CROW: Python-based Configuration Toolbox for Operational and Development Workflows

Authors: Jian Kuang; Kate Friedman; Sam Trahan; Terry McGuinness; Kenneth Hammett; Mark Iredell; Arun Chawla

Abstract: The increasing level of complexity of EMC workflows lead to several challenges for researchers and users. One of the major issues is the absence of a modernized and generalized front-end. The Configurator of Research and Operational Workflows (CROW) is developed to fill the gap between developers and users, through an object-oriented programming approach using python. The goal of CROW is to drastically simplify the most time-consuming and error-prone stages of executing a workflow, such as platform adaptation, resource allocation and model configuration. This means more creative work could be done with given resources, in terms of both user hours and computer capacity. Highly human-readable YAML definition files are taken as input of CROW, and Rocoto or ecFlow definition file are generated automatically at the end. The introduction of CROW will greatly increase the efficiency of collaboration, documentation and R2O transition, and benefit users and developers from both EMC and the community.

1. Introductions

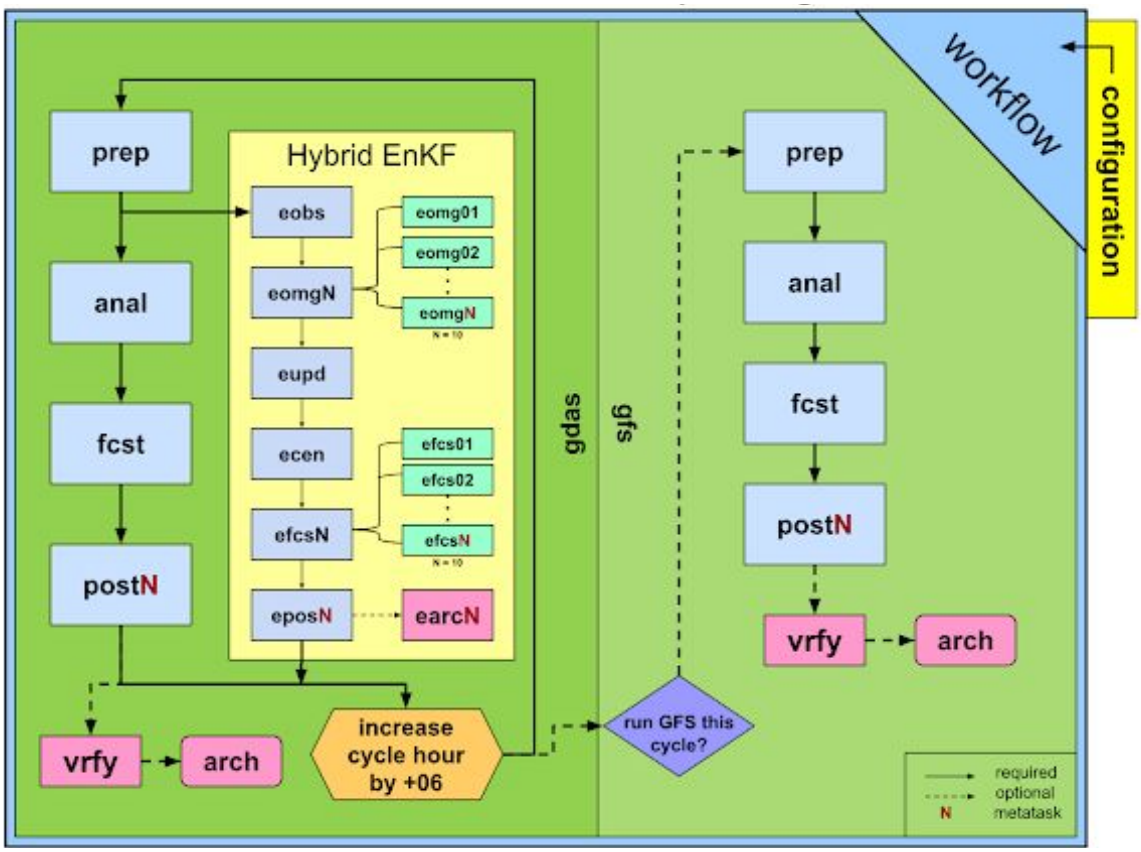
1.1 Workflows in weather forecasting practice

Modern weather forecasting practice across the world are built around evolving numerical weather models, or dynamic cores(“dycore”), such as FV3 and NMM. The jobs of these models are to calculate the weather state for a certain amount of time into the future, based on the current condition. As the core components of all weather forecasting systems, these models are essentially numerical applications(software) solving the governing equations of the environment on a spatial and temporal discrete domain. In most cases, they are running on supercomputers by parallel computing utilities.

However, a reliable dynamic core is only part of the story of building a weather forecast system serving the society. In order to have a complete operational forecasting system, a data assimilation system is often needed in order to intake environmental measurements across the domain, and make them into the initial condition state of the dynamic core, where the forecast starts from for that particular forecasting cycle. Furthermore, a full set of auxiliary programs including observation preparation, post processing and archiving are necessary when running in operational, since product generation needs to be standardized and automated.

As a result, the concept of “workflow” is introduced to describe the inter-application software connection between programs and applications that are included in a weather forecast system. Structurally, a workflow is usually a series of scripts taking care of initialization, resource

allocation, application configuration, dataflow, job card submission and status check/feedback. In a lot of cases, multiple workflow layout are provided in a given system for different scenarios, based on what the users need.



In practice, workflows are often embodied as a “suite” consisted of one or multiple files being read by a workflow manager, representing the workflow structures, dependencies, interfaces to individual jobs, and resource allocations. Workflow managers are software or toolbox calling workflow jobs and generate supercomputer job cards when conditions for a given job is met. For example, EcFlow developed by ECMWF which reads a series of ecflow definition files in order to run the workflow, and generates job cards based on detections of which job has all conditions met.

Workflow instantiation -> workflow manager -> scheduler -> launcher

1.2 Configuration system

The configuration manager, or configurator, is an integral component of all applications, They are usually formatted namelists, enabling users to **select** values for a series of required or optional fields that define the application that will be executed later. Practically, configuration

manager usually serves as the “user interface” of a given program so that users are able to start the application in the desired way. Configuration system are usually implemented as one or multiple formatted name-lists with ordered or un-ordered key-value pairs. The configuration system may be named “control file”, “config”, “name list”, “model definition” and so on. But their functionalities are similar.
It worths to mention that configuration system does not necessary to represent ALL functionalities that the application has. In other words, an application could, and often, has some functionalities that are “hidden” from the configuration front-end. Such functions are usually on their beta stage so that developers are still busy working on. And for that reason not open them to general users, which the configuration system serves.

All variables that are made available to the configuration system, or the front end, are called configurable variables. The combined set of all configurable variables are called configurable domain. The reason that calling it “configurable domain” instead of “configurable space”, is largely because configurable variables are not necessarily independent with each other thus in conflict with the algebraic definition of “space”. For example there could be, and often are, relations like this:

```
If varA == condition1
        varB = X1
        varC = Y1
elif varA == condition2
        varB = X2
        varC = arbitrary
else:
        varB = arbitrary
        varC = arbitrary
```

However, making the configurable domain a N-Dimensional space - like structure, with all variables independent with each other, is highly desirable since it will minimize script work in the workflow and thus make the workflow less prone to human errors. Will talk about it later.

1.3 Workflow configuration

Instead of focusing on one application, workflow configuration system serves as the front-end of a workflow consisting of multiple applications with dependencies in between. The purpose of workflow configuration, mainly consisted of two aspects: 1, Design and select the workflow structure; 2, Pass through certain configurable variables into applications that are included in the workflow.

As stated in section 1, A workflow could have multiple forms of structure. In other words, different combinations of jobs, and different dependencies are often available. For example,

whether the user wants to run the workflow with or without data assimilation suite; how many forecast hours should be included and so on.

On the other hand, all applications included in the workflow have their own configuration system that serves their own needs. For this reason, a successful workflow configuration system also needs to make connection with the individual configuration system that the workflow has invoked during the jobs.

1.4 CROW-based workflow configuration

In CROW, we use YAML as the describing language about user inputs as well as entire workflow definition; and developed a python toolbox based on PyYAML 5.1 to read it and write out the necessary workflow documents. The reason choosing YAML is because 1, It is absolutely similar to natural English so that making interpreting effortless; 2, There are multiple choices of parsers across different computer languages like C++, JAVA, Ruby and so on. Though a CROW version that is not based on python does not exist yet, the choices are open for the future when needed.

2. Structure of CROW-based workflow configuration

2.1 YAML and CROW-YAML

YAML (a recursive acronym for "YAML Ain't Markup Language") is a mainstream data serialization language. Though having a similar appearance and organization style with markup languages, YAML adopts the concept of “data serialization language, The concept of data serialization language differs from markup languages by focusing on delivering the data or essential information by simple and standardized format, while markup languages often focus on text document processing. YAML is extremely suitable for configuration purpose in a lot of ways, most notably it’s high-readability and free from built-in reserved words. In addition to that, YAML allows customized data type to be built which is also very important to workflow configuration, and the convenience of handling multiple types of value within a single object.

CROW YAML is developed on top of PyYAML with the addition of several customized data types. User-defined data type is named alpha-numerically with “!” at the beginning when calling. CROW defines several of these to handle calculations, conditionals, templates, times, dependencies, and some others.

Configuration information handling by CROW features:

1. **Only once**. Though not recommended, CROW enables users to specify a certain variable and override it at a later stage. But, only the final value of a certain variable could be passed into the targeted workflow that is generated.
2. **All or none**. CROW always reads all YAML files at one time. This design helps ensure consistency within the Configuration Suite and Workflow Suite.

3. **Diagnosable**. In CROW, the experiment directory has effectively become a “checkpoint” for the configuration. This helps minimize human error and increase accountability, comparing with manually editing throughout experiment directory.

This file will be read by crow.config and become a eval_dict object of python. There are two sections in the eval_dict created, original contents are stored in _child and calculated values are stored in _cache.

2.2 CROW toolbox

The core of CROW toolbox is a YAML parser, namely crow-yaml, developed based on PyYAML 5.1. A YAML parser, is a toolbox written in compiling languages (JAVA) or interpreting languages (Python), that could perform two-way conversion between YAML file(s) in ASCII format, and a data object, in most cases a dictionary, in memory. User-defined data type is named alpha-numerically with “!” at the beginning when calling. CROW defines several of these to handle calculations, conditionals, templates, times, dependencies, and some others.

Example YAML file read by CROW

```
my_crow_yaml:
  five: 5
  six: 6
  thirty: !calc five*six
  sixty: !calc thirty*2
  true: !calc thirty<sixty
  tomorrow: !timedelta +24:00:00
  htxt: hello
  wtxt: world
  hello_world: !expand “{htxt} {wtxt}”
```

Python Data structure generated by CROW:

```
my_crow_yaml=eval_dict({
  “five”: 5,
  “six”: 6,
  “thirty”: strcalc(“five*six”),
  “sixty”: strcalc(“thirty*2”),
  “true”: strcalc(“thirty<sixty”),
  “tomorrow”: TimedeltaMaker(“+24:00:00”),
  “htxt”: “hello”,
  “wtxt”: “world”,
  “hello_world”: strexpand(“{htxt} {wtxt}”)
```

```
})
```

In a CROW-configured workflow system, all configuration settings, defaults or customized, are written in a series of YAML files. These files will be read by crow-yaml which will generate a configuration suite, a specialized python dictionary object that includes all configuration settings for a particular experiment of the workflow. It will then write out instantiated workflow files (config files; Rocoto XML or EcFlow definition files) to the targeted location at the user's discretion.

3. Implementation

3.1 CROW based global workflow

When implemented, CROW is situated a submodule under the targeted workflow. However, CROW will not function without an interface layer between the targeted workflow and CROW, which needs to describe information about the workflow itself, for example, the viable possible layout(s) of the workflow; the default values of variables; the resource settings and so on. As discussed before, CROW reads a series of YAML files to decide what the user wants to run for this experiment. Consequently, a series of YAML files become the interface between the workflow itself and the CROW toolbox. These files are named "background files", for the reason that they stay at the workflow repository, and stay in the background which means normal users are not supposed to modify it in most cases.

(Example) Repository structure of global workflow:

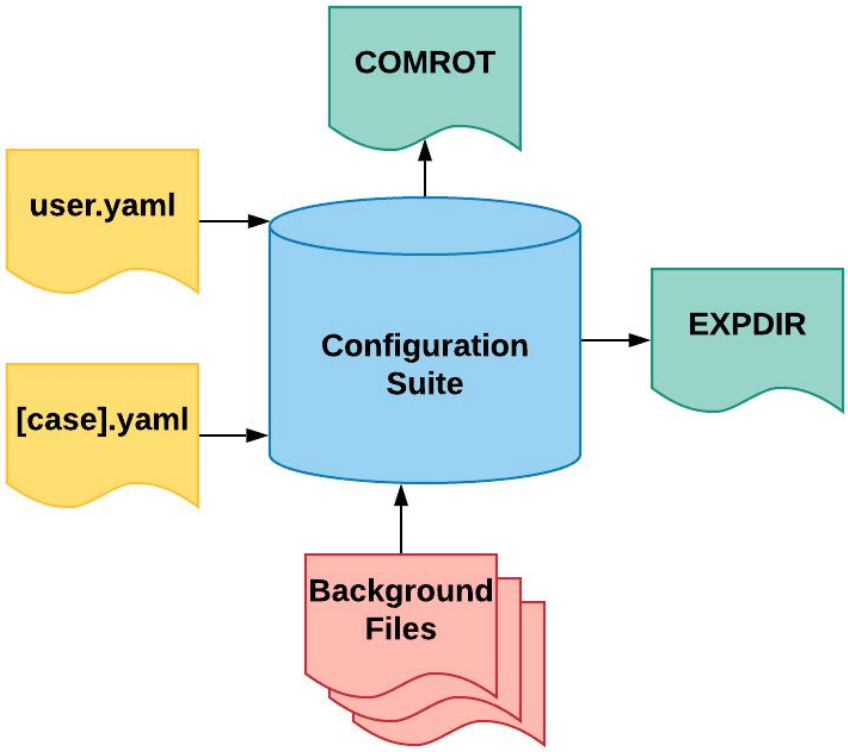
(top)global-workflow

- **docs** : related documentations
- **fix**: Directory for fix files
- **jobs**: Top level job scripts (J_* job)
- **modulefiles**: modules to be loaded
- **parm**: templates for config
- **scripts**: driving scripts
- **sorc**: application source code
- **ush**: lower level scripts
- **util**: various utilities
- **workflow: Workflow Configurator**
  - **CROW repository**
  - **background files**
  - **user file**
  - **case file**

CROW uses a two-step approach to generate a workflow. The first step is called **Configuration**. In this step, CROW will detect the running platform and read in all YAML files to make sure all required variables are properly set. If no problem occurs, a **Configuration Suite** will be generated. The Configuration Suite is a python eval_dict object in memory. In the end, CROW

will rewrite all input YAMLs into **experiment directory**, which is created in the beginning of this step, and parse all configuration variables to config files into the experiment directory.

Namely, a **configuration suite** contains all configuration information of a given experiment, including default settings (clock, alarm) and most importantly, a series of **tasks** with dependencies between each other. A suite can be defined over one or more cycles, while each task has a section to define which cycle it will run.



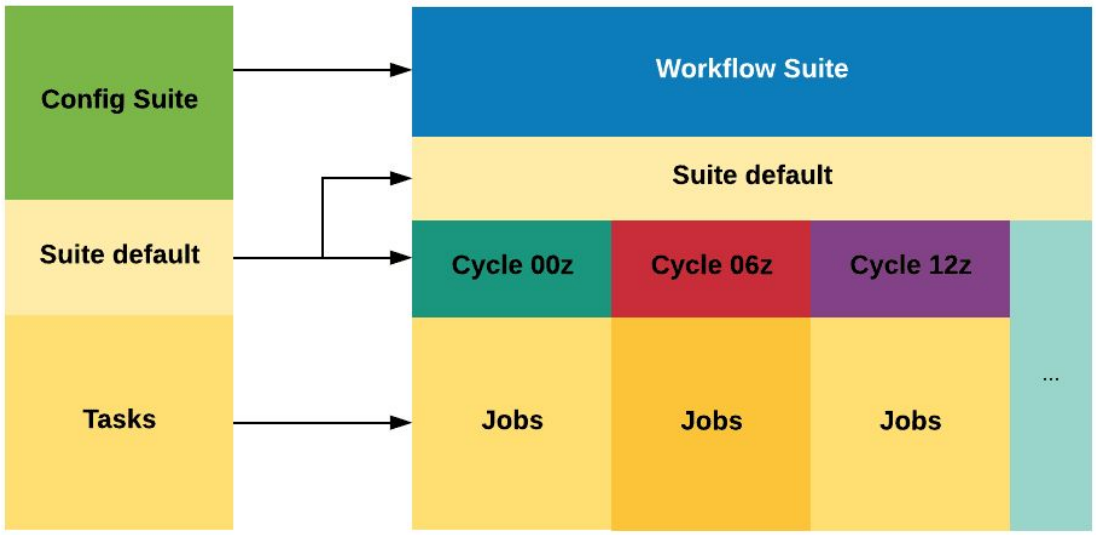
CROW Step 1, Configuration
Flag-shaped boxes are text files; Circular bins are python objects;
Yellow: User input files;
Red: Background configuration files
Green: Output files;

The forms and organizations of background files could be different across workflows, which normally have distinct structure, features, and restrictions. However, the following sections are commonly included:

1. runtime:

This is the section for a series of task templates, as together with a series of suite default sections of rocoto and ecFlow. For example, templates for the rocoto "header" section. An important section here is the "suite_default" which represents utilities and common sections which are shared through the entire workflow.

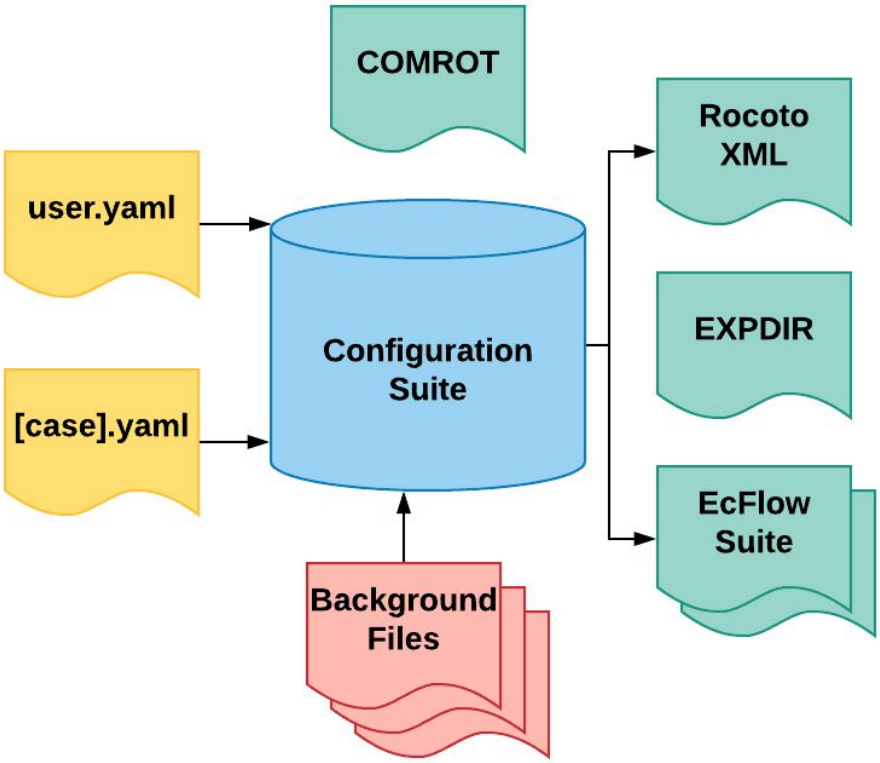
2. defaults
   This is the section for default values of configurables. Though not mandatory, we encouraged that all configurables have one and only one default value defined here. Because otherwise, that variable will have to be set by the user EVERY time starting a new configuration. In addition to these, default resource allocation are usually in this section.
3. layout
   This section describes the overall structure of the targeted workflow. Most notably the tasks that are available and the relationship between them. The relationship between "task" and a "job" is somewhat the class-object relationship in object-oriented programming paradigm. In other words, a "job" is an instantiation of a "task" at a particular temporal cycle (00z, 06z).



4. platform
   In this section, a series of supercomputer description files are included. The purpose of this section is to enable automatic platform detection and adaptation. Each yaml file represents a supported platform of that workflow, and a "detect" line is mandatory which set the criteria for if it is the platform that is currently active.

In the second step, named Generation, CROW will read in all YAML files and generate the Configuration Suite again, while also populate all tasks, along with suite default settings, onto

designated cycles to generate a Workflow Suite, with choice of Rocoto or ecFlow as workflow manager.



CROW Step 2, Generation
Flag-shaped boxes are text files; Circular bins are python objects;
Yellow: User input files;
Red: Background configuration files
Green: Output files;

The basic component of a CROW **configuration suite** is called a **task**. In a workflow, a task is a single element or step of the modeling system providing a certain functionality by a set of scripts (For example: gdasefcs01). Inside CROW toolbox, a task is represented by python objects. Most tasks are associated with a J-Job script which handles submission of a supercomputing job card, with specific resource requests (time, cpu, memory ...). Some tasks come as an array called a “taskarray”, where multiple tasks performing the same functionality are grouped together (ie, gfs_post_XX, enkf_forecast_XX …..). Tasks will be instantiated as “jobs” when the **workflow suite** is generated and submitted to the workflow manager (For example: gdasefcs01 at 00 cycle and 06 cycle are two distinct jobs, but are derivatives of the same task gdasefcs01).

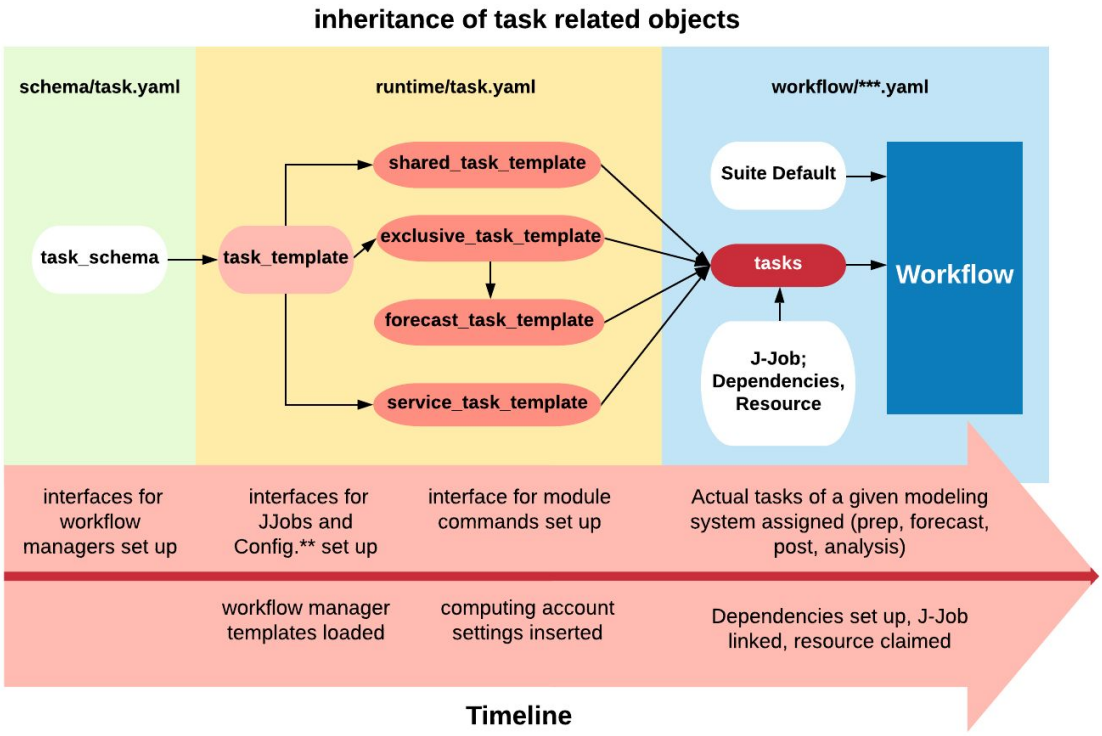
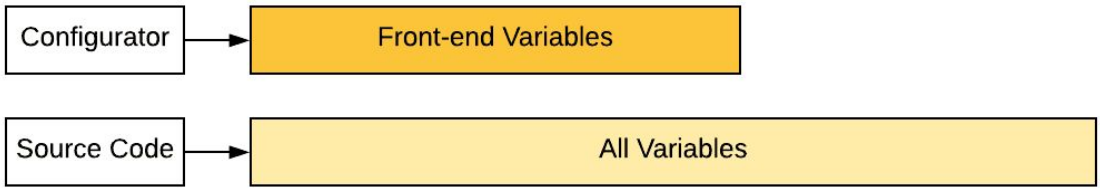


Fig 3. Detailed explanation of how tasks are defined will be given later

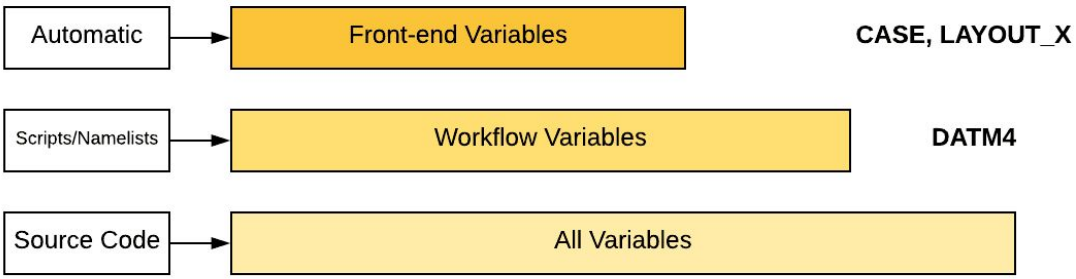
3.2 Configurable domain and variables

We use “configurable domain” to describe the set of variables that could be configured within the configuration system. However, it is often impossible and also unnecessary for the configuration domain to be “complete”, in which case it includes ALL required parameters of the application it serves.



Same principles apply to workflow configuration systems. The difference is that, there’s one more layer of variables named “workflow variables”. These variables, while being part of the configurable variables of the underlying applications (FV3, post...), are sometimes being excluded from the workflow configuration system higher above. The reason for this is mostly 1. All of the three sets are evolving simultaneously so keeping up the underlying layer(workflow

variables) would be too repetitive and time consuming to be done in real time. 2. The underlying applications may have additional parameters needed when running standalone, thus being inactive when running a workflow.



variable structure of a workflow and configuration system, automatic

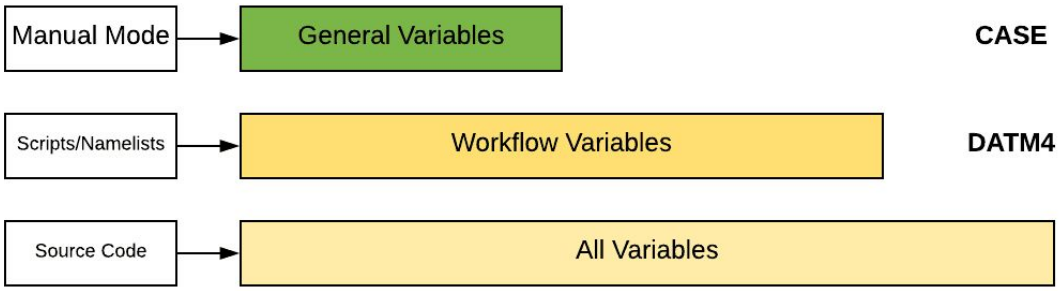
3.3 Automatic vs Manual

For developers that are making changes of the underlying system, keeping update the workflow configuration system together with their work would become an extra burden. For this reason, a manual mode has been developed.

In the manual mode, the configurable domain would be much smaller than the manual (default) mode. Because only variables that are “general” will be included.

A variable is considered “general” if one of the following is true:

1. It appears in more than one job. In other words, it is included in the configurator of more than one underlying application. Common example: start/end date.
2. It will affect the workflow structure overall. Common example: do_postprocessing, decide that if post_processing jobs will be included.



Summary
CROW (Configurator of Research and Operational Workflows) is a python-based developed to serve the need to configure the workflow with complex structures, dependencies, using the concept of object-oriented programming paradigm. The potential of CROW is to enable a common configurator through all weather workflow configurations of NOAA, and provide a user-friendly front-end for everyone in the community that would like to attempt running the workflows.

Reference
Github repository - CROW: https://github.com/NOAA-EMC/CROW
Github repository - global-workflow: https://github.com/NOAA-EMC/global-workflow
CROW documentation: https://noaa-emc.github.io/CROW/docs/html/index.html