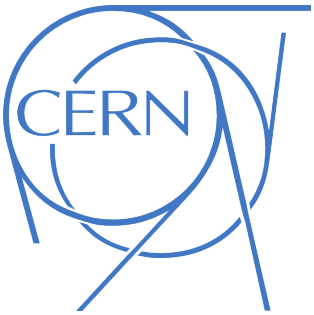

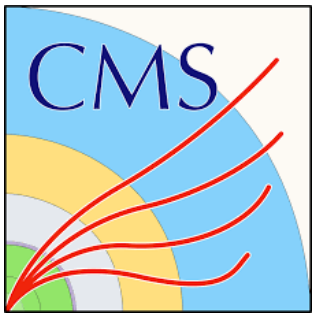

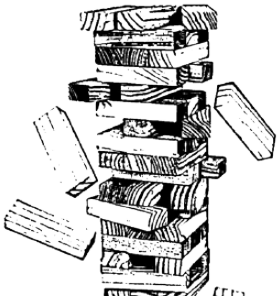

CMSSW Config Builder with LAW

Yusif Askari

August 9, 2024

Contents

Chapter 1

Introduction

1.1 Project Overview

During my summer school at CERN, I was tasked with automating the creation of CMSSW [1] (CMS Software) configuration files. Given the complexity and repetitive nature of manually generating these files, an automated solution was necessary to streamline the process. My project involved developing a Python-based system that leverages LAW (Luigi Analysis Workflows) for task management, Pydantic [2] for configuration validation, and multiple layers of caching to improve efficiency and reusability.

1.2 Importance of Automation in CMSSW

CMSSW is a large and complex framework used in the CMS experiment at CERN to simulate, perform the event reconstruction and analyze particle collisions. The configuration of CMSSW involves setting up various parameters and workflows, which can be time-consuming and prone to human error if done manually. This is a critical task, and it is often needed to produce a large amount of data samples to perform physics studies. One crucial task is to generate simulated events that can be analyzed to evaluate the performance of the Event Software Reconstruction. Producing data samples can be computational and time-expensive since the simulated events can be very complex. For this reason, the usage of tools like HTCondor or Cms Remote Analysis Builder (CRAB) to exploit all the computing resources available is mandatory. The event generation is often repetitive and it need few changes in the job configuration, this is why this task can be easily pipelined to reduce errors and to save time in preparing all the configurations.

1.3 Manual Configuration of CMSSW

To manually configure and run a CMSSW workflow, follow these steps:

1. **Set Up CMSSW Environment:** Initialize the CMSSW environment by creating a new release area, navigating to the source directory, and setting up the environment.

   ```
   cmsrel CMSSW_XX_XX_X
   cd CMSSW_XX_XX_X/src
   cmsenv
   ```

2. **List and Choose Workflow:** Use `runTheMatrix.py` to list available workflows and select the one that matches your parameters. (`era`, `pileup`, `type` ...)

   ```
   runTheMatrix.py -w upgrade -n
   ```

3. **Run Selected Workflow:** Execute the chosen workflow with the selected ID. Replace <`workflow_id`> with the actual workflow ID.

   ```
   runTheMatrix.py -w upgrade -l <workflow_id> -j 0
   ```

4. **Modify Configuration Files:** Edit the `step*.py` files as needed to customize the workflow steps. The step files are python configuration file that schedule the process to run: physics process generation, particle interaction with the detector (Geant4), conversion from RAW data format to DIGI data format, High Level Trigger reconstruction, and full event reconstruction. The configuration of each step is the most time-consuming as there is already a set of parameters that need to be tested and would require manual modifications.

5. **Run Specific Steps:** Execute the steps by running the modified scripts.

   ```
   cmsRun stepX.py
   ```

Chapter 2

Project Setup

2.1 Tools and Technologies

This project leverages several key tools and technologies to facilitate efficient workflow management and data validation:

- **LAW (Luigi Analysis Workflows)**: LAW is a framework designed to streamline the execution and management of complex workflows, particularly in large-scale data processing environments. It integrates seamlessly with existing batch systems and provides a robust mechanism for managing dependencies, ensuring tasks are executed in the correct order.
- **Pydantic v2**: Pydantic is a data validation and settings management library for Python. It ensures that configurations adhere to specified schemas, enhancing the reliability and robustness of the application by catching configuration errors early.

2.2 Environment Setup

To prepare the environment for this project, follow these steps:

2.2.1 Clone the Repository

Start by cloning the repository that contains the project's code and configuration files:

```
git clone https://github.com/solo-driven/cmssw-configurator.git
cd cmssw-configurator
```

2.2.2 Set Up the Python Environment

It is recommended to use a virtual environment to manage dependencies and avoid conflicts. You can either create a new virtual environment or install the dependencies globally.

Option 1: Create a Virtual Environment

To create and activate a virtual environment, run:

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Option 2: Install Dependencies Directly

Alternatively, install the required packages globally using:

```
pip install -r requirements.txt
```

2.2.3 Configure Environment Variables

To configure the necessary environment variables, modify env vars in `setup.sh` in `cmssw_configurator_project`:

```
# Define essential environment variables

# output the root files generated by CMSRunTask
export ROOTS_DIR=

# store release worflows for future use
export CONFIGURATOR_CACHE_DIR=

# store combinations of config files
export WORKSPACE_DIR=
```

Execute the script to set the environment variables:

```
source setup.sh
```

Then what is left is to provide the `workflow.toml` and run the task you want by passing also the args.

Chapter 3

Code Structure

3.1 Project Code Structure

The project is structured into several key modules, each responsible for different aspects of the CMSSW configuration process:

- **configurator/**: This is the central module where the core logic of the project is implemented.
  - **law_tasks/**: Contains custom tasks designed to work with LAW, including tasks for retrieving CMSSW releases, caching them, and executing `cmsRun`.
  - **schemas/**: Defines Pydantic models used for validating configuration files, ensuring they adhere to the required specifications.
  - **modifiers/**: Contains Python modules named after workflow types. The corresponding module is imported and used to generate code that modifies the step1 configuration file for each specific workflow type. Additionally, this directory includes `steps.py`, which contains the logic for updating the necessary parts of the step1 file.
  - **tests/**: Includes unit tests to validate the Pydantic schemas and ensure the integrity of the configuration process.
  - **workflow_specs_mapper.py**: Maps the specifications from the `workflow.toml` file, enabling `grep` to filter configurations based on these specifications.
- **workflow.toml**: The primary configuration file from which parameters are extracted to modify CMSSW configuration files.

- **law.cfg**: Configuration file for LAW, specifying essential details required for its operation.
- **setup.sh**: A script that sets up the environment by initializing the necessary environment variables for the project.

3.2 Workflow configuration file

The `workflow.toml` file is divided into several sections:

- `[specs]`: Defines general workflow specifications.
- `[parameters]`: Configures workflow-specific parameters.
- `[generator]`: Contains settings for the particle generator.
- `[generator.parameters]`: Detailed configuration for generator parameters.

3.3 Law configuration file

For configuring the number of processes (workers) and other parameters specific to the Law law.cfg [3] and set `LAW_CONFIG_FILE` environt var to point to it.

Chapter 4

Validation with Pydantic

4.1 Overview of Pydantic v2

Pydantic provides robust data validation and settings management by leveraging type annotations. It ensures that data structures conform to defined schemas and constraints, enhancing data integrity and consistency in Python applications.

4.2 Custom Annotations and Validators

Pydantic supports custom annotations and validators to extend default validation capabilities:

- **SingleOrMultiple**: This custom annotation allows a field to accept either a single value or a list of values. It uses the `SingleOrMultiple` class and its associated validator to determine whether to process the input as a single value or a list.
- **Range**: Applied to tuples to ensure they contain exactly two numeric values, where the first value is less than or equal to the second.
- **UniqueTuple**: Ensures that tuples are non-empty and contain unique elements. This prevents duplicate entries and is used for fields like `particle_ids` to ensure distinct sets of particles. This is enforced by the `validate_unique_values` function, which checks for duplicates and handles ordering variations.
- **Particle ID Mapping Validator**: The `get_particle_ids` function maps particle names to their corresponding numerical IDs using predefined mappings. This ensures that particle names provided in configurations are accurately translated to their respective numerical identifiers.

4.3 Pydantic Schemas

Pydantic schemas define the structure and constraints of data, leveraging type hints and validators to ensure data validity:

- **Geometry (Enum)**:
  - **Purpose**: Enumerates possible CMS geometry version with auto-generated values.
  - **Attributes**:
    - `D86` - `D114`: Enum members representing different geometry types.
- **ProcessModifier (Enum)**:
  - **Purpose**: Enumerates possible process modifiers, used to customise the scheduling of the tasks that will perform the reconstruction (modify pipelines, algorithms, parameters).
  - **Attributes**:
    - `ticl_v3`: Enable version 3 of the TICL Framework
    - `ticl_v5`: Enable version 5 of the TICL Framework
    - `ticl_fast_jet`: Enable FastJet as Pattern Recognition in the TICL Framework
- **WorkflowSpecs**:
  - **Purpose**: Defines specifications for a workflow, including metadata and optional process modifiers.
  - **Attributes**:
    - `release` (`str`): CMSSW release used to process the samples.
    - `type` (`str`): Type of the workflow, used to dynamically load a generator and the resto of the configuration files (steps).
    - `era` (`int`): Era number associated with the workflow. To define which Era to consider (e.g. Run-3, Phase-2)
    - `pileup` (`bool`): Indicates whether to simulate the events with pileup.
    - `geometry` (`Geometry`): Specifies the CMS Geometry version using the `Geometry` enum.
    - `process_modifier` (`Optional[ProcessModifier]`): Optional modifier for the process, using the `ProcessModifier` enum.
- **WorkflowParameters**:

  - **Purpose**: Specifies parameters for the workflow execution.
  - **Attributes**:
    * `max_events` (`PositiveInt`): Maximum number of events, must be a positive integer (default: 10).
    * `n_jobs` (`PositiveInt`): Number of jobs to run in parallel, must be a positive integer (default: 1).
    * `customize_step3` (`bool`): Flag to indicate customization of step 3 (default: False).

- **Workflow**:
  - **Purpose**: Represents the overall workflow configuration, including specifications, parameters, and a generator.
  - **Attributes**:
    * `specs` (`WorkflowSpecs`): Specifications for the workflow, using the `WorkflowSpecs` schema.
    * `parameters` (`Optional[WorkflowParameters]`): Optional parameters for the workflow, using the `WorkflowParameters` schema.
    * `generator` (`Any`): Field for a generator that is further validated based on the workflow type.

- **ParticleGunParameters**:
  - **Purpose**: Defines parameters for configuring a particle gun. This particular workflow called Close by Particle Gun, allow to produce samples by shooting particle in fron of a specific detector and it is extremely useful to study reconstruction algorithms.
  - **Attributes**:
    * `controlled_by_eta` (`SingleOrMultiple[bool]`): Indicates whether the position of the particle can be chosen by pseudorapidity η instead of the Radius (R), can be a single value (default: True).
    * `max_var_spread` (`SingleOrMultiple[bool]`): Maximum variation spread flag. Allow to define an energy or transverse momentum range in which multiple particle can be produced. (default: False).
    * `delta` (`SingleOrMultiple[float]`): Delta value for particle generation. It is used to define the separation at which particles will be produced. (default: 10.0).
    * `flat_pt_generation` (`SingleOrMultiple[bool]`): Flag for flat pT generation. Used to generate the process modifying the transverse

momentum (pT) of the particle instead of the energy. (default: False).

  * `pointing` (`SingleOrMultiple[bool]`): Pointing flag. Used to produce particles that are pointing to the axis origin (0,0,0) instead of being parallel to the beamline (default: True).
  * `overlapping` (`SingleOrMultiple[bool]`): Overlapping flag. Used to allow overlap between multiple particle. (default: False).
  * `random_shoot` (`SingleOrMultiple[bool]`): Random shooting flag. (default: False).
  * `use_delta_t` (`SingleOrMultiple[bool]`): Use delta T flag. Used to set a difference in time between multiple particles. (default: False).
  * `eta` (`SingleOrMultiple[Range]`): Range for eta values (default: (1.7, 2.7)).
  * `phi` (`SingleOrMultiple[Range]`): Range for phi values (default: (-3.14159265359, 3.14159265359)).
  * `r` (`SingleOrMultiple[Range]`): Range for r values (default: (54.99, 55.01)).
  * `t` (`SingleOrMultiple[Range]`): Range for t values (default: (0.0, 0.05)).
  * `var` (`SingleOrMultiple[Range]`): Range for energy or transverse momentum (depends on texttttflat_pt_generation) values (default: (25.0, 200.0)).
  * `z` (`SingleOrMultiple[Range]`): Range for Z values (default: (320.99, 321.01)).
  * `n_particles` (`SingleOrMultiple[int]`): Number of particles (default: 1).
  * `offset_first` (`SingleOrMultiple[float]`): Offset for the first particle (default: 0.0).
  * `particle_ids` (`Annotated[SingleOrMultiple[UniqueTuple[Particle]], AfterValidator(get_particle_ids)]`): Unique tuple of particle IDs, validated by `get_particle_ids`.

- **CloseByParticleGun**:
  - **Purpose**: Defines configuration for a particle gun with additional parameters.
  - **Attributes**:

* `add_anti_particle` (`bool`): Indicates whether to add anti-particles (default: False).
* `verbosity` (`int`): Level of verbosity (default: 0).
* `first_run` (`int`): Specifies the first run number (default: 1).
* `psethack` (`str`): Custom string for particle configuration (default: 'random particles in phi and r windows').
* `parameters` (`ParticleGunParameters`): Parameters for the particle gun, using the `ParticleGunParameters` schema.

Chapter 5

Workflow Management with LAW

5.1 Overview of Luigi Analysis Workflows (LAW)

LAW is a framework built on top of Luigi that enables complex workflows to be easily defined and managed. It supports remote execution of tasks, integration with job schedulers like HTCondor, and provides mechanisms for handling large data storage solutions like EOS.

In this project, LAW was configured to manage CMSSW workflows. Tasks were defined to automate various steps, such as data generation, processing, and analysis, which traditionally would require manual configuration and execution.

One of LAW's significant features is its ability to manage tasks across remote systems. In this project, tasks were executed on the CERN grid using HTCondor and CRAB, which allowed for efficient processing of large datasets. However, as the law repo did not provide documentation for configuring it, implementing this feature was not possible even after multiple talks with the contributor of that package. This feature remains for future work.

5.2 Law Tasks

This section outlines the set of LAW (Luigi Analysis Workflow) tasks (which are python classes) designed to manage and automate the setup and execution of workflows within the CMS Software (CMSSW) environment. As each task is just a class adding new requirements to them is very easy, just like extending the outputs which is very crucial if multiple outputs on different locations are expected (local and remote for instance)

Tasks are organized to handle:

- The creation and management of directories for configuration and cache purposes.

- The setup and extraction of CMSSW releases from scratch or tarball files.
- The generation, filtering, and execution of CMSSW workflows, including the running of CMS jobs using configuration from TOML file.

5.3 Task Descriptions

5.3.1 CreateConfiguratorCachDirTask

This task creates a directory for caching configuration files. It ensures that the environment variable `CONFIGURATOR_CACHE_DIR` points to an existing directory.

CashReleaseDirTask

This task manages the directory for caching specific CMSSW releases. It relies on a specified release name and ensures that this release is available in the local cache.

GetCmsReleaseTask

This task sets up a specified CMSSW release environment within the directory defined by the `WORKSPACE_DIR` environment variable. It runs the command `cmsrel {release}` to create the necessary environment for the CMSSW release.

5.3.2 GetCmsReleaseDirFromTarTask

This task extracts a CMSSW release from a provided tarball into the workspace and runs `scram b ProjectRename` to rename the project appropriately.

5.3.3 SrcTask

`SrcTask` initializes the `src` directory, which serves as an output directory for other tasks. It sets up the necessary environment (`cmsenv`) for the CMSSW release.

5.3.4 CashCMSReleaseWorkflowsTask

This task caches the available workflows for a specified CMSSW release by running `runTheMatrix.py -w upgrade -n` and saves the output to `workflows.txt`.

5.3.5 CashSpecifiedWorkflowsTask

This task filters and caches specific workflows based on type, era, pileup, and geometry by running `grep` and saving the result to a text file.

5.3.6 GetCMSWorkflowTask

This task identifies and retrieves a specific CMS workflow based on type, era, pileup, and geometry by extracting the workflow ID and retrieving the corresponding workflow directory.

5.3.7 CreateCombinationsDirTask

This task creates a directory for storing combinations of workflow parameters within the source directory.

5.3.8 CreateCombinationTask

This task generates a specific workflow based on given input parameters, copying workflow files into a new directory and modifying them accordingly.

5.3.9 CMSRunTask

This task runs a CMS workflow using `cmsRun` based on a configuration file and a specific workflow step, saving the output to a ROOT file.

For more details on input, output and requirements of the task see the Appendix 7.

5.4 Working Example

I will display an example of a workflow.toml file and how it will appear in the Luigi interface.

The TOML file `workflows.toml` contains:

```
[specs]
release = "CMSSW_14_1_0_pre4"
type = "close_by_particle_gun"
era = 2026
pileup = false
geometry = "D110"

[parameters]
```

```
max_events = 50
n_jobs = 4
customize_step3 = false

[generator]
verbosity = 0
first_run = 1
psethack = "random particles in phi and r windows"

[generator.parameters]
particle_ids = [["PROTON", "GAMMA"], ["ELECTRON", "PROTON"]]
delta = [4, 5]
flat_pt_generation = false
eta = [[1.0, 2.0], [2 , 4] , [4, 6]]
```

Listing 5.1: workflow.toml

and running the following command and `luigid` on the other terminal:

```
law run CMSRunTask --step 3 --workers 12 \
--tar-file <tar_file>  --local-scheduler False
```

By going to `localhost:8082`, the user can see the workflow, which will look like this:

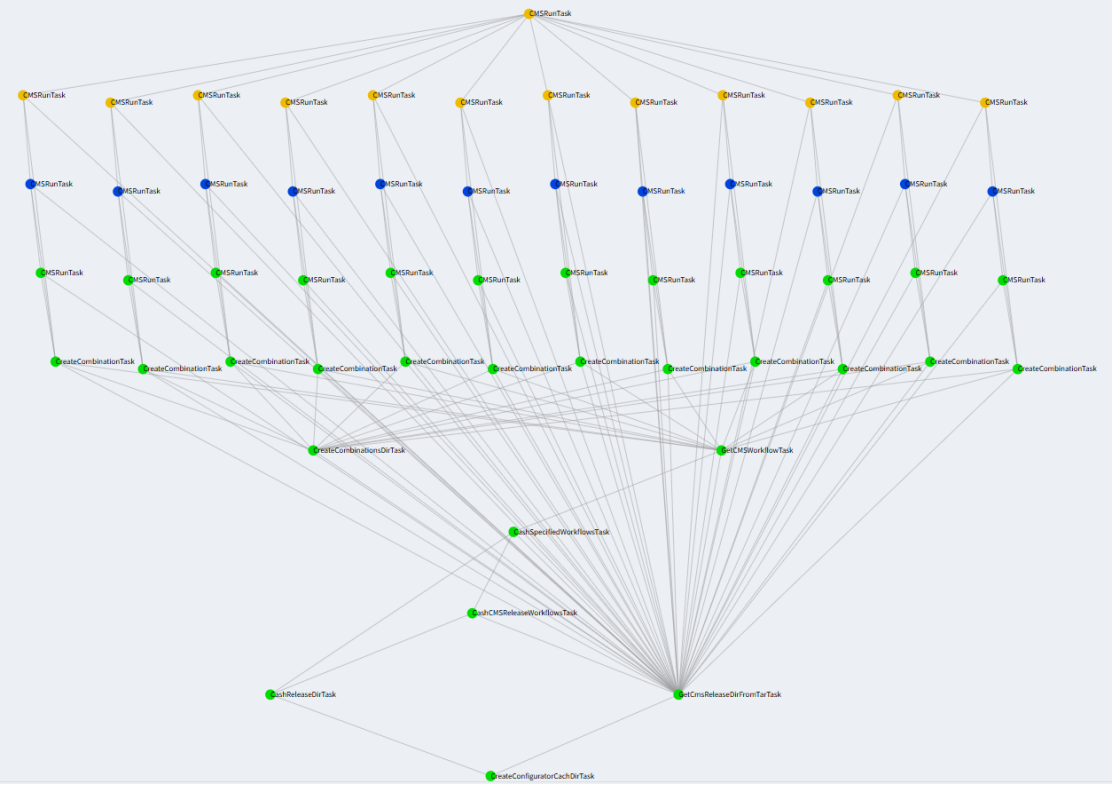

Figure 5.4: Luigi workflow visualization.

And as described in details in Tasks section. It is possible to clearly see all dependencies and how they influence each other

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this project I have have successfully created an automated solution for generating and validating CMSSW configuration files using Python, LAW, and Pydantic. The system can take a list of parameters and create a separate config file for each combination (permutation) of them a separate configuration file and run `cmsRun` on flawlessly. By using law, ease of visualization is also achieved. Users can easily see what stage the workflow is in, and if there are any failures, the error will also be displayed.

6.2 Future Work

Future improvements could involve integrating the system with HTCondor to facilitate automated job submission. Expanding the system's capabilities to handle a wider variety of CMSSW workflows is also important (Currently, only CloseByParticleGun is supported). Additionally, developing tests and incorporating users' feedback will help improve the system, which is also crucial.

Chapter 7

Appendix

Tasks Inputs, Outputs, and Requirements

CreateConfiguratorCachDirTask

Inputs: None
Outputs: {`CONFIGURATOR_CACHE_DIR`}
Requirements: None

CashReleaseDirTask

Inputs: `release`
Outputs: {`configurator_cache_dir`}/{`release`}
Requirements: `CreateConfiguratorCachDirTask`

GetCmsReleaseTask

Inputs: `release`
Outputs: {`workspace_dir`}/{`release`}
Requirements: `CreateConfiguratorCachDirTask`

GetCmsReleaseDirFromTarTask

Inputs: `tar_file`
Outputs: {`workspace_dir`}/{`release`}
Requirements: `CreateConfiguratorCachDirTask`

SrcTask

Inputs: `tar_file` (optional), `release`
Outputs: None
Requirements: `GetCmsReleaseTask` or `GetCmsReleaseDirFromTarTask`

CashCMSReleaseWorkflowsTask

Inputs: `SrcTask inputs`
Outputs: `workflows.txt`
Requirements: `SrcTask`, `CashReleaseDirTask`

CashSpecifiedWorkflowsTask

Inputs: `SrcTask inputs`, `type`, `era`, `pileup`, `geometry`
Outputs: Filtered workflows text file
Requirements: `SrcTask`, `CashCMSReleaseWorkflowsTask`, `CashReleaseDirTask`

GetCMSWorkflowTask

Inputs: `SrcTask inputs`, `type`, `era`, `pileup`, `geometry`
Outputs: Workflow directory
Requirements: `SrcTask`, `CashSpecifiedWorkflowsTask`

CreateCombinationsDirTask

Inputs: `SrcTask inputs`
Outputs: `combinations` directory
Requirements: `SrcTask`

CreateCombinationTask

Inputs: `SrcTask inputs`, `workflow_params`, `workflow_specs`, `generator_params`, `out_dir_name`
Outputs: Combination directory
Requirements: `SrcTask`, `CreateCombinationsDirTask`, `GetCMSWorkflowTask`

CMSRunTask

Inputs: `SrcTask inputs`, `workflow_file`, `step`
Outputs: `step{step}.root`

Requirements: `SrcTask`, `CreateCombinationTask`, `CMSRunTask` (for previous step)

References

[1] CMSSW github https://github.com/cms-sw/cmssw.

[2] Pydantic Documentation, version 2.7, https://docs.pydantic.dev/2.7/.

[3] law documentation, "Configuration", https://law.readthedocs.io/en/latest/config.html.