Sampo workshop

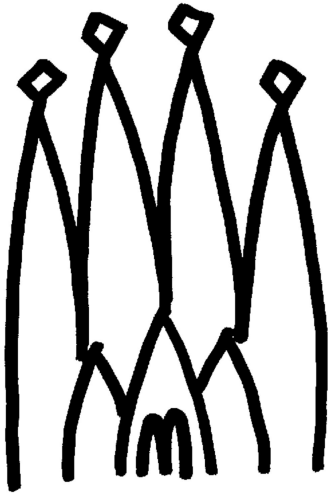
15.04.2026

Project overview

Gaudi-based offline software, substitution for SPDRoot.

Current functionality:

- Pythia8 for events generation
- GeoModel for detector description
- HepMC3 as main format for generators
- Geant4 for simulation
- Magnetic field description features
- Python for job configuring



Gaudi logo. I hope one day we will have our own.

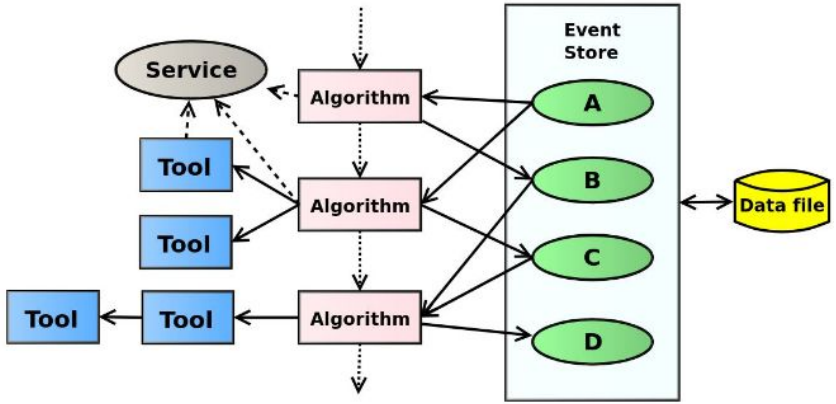
Useful links:

- Docker images: https://git.jinr.ru/spd/spd-sw/bisgm
- Sampo repo: https://git.jinr.ru/spd/spd-sw/sampo/ (check develop branch)
- Sampo wiki: https://git.jinr.ru/spd/spd-sw/sampo/-/wikis/home

Gaudi terminology

components

- DataObjects - any quantity-like objects (hits, tracks, etc.)
- Algorithms - anything that can be represented as procedure
- Services - global objects used by other components
- Tools - encapsulated subprocedures that can be injected to other components
- Transient event store (TES) - service which stores data objects and thus allows components to exchange data




```
private:
  double m_Ethreshold {10};
```

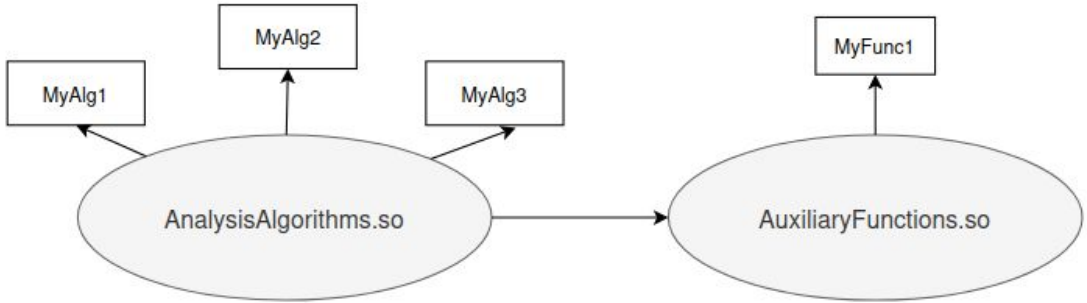
All components are configurable!

```
private:
  Gaudi::Property<double> m_Ethreshold {this, "Ethresh", 10, "Energy threshold"};
```

Components compiling

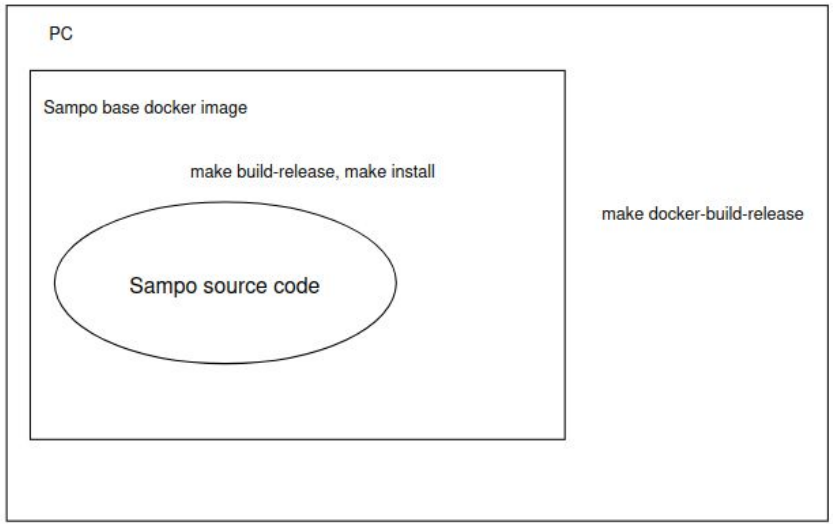
CMake is used to set up the build process. Gaudi provides a few cmake functions for different purposes:

- `gaudi_add_module` to build any component or number of components. Produced libraries are “final destination”, no other library can depend on them (meanwhile component libraries themselves can depend on other libraries and thus link to them)
- `gaudi_add_library` to build regular libraries without any components in them. Component interfaces must be organised in regular libraries though. Implicitly this function expects all header files to be located under *include* directory. Use *target_include_directories* instead if it is not the case for you.
- `gaudi_install(PYTHON|SCRIPTS)` to install python modules or any executable scripts.



Sampo build process

- Sampo is developed and built using docker
- Sampo is provided with CMakePresets: *release* and *debug.* One can either call *cmake* directly or use handy *make* instructions: *make build-release, make build-debug, make install*
- It is recommended to attach code editor to docker container



Algorithms in a nutshell

```
1  // Gaudi includes
2  #include "GaudiKernel/Algorithm.h"
3  #include "GaudiKernel/DataObjectHandle.h"
4
5  // some other includes here
6
7  class MyAlgorithm : public Algorithm {
8  public:
9      using Algorithm::Algorithm;
10     StatusCode initialize() override;
11     StatusCode execute() override;
12     StatusCode finalize() override;
13
14     bool isClonable() const override;
15     bool isReEntrant() const override;
16 private:
17     DataObjectReadHandle<MCStepCol> m_in+putStepsHandle{this, "InputSteps", TDSAddresses::MC::MylMCStepCol};
18     DataObjectWriteHandle<MyMCHitCol> m_outputHitsHandle{this, "OutputHits", TDSAddresses::MC::MyMCHitCol};
19 };
```

`initialize()`: called when algorithm is created. Initializing resources, creating histograms…

`execute()`: called once for every event. This is where most of the real action should take place

`finalize()`: called after event loop finishes. Closing files, summarizing results…

Algorithms can be clonable (more than 1 instance will be created) or reEntrant (only 1 instance, but it doesn't get blocked)

Also check Gaudi Functional: https://indico.cern.ch/event/655064/contributions/2667831/attachments/1529767/2393759/GaudiWorkshop2017.pdf

DataObjectHandles in a nutshell

- Algorithms declare their input/output in a form of *DataObject[Read/Write]Handles*
- *DataObjectHandles* are class templates, you must provide type of the object
- *DataObjectHandles* treat *DataObjects* (and their descendants) and all other classes differently. For non-*DataObjects* move semantics is applied

DataObjects

```
template <typename T>
T* DataObjectHandle<T>::put( std::unique_ptr<T> objectp ) const {
  assert( m_init );
  StatusCode sc = m_EDS->registerObject( objKey(), objectp.get() );
  if ( !sc.isSuccess() ) { throw GaudiException( "Error in put of " + objKey(), "DataObjectHandle<T>::put", sc ); }
  return objectp.release();
}
```

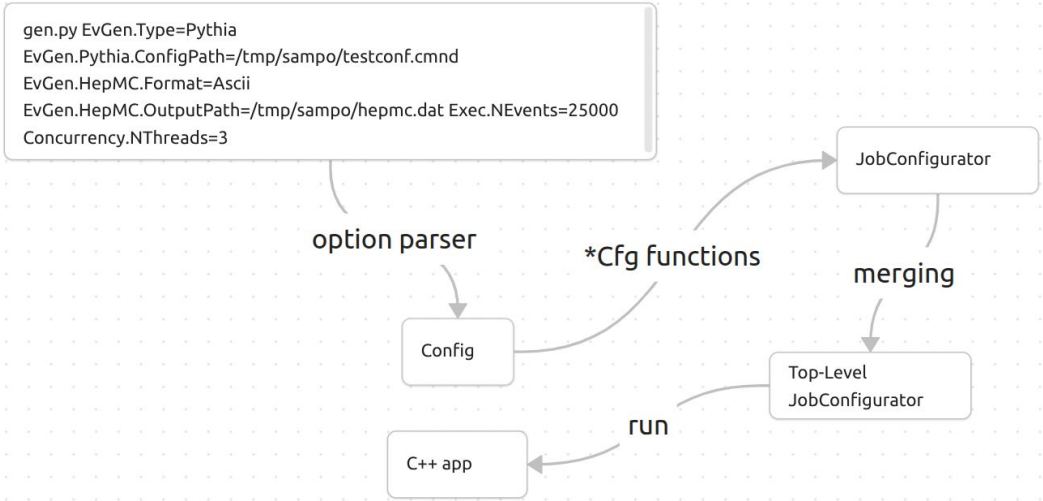
non-DataObjects

```
const T* put( T&& obj ) const {
  assert( m_init );
  auto objectp = std::make_unique<AnyDataWrapper<T>>( std::move( obj ) );
  if ( auto sc = m_EDS->registerObject( objKey(), objectp.get() ); sc.isFailure() ) {
    throw GaudiException( "Error in put of " + objKey(), "DataObjectHandle<AnyDataWrapper<T>>::put", sc );
  }
  return &objectp.release()->getData();
}
```

Sampo job configuring

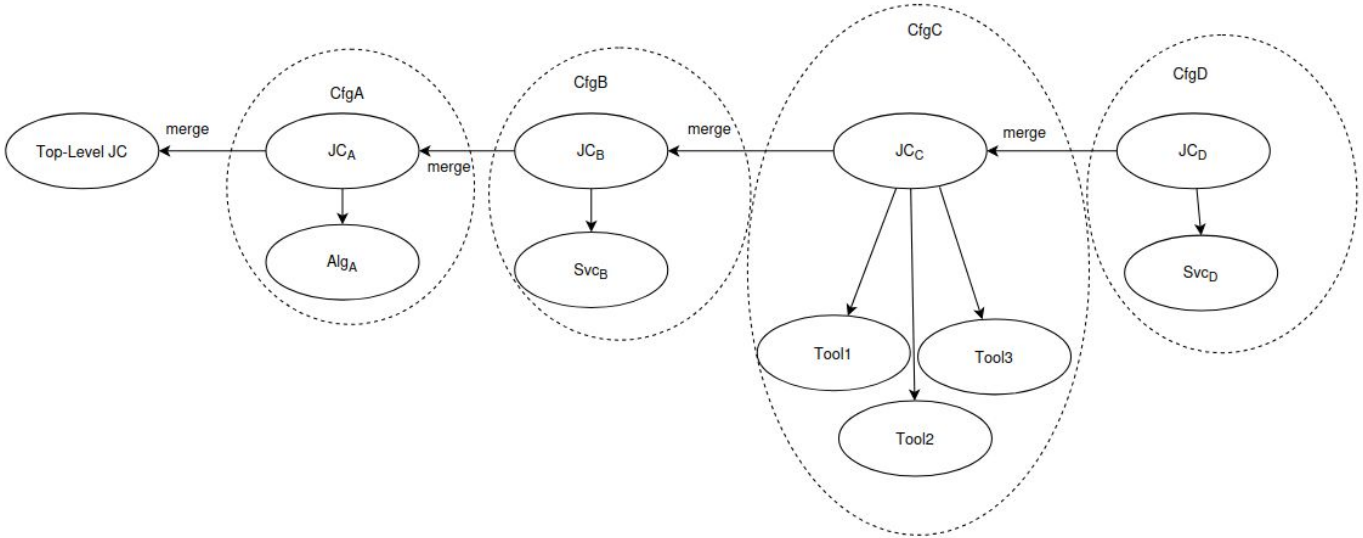
In Sampo Job is configured by means of 3 entities: config, Cfg-functions and JobConfigurators.

- JobConfigurator collects all configured components. One JobConfigurator instance can be merged into another
- config collects all settings of the Job (strict types and names)
- Cfg-functions accept config as a parameter and based on it's collected settings perform component configuring, put configured component and it's dependencies into JobConfigurator and return it



JobConfigurator

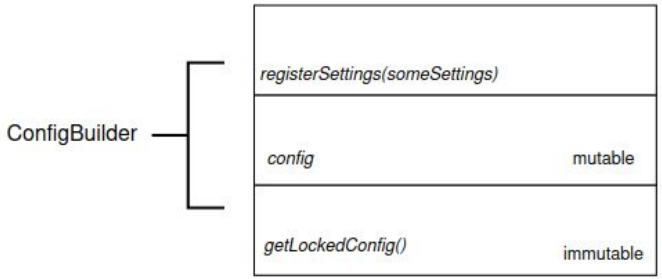
- *addService(service, primary=False, create=False)*
- *addAlgorithm(algorithm, primary=False, sequenceNames=None)*
- *addPrivateTool(tool), addPrivateToolsArray(tools)*
- *merge(other, destinationSequenceName=None)*
- *run()*



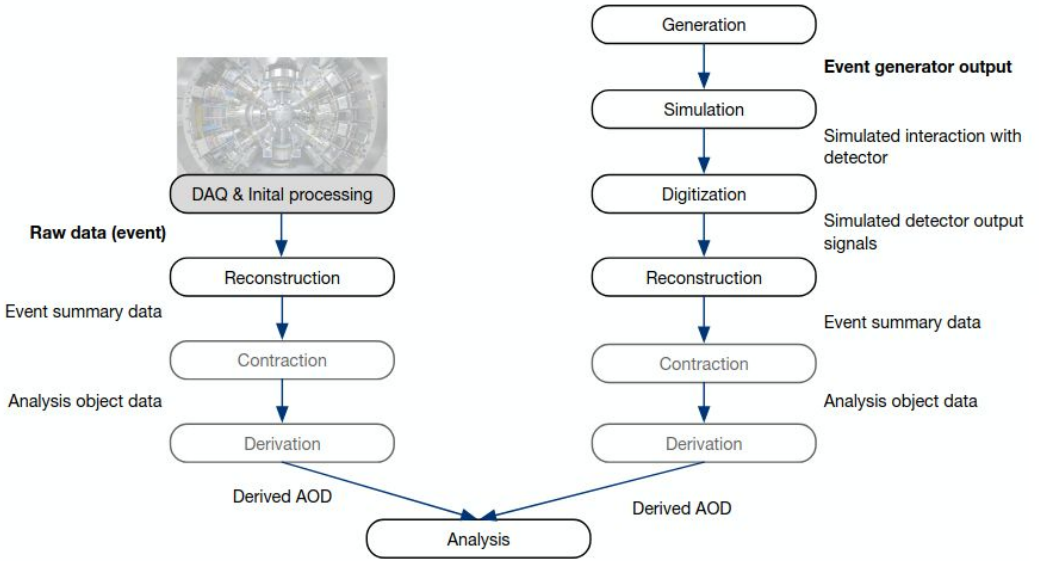
ConfigBuilder

ConfigBuilder is used to collect all possible settings in a single config object:

- register job settings in ConfigBuilder with *`registerSettings(datacls, domain=None)`*
- build config with *`buildConfig()`*. Here CLI-specified settings will override default ones
- access built config as *`ConfigBuilder.config`* to make modifications
- get immutable config with *`getLockedConfig()`*



Production stages



gen.py

sim.py

dig.py

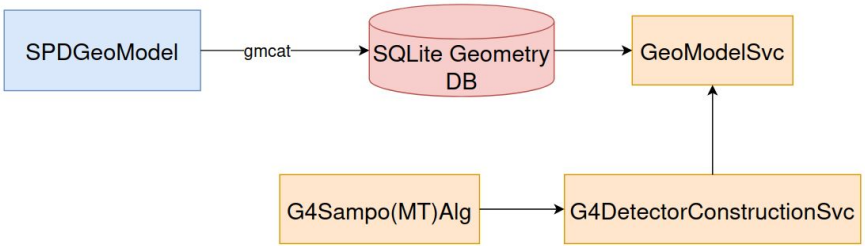
Generator
Input: none
Output: HepMC::GenEvent

Detector simulation
Input: HepMC::GenEvent
HepMC::GenEvent→ McParticle
Output: [McParticle] [McHit] [EventHeader]

Digitization
Input: [McParticle] [McHit][EventHeader]
Output: [McParticle] [McHit] [Digi][EventHeader]

Detector description in Sampo

- Sampo uses GeoModel for detector geometry description. Geometry is described in separate SPDGeoModel project
- Direct access to raw geometry is prohibited, one must use dedicated *GeometryManagers* instead
- Each sensitive volume has it's own unique 32-bit id (DetectorID)




```
class ECalGeoManager {
public:
    friend class GeoModelECalGeoManagerInitTool;
    ECalGeoManager(const ECalGeoManager&) = delete;
    ECalGeoManager(ECalGeoManager&&) = delete;
    ECalGeoManager& operator=(const ECalGeoManager&) = delete;
    ECalGeoManager& operator=(ECalGeoManager&&) = delete;

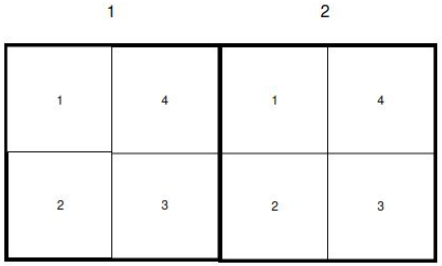
    static ECalGeoManager* Instance();

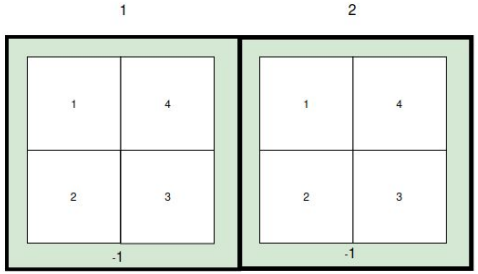
    double GetBarrelCellLenght() const;
    double GetBarrelCellLayerLenght() const;
    double GetEndcapCellLength() const;
    double GetEndcapCellLayerLenght() const;
    int GetEndcapMaxX() const;
    int GetEndcapMaxY() const;
    int GetBarrelMaxZ() const;
    int GetBarrelMaxSector() const;
    int GetBarrelMaxAzimutalNumber() const;

    const CLHEP::Hep3Vector& GetCellPosition(uint32_t id) const;
```

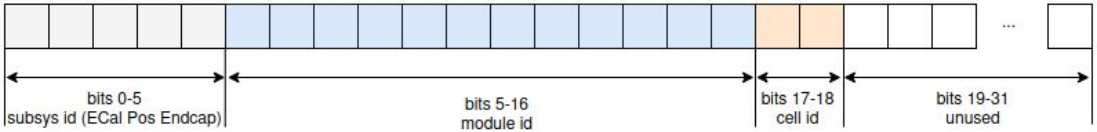
Detector ID

- All Detector IDs are 32-bit unsigned integers (*uint32_t*). Those IDs are composed based on path in geometry graph
- For volumes that are not part of ID building procedure copy number must be set to -1
- IDs can be decoded to structures
- For each subsystem encoding/decoding procedure is different

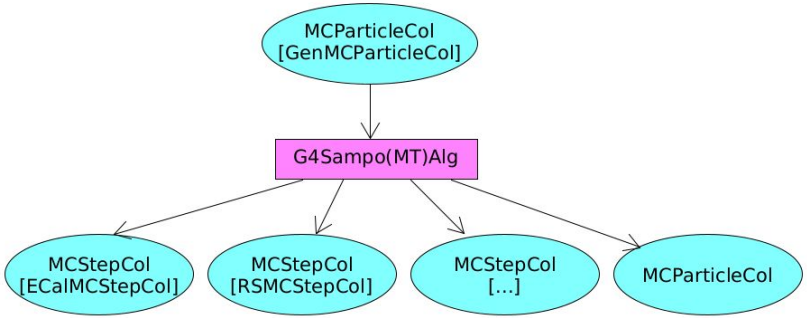


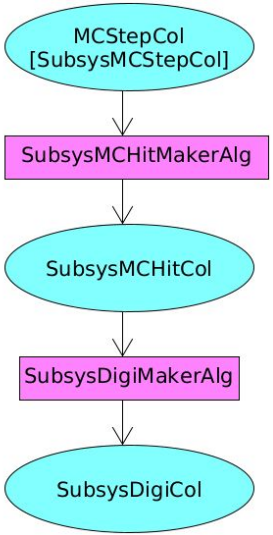


ECal Endcap ID example



Simulation and digitization in Sampo





- MCStep is a particle propagation quantum. It keeps coordinates, momentum, time, track id, detector id and some other parameters.
- MCHit is individual for each subsystem and is constructed based on data stored in MCSteps. For Straw Tracker case (just example) it can contain drift time and charge.
- Digi is produced from SubSysHits. Digis contain data which comes from real detector (smeared by the readout electronics and possible pile-ups)

“Interesting” Particles (SimParticles)

Track is considered to be interesting if:

1) it is primary
2) it is a product of decay process
3) it is product of gamma conversion

If track is selected as interesting then new MCParticle is created and it’s most recent “interesting” ancestor is marked as it’s parent. Parent particle id for primary particles is -1. Each MCHitskeeps id of “interesting” particle, which (or not “interesting” children of which) produced it.

