Table of Contents

6.3 How to Configure and Run Detector Simulation and Digitization

Goals of this page:

The purpose of this document is to quickly enable the CMSSW framework users to produce simulated event samples that can be used for physics studies.

You'll learn:

- What steps and software components are involved in the detector simulation
- How to run centrally-provided configurations
- How to make some simple customizations according to your needs

Contents

Introduction

This document **continues** from the WorkBook Event Generation page, and focuses on the steps that compose **modeling of the Interaction Region (IR) and full-scale CMS Detector simulation**.

By default, the nominal vertex of a generated event is (0,0,0). However, in the real-life the IR is not point-like, thus a specialized software, also called Vertex Smearing, has been introduced to model the spread.

The full-scale simulation of the CMS detector is based on the Geant4 toolkit. It relies on a fairly detailed description of the hierarchy of volumes and materials, and knowing which parts are "sensitive detector" (i.e., furnished with a readout) as opposed to "dead materials". This step follows immediately after the Vertex Smearing.

The next step in the process is reproducing the response of the detector readout electronics, which is also known as Digitization.

More **detailed information** on the software, including description of algorithms where applicable, is given in the SWGuideSimulation (WARNING: materials under development !).

As with all CMSSW components, the software involved in the detector simulation process is configurable at run time.

In this document, we will offer a walk-through of a pre-fabricated example configuration application, which is composed of the use of **standard**, CMS-approved settings. We'll point you to fragments and modules that are related to the IR modeling and detector simulation. We'll also provide some tips on how to change some configuration parameters, in case your study calls for custom configuration rather than the use of default settings.

In general, we assume the use of the CMSSW_10_2_X cycle. However, many tips are still valid for many earlier releases. We'll provide additional, release-specific tips where necessary.

Simulation Software and Existing Applications

The software that is related to the IR modeling and detector simulation, including related components, is quite extensive and resides in several subsystems: IOMC, SimGeneral, SimG4Core, SimG4CMS, SimTracker, SimCalo, SimMuon.

The simulation part in the event processing chain is performed by several CMSSW modules, that need to be composed in the specific order in the configuration application, as each module in the chain will require certain output from an earlier step, performed by another module. Also, these modules need to be assigned certain labels, because it that by the label that each module will access necessary information delivered into `edm::Event`.

In the earlier section of this chapter, WorkBookGenIntro, we have already pointed at the existing pre-fabricated example, and have provided brief information about its "building blocks".
In this document we offer more details about services and steps involved in the detector simulation procedure.

Instructions are also provided on how one can configure such applications, starting from a generator-level configuration fragment and using CMSSW utilities and standard settings.

In addition, we will explain how to change some of the standard settings, as we presume some tasks may need that.

How to Find and Run an Example Simulation Application

Although we have already explained in the earlier materials how to setup the environment for running an application, we repeat it here for clarity:

```
cmsrel CMSSW_X_Y_Z
cd CMSSW_X_Y_Z/src
cmsenv
```

step0:GEN-SIM

```
cmsDriver.py
Configuration/GenProduction/python/ThirteenTeV/RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_py
--fileout file:RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_pythia8_GEN-SIM.root -s
GEN,SIM --mc --datatier GEN-SIM --beamspot Realistic25ns13TeV2016Collision --conditions
auto:phase1_2017_realistic --eventcontent RAWSIM --era Run2_2017 --python_filename
RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_pythia8_GEN-SIM_cfg.py --no_exec -n 50
```

step1:DIGI2RAW-HLT

```
cmsDriver.py -s DIGI,L1,DIGI2RAW,HLT --datatier GEN-SIM-RAW --conditions
auto:phase1_2017_realistic --eventcontent RAWSIM --era Run2_2017 --filein
file:RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_pythia8_GEN-SIM.root --fileout
file:RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_pythia8_GEN-SIM-RAW.root
--python_filename
RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_pythia8_FULLSIM_GEN_SIM_RAW_cfg.py -n -1
--no_exec
```

step2:RECO

```
cmsDriver.py -s RAW2DIGI,L1Reco,RECO --datatier RECO --conditions
auto:phase1_2017_realistic --eventcontent AODSIM --era Run2_2017 --filein
file:RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_pythia8_GEN-SIM-RAW.root --fileout
file:RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_pythia8_GEN-SIM-RAW-RECO.root
--python_filename
RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_pythia8_FULLSIM_GEN_SIM_RAW_RECO_cfg.py -n
-1 --no_exec
```

How to Find and Run an Example Simulation Application

Although we have already explained in the earlier materials how to setup the environment for running an application, we repeat it here for clarity:

```
cmsrel CMSSW_X_Y_Z
cd CMSSW_X_Y_Z/src
cmsenv
```

step0:GEN-SIM

```
cmsDriver.py Configuration/GenProduction/python/BPH-RunIIFall18GS-00170-fragment.py
--fileout file:BPH-RunIIFall18GS.root --mc --eventcontent RAWSIM --datatier GEN-SIM
--conditions 102X_upgrade2018_realistic_v11 --beamspot
Realistic25ns13TeVEarly2018Collision --step GEN,SIM --geometry DB:Extended --era Run2_2018
--python_filename BPH-RunIIFall18GS_cfg.py --no_exec --customise
Configuration/DataProcessing/Utils.addMonitoring --customise_commands
process.source.numberEventsInLuminosityBlock="cms.untracked.uint32(1408450)" -n 10000
```

step1:DIGI2RAW-HLT

voms-proxy-init

```
cmsDriver.py step1 --filein file:BPH-RunIIFall18GS.root --fileout
file:BPH-RunIIAutumn18DR_step1.root --pileup_input
"dbs:/MinBias_TuneCP5_13TeV-pythia8/RunIIFall18GS-102X_upgrade2018_realistic_v9-v1/GEN-SIM"
--mc --eventcontent FEVTDEBUGHLT --pileup "AVE_25_BX_25ns,{'N': 20}" --datatier
GEN-SIM-DIGI-RAW --conditions 102X_upgrade2018_realistic_v15 --step
DIGI,L1,DIGI2RAW,HLT:@relval2018 --nThreads 8 --geometry DB:Extended --era Run2_2018
--python_filename BPH-RunIIAutumn18DR_cfg.py --no_exec --customise
Configuration/DataProcessing/Utils.addMonitoring -n 100
```

step2:RECO

```
cmsDriver.py step2 --filein file:BPH-RunIIAutumn18DR_step1.root --fileout
file:BPH-RunIIAutumn18DR_step2.root --mc --eventcontent AODSIM --runUnscheduled --datatier
AODSIM --conditions 102X_upgrade2018_realistic_v15 --step RAW2DIGI,L1Reco,RECO,RECOSIM,EI
--nThreads 8 --geometry DB:Extended --era Run2_2018 --python_filename
BPH-RunIIAutumn18DR_2_cfg.py --no_exec --customise
Configuration/DataProcessing/Utils.addMonitoring -n 100
```

How to Compose a Simulation Application Using Standard Utilities

In most cases, simulation applications will only differ by the topology of the physics events want wants to study, i.e. the differences are at the Generator level only, while steps related to IR modeling and detector simulation can be added via use of pre-fabricated, **standard** (CMS-approved) configuration fragments.
The most reliable and safe way of composing a **standard** simulation chain is with the use of the `cmsDriver.py` utility. This utility is distributed together with CMSSW releases, and will automatically appear in your PATH upon setting up CMSSW run time environment, i.e. execution of `cmsenv` script (see above). It produces a framework .cfg file that you can run with `cmsRun`.

Detailed information on the `cmsDriver.py` is provided in the dedicated documentation. We have also offered some tips and shortcuts in the earlier part of this chapter. There are also up-to-date cmsDriver statements inside of the files contained in the `Configuration/PyReleaseValidation/data` directory; these can be used as templates for making your own statements.

Obviously, before executing `cmsDriver.py` a user must choose a configuration fragment that will define the event topology to be generated. In the Event Generation part of this chapter, we point users to a large collection of pre-fabricated generator level fragments one can use. In case a configuration for a topology of the user's interest is not readily available within CMSSW, we also offer there a variety of instructions and pointers to related materials, to help users compose their own applications.

Once you have selected your generator level configuration fragment, you can compose your application like this:

```
cmsDriver.py PATH/To/Your/GenFragment.py -s GEN,SIM --mc --datatier GEN-SIM --beamspot
Realistic25ns13TeVEarly2018Collision --conditions 102X_upgrade2018_realistic_v11
--eventcontent RAWSIM --era Run2_2018 --no_exec -n 50
```

Of course, in the example above `PATH/To/Your/GenFragment.py` is a generic form. If you place your generator-level configuration fragment into `Coonfiguration/Generator/python` directory, you will only need to specify the file name.

i.e. you can omit `Configuration/Generator/python` portion, as it is default in `cmsDriver.py`.
But if you place your generator level fragment into another package under your local CMSSW area, you will need to specify more details, i.e. provide it in the format that can be schematically described as `Subsystem/Package/python/YourFile`.

Walk-through a Configuration File: Simulation Related Steps and Components

In the earlier section of this chapter we have already pointed users to some of the top-level "building blocks" of the detector simulation application.
In this document we would like to provide more information, including some of the details on the lower-level fragments the "building blocks" are made of.

Random Number Engine and Seed Settings

All modules that are involved in the IR modeling and detector simulation need to be given an initial random seed and, optionally, an engine.

In CMSSW all modules that need a random seed pick it up from a single *Service* called RandomNumberGeneratorService.

The complete configuration of the service is defined in the **standard** configuration include: IOMC/RandomEngine/python/IOMC_cff.py.

This gets included in the application configuration via this statement that you can see in the config file created in the step 1.

```
process.load('Configuration/StandardSequences/Services_cff')
```

which in turn imports the standard settings:

```
from IOMC.RandomEngine.IOMC_cff import *
```

Please note that when you use `cmsDriver.py` to compose your application, the `Services_cff.py` will be included by default.

Later in this document, we will also show how one can alter the default seed for one or another module if this is needed.

Add Vertex Smearing

As already mentioned above, the vertex smearing software allows accounting for the actual dimensions of the IR, which is not point-like at (0,0,0). The software is located in the package IOMC/EventVertexGenerators. We have modules for modeling several hypothesis of the IR spread. Currently, we assume that the most realistic model is based on the beta-function smearing. For this option, we have defined several beam configuration scenarios according to the latest LHC simulations.

This configuration is included in the example application via this statement:

```
process.load("Configuration.StandardSequences.VtxSmearedRealistic25ns13TeVEarly2018Collision_cfi"
```

which in turn imports the standard settings:

```
from IOMC.EventVertexGenerators.VtxSmearedRealistic25ns13TeV2017Collision_cfi import *
```

Users are welcome to examine the settings: IOMC/EventVertexGenerators/python/VtxSmearedRealistic25ns13TeVEarly2018Collision_cfi.py.

Other options and various details are described in the dedicated Vertex Smearing guide.

The IR modeling operates on the generator particles. Obviously, this needs to be applied before the detector simulation can come into play.

Users should bear in mind that a **single label "VtxSmeared"** should be used in all production applications, regardless of what IR modeling module a user decides to employ. It is by this label that the IR modeling step will be included in the event processing chain.

Please note that even if we consider IR modeling together with the detector simulation procedure, the `cmsDriver.py` utility includes it via the `GEN:ProductionFilterSequence` argument in the `-s` field (together with it also get included several other modules of the Physics Analysis domain, that operate on generator's particles).

Geometry and Magnetic Field

In order to run, the CMS detector simulation application requires 2 common-use components - descriptions of the Detector Geometry and of the Magnetic Field. Both are service-type components, i.e. they are not steps in the event processing chain, but they deliver mandatory information that is necessary for the detector simulation software.

The detector simulation is loaded via the following instruction:

```
process.load("Configuration.StandardSequences.Geometry_cff")
```

This is the configuration that should be used in the production applications.
In a private configuration, one can replace the official cfi/cff file for a specific studies.
However, we suggest that you modify the official complete geometry with your private geometry only if you know what you are doing!

Another mandatory inclusion is the magnetic field record:

```
process.load("Configuration.StandardSequences.MagneticField_cff")
```

Please note that the magnetic field record must be present for the proper initialization of your Geant4-based detector simulation job, due to how the hierarchy of volumes with/without magnetic field is coded.
Later in this document we will explain how to switch off the magnetic field, if this is needed for a specific study.

If you use `cmsDriver.py` utility to compose your simulation application, both standard configurations will be loaded by default.

Detector Simulation Module : *OscarProducer*

The full-scale simulation of the CMS detector is based on the Geant4 toolkit. It relies on a fairly detailed description of the hierarchy of volumes and materials, and knowing which parts are "sensitive detector" (i.e., furnished with a readout) as opposed to "dead materials".
It takes as input generated particles (vertex smearing included), traces them through the hierarchy of volumes and materials, and models physics processes that accompany particle passage through matter. Results of each particle's interactions with matter are recorded in the form of ***simulated hits***. An example of a simulated hit can be energy loss by a given particle within a "sensitive volume" of one of the subdetectors, stored along with several other characteristics of the interaction. Particles can be either "primary" (generated particle) or "secondary" (a result of Geant4-modeled interactions of a primary particle with matter).

The software is quite extensive and is located in the SimG4Core and https://github.com/cms-sw/cmssw/tree/CMSSW_10_2_X/SimG4CMS subsystems.
Altogether it is composed in a single module called *OscarProducer*.

The top-level "building block" that brings detector simulation into the event processing chain is Configuration/StandardSequences/python/Sim_cff.py. The detector simulation is loaded via the following instruction:

```
process.load('Configuration/StandardSequences/SimIdeal_cff')
```

It brings in an intermediate level:

```
from SimG4Core.Configuration.SimG4Core_cff import *
```

which in turn imports the actual settings of the *OscarProducer*:

```
from SimG4Core.Application.g4SimHits_cfi import *
```

These are the **standard**, CMS-approved settings, and users are welcome to examine them: SimG4Core/Application/python/g4SimHits_cfi.py.
Later in this document we will show how to change some of the settings, that we think could be of interest to end-users.

The module is labeled **"g4SimHits"**, and is included in the event processing chain by this label. It is important to remember that this label is mandatory in the production applications because the subsequent components will need it to look up for the simulation output in the `edm::Event`.

The inclusion of this software in the event processing chain corresponds to the `SIM` argument in the `cmsDriver.py -s` field.

More details can be found in the SWGuideSimulation and its daughter materials.

Modeling Detector Response: *Digitization*

The CMS detector simulation process is wrapped up by modeling the response of detector readout electronics. Corresponding to the CMS detector structure, the software is subdivided into 3 domains: SimTracker, SimCalorimetry, and SimMuon. In turn, each subsystem is composed of several subdetectors, thus the software is a collection of several independent modules.
However, all these modules have one thing in common: they all must operate AFTER the module that simulates the PileUp.
We are providing more a few more tips below; detailed information can be found in SWGuideSimulation (WARNING: materials under constructions !).

As already pointed out earlier, the digitization is collectively brought into the config file via this statement:

```
process.load('Configuration/StandardSequences/Digi_cff')
```

The whole sequence of steps is collectively labeled **"pdigi"**; it is by this label the digitization gets included in the production chain.

Collectively, the inclusion of this software in the event processing chain corresponds to the `DIGI` argument in the `cmsDriver.py -s` field.

Below we'll provide a few more details on what this top-level fragment Configuration/StandardSequences/python/Digi_cff.py is composed of.

PileUp Simulation: *MixingModule*

PileUp simulation relates to the effect of more that one physics interaction per beam crossing, due to high concentration of particles in a bunch, and/or the effect of the electronics signal spill-over from the previous bunch crossings (may be >1) into the current crossing (event), due to the fact that the bunch spacing (time between collisions) is often shorter than the duration of an electronic signal in one or another subdetector.

The PileUp can be **optionally** simulated by the CMSSW component called *MixingModule*. The software is located in the https://github.com/cms-sw/cmssw/tree/CMSSW_10_2_X/SimGeneral/MixingModule][SimGeneral/MixingModule]] package.

Detailed information on the *MixingModule* can be found in the MixingModule manual.

As already stated above, the *MixingModule* must be present in the chain, at least in the zero-pileup mode, if a user needs to digitize simulated hits in the CMS detector subsystems. This is due to the implementation details of the digitization software. In the config file it is included via the following statement:

```
process.load('Configuration/StandardSequences/MixingNoPileUp_cff')
```

Additional pre-fabricated configuration fragments for that users can use in their production applications for the PileUp simulation is the following:

Configuration/StandardSequences/python/Mixing.py

All pileup scenarios available for FullSim are automatically available for FastSim.Pileup scenarios are defined in **`Configuration/StandardSequences/python/Mixing.py`**. For each available pileup option **`X`** there is a line **`addMixingScenario("X","Y")`**, where **`Y`** points to the configuration file or provides parameters that define the pileup scenario.==addMixingScenario("FS_X","Y")==, where **`Y`** points to the configuration file or provides parameters that define the pileup scenario.See PdmVPileUpDescription to find out which PU scenario is used in which production campaign.

If you browse through the fragments, you'll notice that, in turn, they import **standard**, CMS-approved configurations of the *MixingModule* from the package area.
We **strongly recommend** that you use these fragments, rather than configure *MixingModule* explicitly. If you wish to modify any parameter, please use the **replace** option.

Coming soon: Samples generated with the DataMixingModule, which can overlay collision data on simulated events. Extensive documentation can be found in the Offline Simulation guide SWGuideSimulation.

Tracker Digitization

Tracker digitization software belongs to the SimTracker subsystem of CMSSW.
It is imported into Digi_cff.py via the following statement:

```
from SimTracker.Configuration.SimTracker_cff import *
```

(collectively labeled as "trDigi"). It is composed of 2 fairly independent steps: Pixel digitization and silicon strips digitization. You can find more information in the SWGuideSimulation and its daughter materials (under construction).

Calorimeter Digitization

Calorimetry digitization software belongs to the SimSimCalorimetry subsystem of CMSSW.
It is imported into Digi_cff.py via the following statement:

```
from SimCalorimetry.Configuration.SimCalorimetry_cff import *
```

(collectively labeled as "calDigi"). It made of 2 independent steps: Electromagnetic Calorimeter (ECAL) digitization and Hadronic Calorimeter (HCAL) digitization. You can find more information in the SWGuideSimulation and its daughter materials (under construction).

Muon System Digitization

Muon system digitization software belongs to the SimMuon subsystem of CMSSW.
It is imported into Digi_cff.py via the following statement:

```
from SimMuon.Configuration.SimMuon_cff import *
```

(collectively labeled as "muonDigi"). It is composed of 3 steps: digitization of the CSC, DT, and RPC subdetectors. You can find more information in the SWGuideSimulation and its daughter materials (under construction).

Output Data Formats

Of many `edm::Event` data products, only two branches are of general interest to most of the end users: container of simulated Track objects and SimDataFormat/Vertex packages, (see also

SimDataFormats/Track/interface/SimTrackContainer.h and
SimDataFormats/Vertex/interface/SimVertexContainer.h).

Both branches are **labeled "g4SimHits"**.

Please note that, in the official production, these branches are kept in certain samples but not in all of them.

You may find additional details in the SWGuideDataFormatSimG4Core document.

Other `edm::Event` branches produced by various modules involved in the detector simulation chain typically serve as input for the subsequent steps and are not of interest to the majority of end-users. They are excluded from the output in the official production. Additional information on configuring output, if needed for specific studies, are provided later in this document.

Additional How-To's

Change Some of the OscarProducer's Parameters

The default (standard CMS-approved) configuration of the *ParameterSet* for the *OscarProducer* module is given in the following configuration include:
SimG4Core/Application/python/g4SimHits_cfi.py

A user can modify/replace the default parameters if a specific task calls for such change. However, we suggest that you make changes only if you're really familiar with the software, and you will know what you do.

If you decide to change some of the parameters, you can place it it fairly anywhere in your configuration, as long as it's done **after** the inclusion of

```
process.load("Configuration.StandardSequences.Simulation_cff")
```

Detailed information about *OscarProducer* configuration parameters can be found in SWGuideSimulation and its daughter materials.

Here we would like to point at several parameters that we believe are of primary interest for the end-user:

```
      ...
     G4EventManagerVerbosity = cms.untracked.int32(0),
    G4StackManagerVerbosity = cms.untracked.int32(0),
    G4TrackingManagerVerbosity = cms.untracked.int32(0),
    UseMagneticField = cms.bool(True)
      ...
```

The first three (`untracked ...`) will determine the amount of information printed out while Geant4 performs tracing of each particle, primary or secondary. Turning them on may be useful for testing/debugging purposes; for a generic simulation job, they need to stay off (0).

The fourth parameter in the list (`bool ...`) allows you to turn the magnetic field on/off, depending on the type of the simulation study.

If you use standard cfi but you wish to modify, for example, the magnetic field switch, you can do it like this:

```
process.g4SimHits.UseMagneticField = False # this is your "replace" statement
```

Modeling of the physics processes that happen along the particle's path can be chosen via this subset of the configuration cards:

```
       Physics = cms.PSet(
        .....
        # NOTE : if you want EM Physics only,
        #        please select "SimG4Core/Physics/DummyPhysics" for type
        #        and turn ON DummyEMPhysics
        #
        type = cms.string('SimG4Core/Physics/QGSP_BERT_EML'),
        DummyEMPhysics = cms.bool(False),
        CutsPerRegion = cms.bool(True),
        DefaultCutValue = cms.double(1.0), ## cuts in cm
        G4BremsstrahlungThreshold = cms.double(0.5), ## cut in GeV
        Verbosity = cms.untracked.int32(0),
        # 1 will print cuts as they get set from DD
        # 2 will do as 1 + will dump Geant4 table of cuts
        .......
      )
```

Here the QGSP_BERT_EML is the physics list based on the Bertini cascade model (in the intermediate energy range) and on the quark-gluon string model (on the high energy end). Other available physics lists can be, for example, LHEP (formerly known as Gheisha code) that is extensively based on the parametrization of the experimental data for calorimeters. For testing purposes, some may employ so-called DummyPhysics lists that only know how to treat e- and mu- through matter.

For example, if you wish to select LHEP physics list over the default one (QGSP_BERT_EML), you can add the following statement:

```
   process.g4SimHits.Physics.type = "SimG4Core/Physics/LHEP"
```

If you wish to select for Geant4 tracing to be done for only certain generated particles in a specific kinematic region, you can apply the cuts to the generator's input; its default configuration is shown below:

```
     Generator = cms.PSet(
        HectorEtaCut,
        HepMCProductLabel = cms.string('generator'),
        ApplyPCuts = cms.bool(True),
        MinPCut = cms.double(0.04), ## the pt-cut is in GeV (CMS conventions)
        MaxPCut = cms.double(99999.0), ## the ptmax=99.TeV in this case
        ApplyEtaCuts = cms.bool(True),
        MinEtaCut = cms.double(-5.5),
        MaxEtaCut = cms.double(5.5),
        ApplyPhiCuts = cms.bool(False),
        MinPhiCut = cms.double(-3.14159265359), ## in radians
        MaxPhiCut = cms.double(3.14159265359), ## according to CMS conventions
        RDecLenCut = cms.double(2.9), ## the minimum decay length in cm (!) for mother tracking
        Verbosity = cms.untracked.int32(0)
    )
```

For example, if you wish to narrow the eta-range of interest from the default -/+5.5 to -/+1.5, and raise the minimum P cut from 40MeV to 1GeV, you can do so by adding the following statements:

```
   process.g4SimHits.Generator.MinEtaCut = -1.5
   process.g4SimHits.Generator.MaxEtaCut = 1.5
   process.g4SimHits.Generator.MinPCut = 1. # in GeV
```

Configure Simulation Messages in the MessageLogger

At runtime, the level of verbosity of the simulation code can be controlled through a general service of the framework called *MessageLogger*. Different levels of verbosity (*DEBUG*, *INFO*, *WARNING*, *ERROR*) can be selected by the user, as well as output streams to which to send different messages (*cout*, *cerr*, external files of user's choice). A minimalist example of use of *MessageLogger* is given in the detector simulation configuration files, suppressing essentially all the messages (a limit on the number of messages can be set, and this limit can depend on the category of the messages):

```
process.MessageLogger = cms.Service("MessageLogger",
    cout = cms.untracked.PSet(
        default = cms.untracked.PSet(
            limit = cms.untracked.int32(0) ## kill all messages in the log
        ),
        FwkJob = cms.untracked.PSet(
            limit = cms.untracked.int32(-1) ## but FwkJob category - those unlimited
        ),
        SimG4CoreApplication = cms.untracked.PSet(
            limit = cms.untracked.int32(-1)
        )
    ),
    categories = cms.untracked.vstring('FwkJob', 'SimG4CoreApplication'),
    destinations = cms.untracked.vstring('cout')
)
```

For a detailed explanation of all the features please see SWGuideMessageLogger.

Configure Event Output

Event output, which is is performed the *PoolOutputModule* module into a ROOT, may contain all persistent objects produced and stored in the event, or a selected subset of branches.

As already pointed out earlier, with the use of `cmsDriver.py` utility the output is formed via the `--eventcontent` field. For most production applications the subset called **RAWSIM** is used. Detailed description of the data formats/contents adopted in CMS is available in the Data Format Table. If you browse through config file from step 1, we will notice this statement:

```
process.load('Configuration/EventContent/EventContent_cff')
```

which brings in the top-level configuration fragment where various b> subsets are "declared", although the actual "definitions" of what branches are included in one or another subset are distributed over many packages.
Once you specify to the `cmsDriver.py` which of the **standard** event content you desire, it'll automatically compose *PoolOutputModule* for your application.
In our particular example you may notice the `outputCommands` parameter it in the configuration:

```
process.RAWSIMoutput = cms.OutputModule("PoolOutputModule",
...
fileName = cms.untracked.string('file:RSGravitonToZZ_kMpl01_M_1000_TuneCUETP8M1_13TeV_pythia8_GEN
outputCommands = process.RAWSIMEventContent.outputCommands,
...
)
```

Note that if the `outputCommands` is not provided, *PoolOutputModule* will write out the entire event content, which can be of a very large size.

In case your task calls for it, you can manually configure *PoolOutputModule* to write out only branches of interest to you; detailed information is provided in the SWGuideSelectingBranchesForOutput.
However, we would like to warn that you should understand very well what modules add one or another branch to the `edm::Event` and how those branches are labeled.

Reproduce Simulated Events

Coming shortly

View the Simulated Events

The output of the simulation job is a ROOT file, and it can be easily browsed by the standard ROOT browser. But of course, for a more sophisticated analysis, you have to rely either on ROOT macros that load `libFWCoreFWLite`, e.g.,

```
  gSystem->Load("libFWCoreFWLite");
  AutoLibraryLoader::enable() ;
```

or on the full power of the framework by writing your own **EDAnalyzer** module.

Detailed information is provided in the chapter 4 of the WorkBook, dedicated to Analysis and related materials.

Further Documentation

For general information on the CMSSW framework see WorkBookCMSSWFramework for an introduction.

Many useful links can be found:

- http://cms.cern.ch/iCMS/jsp/page.jsp?mode=cms&action=url&urlkey=CMS_OFFLINE CMS Offline Page
- [[https://github.com/cms-sw/cmssw/tree/CMSSW_10_2_X] *WebGIT* allows to browse the code
- http://cmsdoc.cern.ch/cms/cpt/Software/html/General/gendoxy-doc.php *Doxygen* provides a basic documentation of the code itself

The simulation development information is given in the *hypernews* forum:

https://hypernews.cern.ch/HyperNews/CMS/get/simDevelopment.html

e-mail address: *hn-simDevelopment@cern.ch*

Questions related to the simulation should be sent to this forum.

Review status

Reviewer/Editor and Date (copy from screen)	Comments
ElizaMelo - Dec 2019	Major update towards CMSSW_10_2_X
ElizaMelo - 30 Aug 2017	Major update towards CMSSW_9_X_X
JuliaYarba - 24 Aug 2009	Major update towards CMSSW_3_X_X
JuliaYarba, Main.Anne Heavey - 15 Jun 2006	Update it for Jun 06 tutorial
FabioCossutti - 08 May 2006	wrote original tutorial for May 2006 CPT week

Responsible: JuliaYarba
Last reviewed by: FilippoAmbroglini - 26 Feb 2008

This topic: CMSPublic > WorkBookSimDigi
Topic revision: r93 - 2019-12-27 - unknown