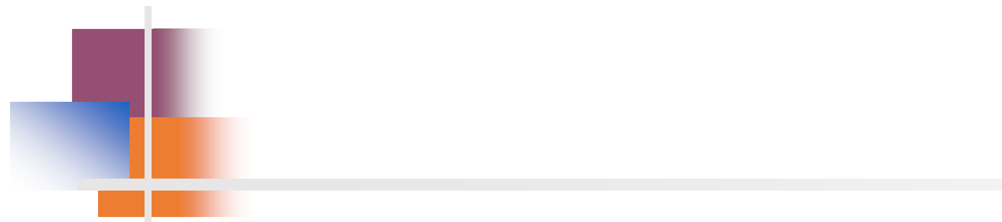
The Gaudi framework for SPD

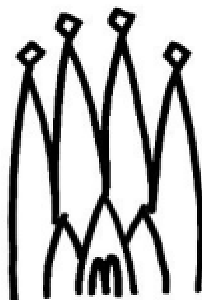
Valeriy Onuchin

JINR/DNLP/SPD



GAUDI

LHCb Data Processing Applications Framework



Architecture Design Document

Reference:	LHCb 98-064 COMP
Version:	1.0
Created:	November 9, 1998
Last modified:	November 24, 1998
Prepared by:	LHCb software architecture group Editor: P. Mato

24 years of development

- Adapted by :
    - LHCb (CERN)
    - ATLAS (CERN)
    - FCC (CERN)
    - EIC (BNL, USA)
    - SCTau (BINP, Novosibirsk)
    - BES III (IHEP, China)

What is the Gaudi framework?

The Gaudi framework runs over a list of events, providing ways to process them and store data in a new format. It creates and manages **Data Objects**, which can hold a variety of data. A **Transient Event Store (TES)** stores data objects in a way to make them accessible to the rest of the framework, and parts of it can be made persistent in a ROOT format file. The data in the TES is created and accessed by **Algorithms**, which produce data objects and process data objects. Gaudi also provides **Services**, which provide access to other parts of the framework, such as histograms. **Tools** are lightweight routines that are also available. The Application Manager manages these components.

Algorithms

This is the most important component of the framework for an user to know. Algorithms are called once per physics event, and (traditionally) implement three methods beyond **constructor/destructor**: **initialize**, **execute**, and **finalize**. Also, **beginRun** and **endRun** are available, though be careful not to misuse state.
A set of Gaudi Algorithm Examples can be found at https://gitlab.cern.ch/gaudi/Gaudi/-/tree/master/GaudiExamples

Very useful Gaudi Algorithms templates were developed by LHCb guys https://gitlab.cern.ch/lhcb/LHCbSkeleton
They include Python scripts to create **Gaudi Functional Algorithm** C++ classes.
Must be adapted to SPD!

Gaudi Functional Algorithm

SPD

Out/in	0	1-n	vector
0		Consumer	
1	Producer	Transformer	MergingTransformer
n	Producer	MultiTransformer	
vector		SplittingTransform	
boolean		FilterPredicate	
boolean+1-n		MultiTransformerFilter	

Possible inputs

- **0** : no input, pure producer
- **1-n** : one or several independent inputs. This number and the type of each input must be known at compile time
- **vector** : any number of inputs (not know at compile time), all of the same type, aka a vector of inputs of the same type

Possible outputs

- **0** : no output, pure consumer
- **1-n** : one or several independent outputs. This number and the type of each output must be known at compile time
- **vector** : any number of outputs (not know at compile time), all of the same type, aka a vector of outputs of the same type
- **boolean** : an additional boolean output, saying whether we should carry on or stop the processing, aka a filter

- Many algorithms look like "data in $\rightarrow$ data out"
- Standardize this pattern, and factor out getting and putting the data
  - less code to write
  - more uniform code, easier to understand
  - move maintenance of annoying details to the framework
  - fix bottlenecks once and for all
- Patterns available
  - Consumer, Producer, Filter, Transformer,

Gaudi::Functional practical code

```
class MySum: public TransformAlgorithm
    <OutputData(const Input1&, const Input2&)> {
  MySum(const std::string& name, ISvcLocator* pSvc)
  : TransformAlgorithm(name, pSvc,
                       { KeyValue("Input1Loc", "Data1"),
                         KeyValue("Input2Loc", "Data2") },
                       KeyValue("OutputLoc", "Output/Data") )
  {}
  // ...
  OutputData operator()(const Input1& in1,
                        const Input2& in2) const override {
    return in1 + in2;
  }
  // ...
}
```

Properties

Algorithms are a **Configurable**, which means they can be accessed in Python and **Properties** can be manipulated there. In the classic API, a property is declared in the *constructor*, using:

```
declareProperty("PropertyName", f_value, "Description of property");
```

Here**, f_value** is a reference to a variable for an int, string, etc. It is almost always a member variable for the class so that you can access it in the other methods.

To use a property, you can simply access it on the configurable in **Python**:

```
my_algorithm.PropertyName = 42
```

Another way to declare property is to define a member variable of type **Gaudi::Property<>**:

```
Gaudi::Property<int> m_some_int{this, "SomeInt", 0, "Description of some int"};
```

Transient Event Store (TES)

The **TES** is a place where you can store items on a per-event basis. It should be viewed as non-mutable; meaning that once you place an item in it, it should never change. Persistence is optional. The path to an event should always start with **"/Event",** though Gaudi is smart enough to assume that a path that does not start with a slash is a relative path, and will get **"/Event/"** prepended to it. In the classic framework, you used get and put functions to access the **TES**. For this you must use **GaudiAlgorithm** instead of **Algorithm**, which is a specialization to add access to the **TES**. You will need to add **GaudiAlgLib** to the linked library list for the **GaudiAlgorithm**.

To place an item in the event store, create a pointer to a new object, and then put it in the event store in an execute method:

```
auto data = new DataObject();
put(data, "/Event/SomeData");
```

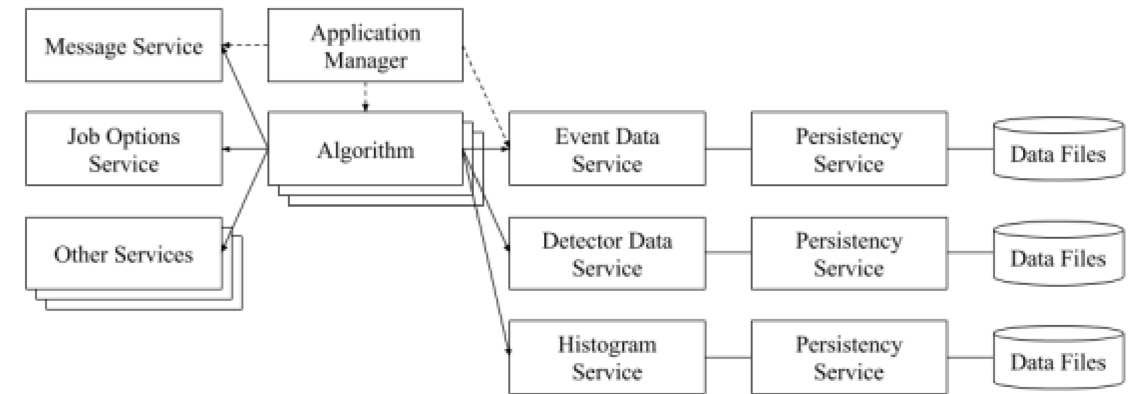
The event store will take ownership of the object, so do not delete it.
To retrieve it, also in an execute method:

```
auto data = get<DataObject>("/Event/SomeData");
```

Gaudi Services

The basic modules of the GAUDI framework and the connections between them are shown in the picture

The Application manager (in code denoted as ApplicationMgr) controls the execution of the jobs within the framework. It creates and initializes the required modules in the system, and retrieves input data. The input data is a collection of highly-structured information that describe particle collisions (also called events) recorded inside the detectors or created in simulations. The Application manager loops over the input data events and executes the algorithms.



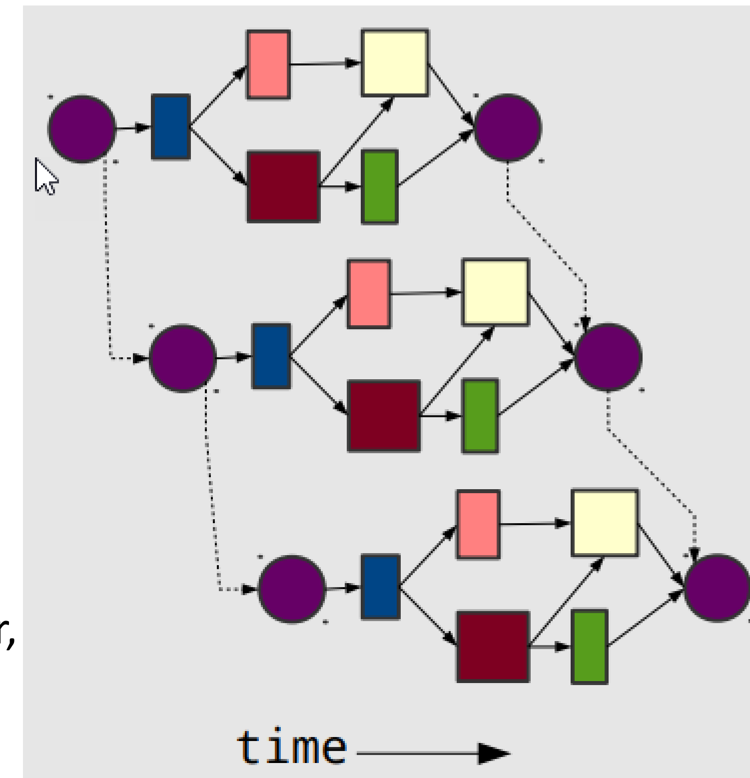
The GAUDI services provide various utilities and services for the Algorithms in the system, which are also initialized by the Application manager at the beginning of a job. Normally, only one instance of a service is required in the job. There are a number of different services within the framework that can be used by the Algorithms but some of the main ones are:

- Event Data service (EventDataSvc) and Histogram service (HistogramDataSvc) that read and process individual events,
- Detector Data service (DetDataSvc) for capturing detector data,
- Message service (MessageSvc) logs progress or errors in the Algorithms and
- Tool Service (ToolSvc) manages Algorithm tools, which are required during the Algorithm execution.

The Persistency services allow writing the output data on the disk. There are many other services in the framework that provide specialized functions that can be enabled and disabled by the users.

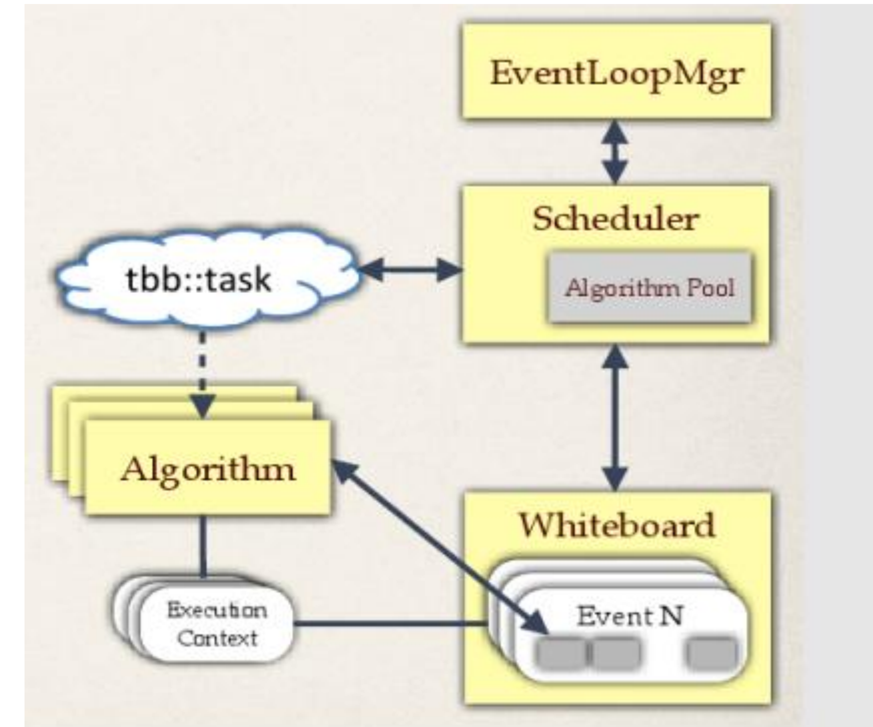
Gaudi Hive

- Gaudi Hive: multi-threaded, concurrent extension to Gaudi
  - uses Intel TBB for thread management
    https://github.com/oneapi-src/oneTBB
- Data Flow driven
  - Algorithms declare their data dependencies
    - build a directed acyclic graph - can be used for optimal scheduling
  - Scheduler automatically executes Algorithms as data becomes available.
- Multi-threaded
  - Algorithms process events in their own thread, from a shared Thread Pool.
- Pipelining: multiple algorithms and events can be executed simultaneously
  - some Algorithms are long, and produce data that many others
    need (eg track fitting). instead of waiting for it to finish, and idling processor, start a new event
- Algorithm Cloning
  - multiple instances of the same Algorithm can exist,
    and be executed concurrently, each with different Event Context.
  - cloning is not obligatory, balancing memory usage with concurrency.
  - support for re-entrant Algorithms



Gaudi Hive Operation

- Configuration, Initialization, Finalization are performed serially in "master" thread
  - only Algorithm::execute is concurrent
- Algorithms are scheduled when data becomes available
  - Algorithms must declare their inputs at initialization or dynamically with DataHandles
  - data only exchanged via whiteboard
  - bb::task wraps the pair (Algorithm*, EventContext)
- Algorithms can be non-cloneable (singleton), cloneable, or re-entrant
  - more clones = more memory, but greater opportunity for concurrency
    - cardinality is tunable at runtime
  - re-entrant is best, but hardest to code
  - tbb layer is normally hidden from users, but Algorithms can explicitly use tbb constructs (parallel_for, concurrent_queue, etc) for finer grained parallelism
    - plays well with the Scheduler
  - Component model allows Scheduler to be replaced as needed



An example of full G4 simulation p.1

SPD

```
from Gaudi.Configuration import *
from Constants import SystemOfUnits as units

# Data service
from Configurables import ScTauDataSvc

decKey = 'gun_geantino'
podioevent = ScTauDataSvc("EventDataSvc")

from Configurables import ParticleGun
from Configurables import GenAlg
from Configurables import HepMCToEDMConverter
from Configurables import HepMCFileWriter
from Configurables import Gaudi__ParticlePropertySvc
from PathResolver import PathResolver

particlePropertySvc = Gaudi__ParticlePropertySvc("ParticlePropertySvc",
    ParticlePropertiesFile=PathResolver.FindDataFile('GenParticleData/ParticleTable.txt')
)
from math import pi
Momentum = 1500
Theta    = pi / 2.0
Phi      = pi / 2.0
dTheta =  pi / 3.0
dPhi = pi / 12.0

guntool = ParticleGun("PdgCodes", PdgCodes=[480000000])
guntool.OutputLevel=DEBUG
guntool.MomentumMin = Momentum * units.MeV
guntool.MomentumMax = Momentum * units.MeV
guntool.ThetaMin = (Theta - dTheta) * units.rad
guntool.ThetaMax = (Theta + dTheta) * units.rad
guntool.PhiMin = (Phi - dPhi) * units.rad
guntool.ThetaMin = (Theta - dTheta) * units.rad
guntool.PhiMax = (Phi + dPhi) * units.rad
```

An example of full G4 simulation p.2

```
gun = GenAlg("ParticleGun", SignalProvider=guntool)
gun.hepmc.Path = "hepmc"

writer = HepMCFileWriter("HepMCFileWriter")
writer.hepmc.Path="hepmc"

hepmc_converter = HepMCToEDMConverter("Converter")
hepmc_converter.hepmc.Path="hepmc"
hepmc_converter.genparticles.Path="allGenParticles"
hepmc_converter.genvertices.Path="allGenVertices"

# DD4hep geometry service
# Parses the given xml file
from Configurables import GeoSvc
from DetBase.DetConfigurator import DetConfigurator
detector_conf = DetConfigurator()
detector_conf.activateSubsystems( [ 'ALL' ] )

detector_geo_input = detector_conf.getGeoConfiguration()

print( 'detector_geo_input =',detector_geo_input )

geoservice = GeoSvc("GeoSvc", detectors=detector_geo_input, OutputLevel=INFO)

# Geant4 service
# giving the names of tools will initialize the tools of that type
from Configurables import SimG4UserSteppingActionTool

userStepAction = SimG4UserSteppingActionTool("ScanningAction")
userStepAction.PrintInfo = True;
userStepAction.G4Hits.Path = "trajectory"

from Configurables import SimG4ConstantMagneticFieldTool
```

An example of full G4 simulation p.3

SPD

```
field = SimG4ConstantMagneticFieldTool("SimG4ConstantMagneticFieldTool", FieldOn=True,
    IntegratorStepper="ClassicalRK4", FieldComponentX=0.0*units.tesla, FieldComponentY=0.0*units.tesla,
FieldComponentZ=1.0*units.tesla, FieldRMax=100.0*units.m, FieldZMax=100.0*units.m)

from Configurables import SimG4EmptySDTool, SimG4SensitiveDetectorMasterTool
sd_names = detector_conf.getSensDetName()
SensDetectors = []

for sd_name in sd_names:
    EmptySD = SimG4EmptySDTool(sd_name)
    SensDetectors.append(EmptySD)

SDMasterTool = SimG4SensitiveDetectorMasterTool("SDMasterTool", sensDetectors = SensDetectors)

from Configurables import SimG4Alg, SimG4GeantinosFromEdmTool
# next, create the G4 algorithm, giving the list of names of tools ("XX/YY")
particle_converter = SimG4GeantinosFromEdmTool("EdmConverter")
particle_converter.genParticles.Path = "allGenParticles"
geantsim = SimG4Alg("SimG4Alg", eventProvider=particle_converter, detector='SimG4DD4hepDetector’,
            physicslist="SimG4GeantinoDeposits", actions=[userStepAction], magneticField=field, SDMaster=SDMasterTool)

# PODIO output algorithm
from Configurables import PodioOutput
out = PodioOutput("out", OutputLevel=INFO, filename = './'+ decKey + '_g4sim.root')
out.outputCommands = ["drop allGenParticles"]

# ApplicationMgr
from Configurables import ApplicationMgr
ApplicationMgr(TopAlg = [gun, writer, hepmc_converter, geantsim, out],
    EvtSel = 'NONE', EvtMax = 10,
    # order is important, as GeoSvc is needed by SimG4Svc
    ExtSvc = [particlePropertySvc, podioevent, geoservice],
    OutputLevel = INFO, AuditAlgorithms = True, AuditTools = True, AuditServices = True)
```

The Status of SPeeD developmnet

SPD

- The project is at the stage of prototyping and testing.
  Hopefully to be ready for public testing ASAP (before collaboration week) .
- The main code was stolen from SCtau Aurora framework (G4 Simulation part)
- Build system was stolen from LHCb
- Dependency:
  - Gaudi ("/cvmfs/sft-nightlies.cern.ch/lcg/latest/Gaudi/master-4f9ac/x86_64-centos7-gcc10-opt")
  - ROOT 6.24
  - Geant4 ("/cvmfs/sft-nightlies.cern.ch/lcg/latest/Geant4/11.0.2-b78b7/x86_64-centos7-gcc11-opt")
  - DD4hep ("/cvmfs/sft-nightlies.cern.ch/lcg/latest/DD4hep/master-1c0dc/x86_64-centos7-gcc10-opt")
  - EDM4hep ("/cvmfs/sft-nightlies.cern.ch/lcg/latest/EDM4hep/00.04.01-d9194/x86_64-centos7-gcc11-opt")
  - Podio ("/cvmfs/sft-nightlies.cern.ch/lcg/latest/podio/00.14.01-642d4/x86_64-centos8-gcc10-opt")
  - CLHEP ("/cvmfs/sft-nightlies.cern.ch/lcg/latest/clhep/2.4.5.1-ebe73/x86_64-centos9-gcc11-opt")
  - HepMC3 (" /cvmfs/sft-nightlies.cern.ch/lcg/latest/hepmc3/HEAD-cfcd1/x86_64-centos7-gcc11-opt")
- All externals are precompiled. **PATH, LD_LIBRARARY_PATH, ROOT_INCLUDE_PATH, PYTHONPATH** are hardcoded.
- Pythia8 generator from https://github.com/HEP-FCC/k4Gen will be added ASAP

References

Repos & Docs:

- https://gitlab.cern.ch/gaudi/Gaudi https://gitlab.cern.ch/lhcb/Gaudi
- https://gitlab.cern.ch/gaudi/Gaudi/tree/master/GaudiExamples
- https://github.com/HEP-FCC/FCCSW
- https://gaudi-framework.readthedocs.io/en/latest/index.html
- Gaudi Workshop 2016 (https://indico.cern.ch/event/556551/)
- LHCb Gaudi docs (https://lhcb.github.io/developkit-lessons/first-development-steps/02b-gaudi-intro.html)
- Code template for the gaudi functional algorithm (https://gitlab.cern.ch/lhcb/LHCbSkeleton)

SCtau Wiki:

- https://ctd.inp.nsk.su/wiki/index.php/Simple_SCT_parametric_simulation
- https://ctd.inp.nsk.su/wiki/index.php/Use_Analysis_package

C++ programmer must read:

- Стандарт C++20: обзор новых возможностей (https://habr.com/ru/company/yandex_praktikum/blog/554874/)
- C++ Core Guideline (https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines)
- https://github.com/x2v0/x2v0.github.io/blob/master/C__Concurrency_in_Action_by_Anthony_Williams_z-lib_org.pdf