The SPeeDy framework

Valeriy Onuchin

JINR/DNLP/SPD

3.10.2022

SPeeDy is based on Gaudy framework

The great features of Gaudi :

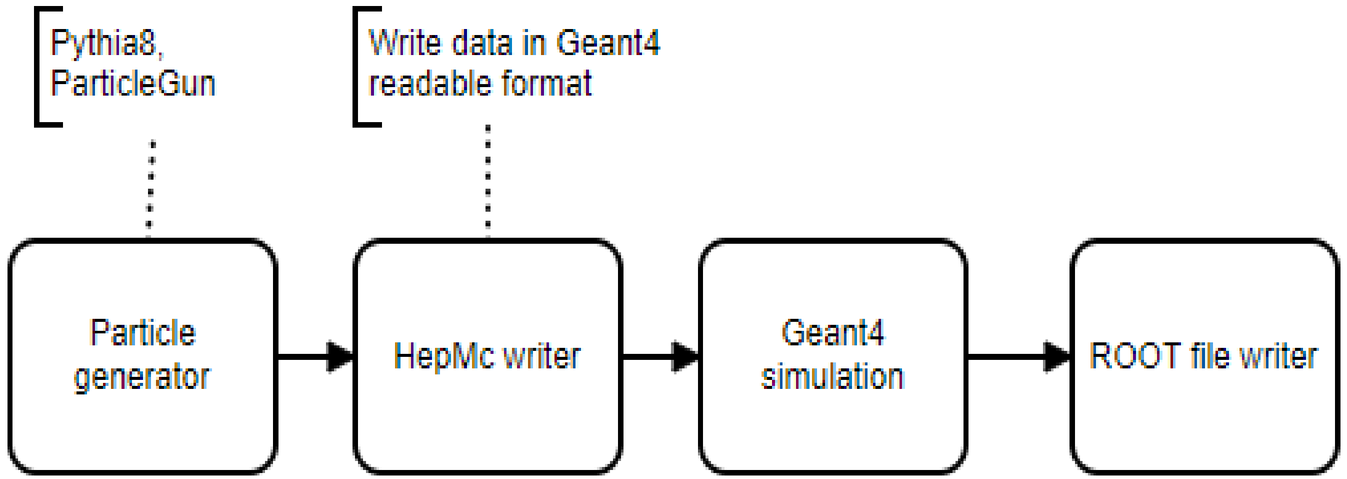
- Modularity
- Another programming paradigm
- Easy configuration via Python scripting
- Plug-in architecture
- Flexibility
- Multithreading is out of the box
- It’s mature (24 years old) but still under active development
- Adapted by old and new experiments
  - LHCb (CERN)
  - ATLAS (CERN)
  - BES III (IHEP, China)
  - Harp
  - FCC (CERN)
  - EIC (BNL, USA)
  - SCTau (BINP, Novosibirsk)

Modularity

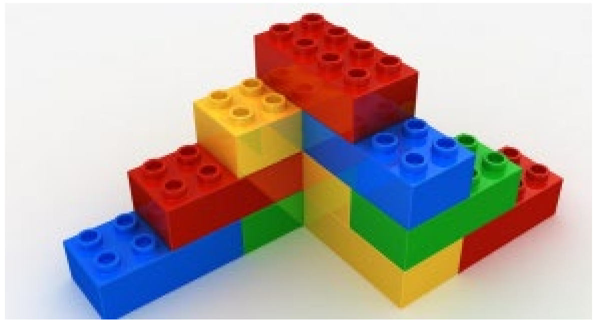
Architecture consists in processing of independent modules (algorithms)

Algorithms the most important component of the framework. Algorithms are called once per physics event.

Minimal Geant4 simulation



Programming in Gaudi is similar to to creating figure with LEGO constructor, connecting blocks together.



Algorithm structure

Python script to set algorithm's properties

C++ code of algorithm.
Class inherited from GaudiAlgorithm.
Implements **initialize, execute**, and **finalize** methods

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. Imitates file system in memory, e.g. **/Data/MyData/Event**

Configuration via Python scripting

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"};
```

Plug-in architecture

```
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
```

Flexibility

Let’s consider the track reconstruction task.

For example we have 2 track finding algorithms:

- **Traditional**
- **Machine Learning based algorithm**

We have 3 track fitting algorithms:

- **Iterative KalmanFilter**
- **Global fitting**
- **HoughTransform**

The framework allows to load (import) all of them into Python script which do processing and by switching a single parameter one can run all 6 combinations without any recompilation.

SPeeDy development

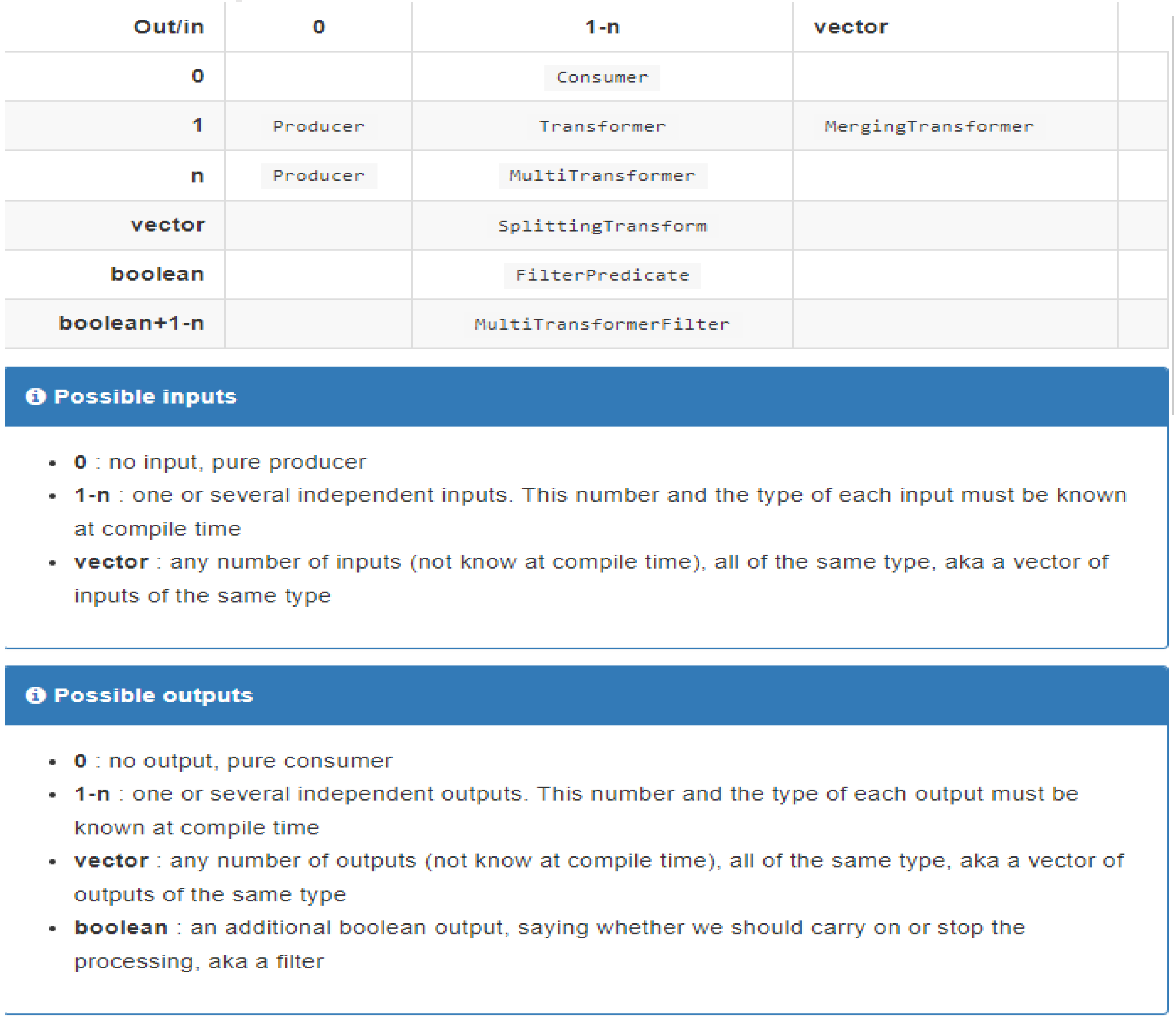
- ❑ I pursued 2 goals: **minimize initial code and minimize dependencies**
- ❑ The main code was stolen from SCtau Aurora framework (only G4 Simulation part)
- ❑ Build system (Cmake scripts) was stolen from LHCb/Gaudi
- ❑ Dependency:
  - Gaudi ("/cvmfs/sft-nightlies.cern.ch/lcg/latest/Gaudi/master-4f9ac/x86_64-centos7-gcc10-opt")
  - 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.

The status of development:

- Finally successfully built
- The repository https://github.com/x2v0/SPeeDy created
- The simple testing is under way:

  ParticaleGun shoots electrons into Electromagnetic Calorimeter (primitive geometry)

Gaudi Functional Algorithm



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 → 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, MultiTransformer, ScalarTransformer

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;
  }
  // ...
}
```

Which way to go ?

Recently I ran examples from FCCW including full Geant4 simulation based on Gaudi/key4hep.
I like it. These examples include calorimeter reconstruction.

Key4HEP is framework which claims to be an universal solution to all problems.
This collaborative effort of international team from many institutes, mostly from CERN.
Their slogan “**Key4HEP - Turnkey Software for Future Colliders**”.

https://key4hep.github.io/

My question - where to go?
Develop our minimalist framework or join this collaboration?