LHCC-2018-007. LHCB-TDR-017 (2018)
[4] G. Barrand et al., Comput. Phys. Commun. **140**, 45 (2001)