SMARTS - Simple MPAS Atmosphere Regression Testing System - User Guide

v.000, 21 January 2020

Contents

Chapter 1

Introduction

SMARTS is a portable, efficient and flexible regression testing system originally built to preform regression tests on the Atmospheric component of the https://github.com/MPAS-Dev/MPAS-Model and its auxiliary tool set. However, SMARTS was designed to test a wide variety of programs written in a wide range of programming languages on a number of different machines, so creating tests for other projects should be easy.

Since some regression testing for MPAS is best done on an HPC (like UCAR's Cheyenne) SMARTS was designed to submit jobs on compatible HPC workload managers (Current, only PBS and soon to be SLURM).

1.1 Background

There are three challenges that a regression testing system for a community, numerical weather prediction (NWP) model is tasked with solving:

- Portability - Community NWP models need to be portable to enable community use. With differences between machines, compilers and libraries (and versions of these compilers and libraries) changes to a NWP model needs to be tested and ran upon these different platforms, and the different combinations of these platforms. Thus a regression testing system needs to be able to run the same tests upon different machines, compilers and libraries.
- Efficiently and Resource Management - Regression tests need to run as quickly as possible. Running independent tests concurrently is one way to increase testing runtime. Regression tests for a NWP model may themselves be multiprocess program themselves, thus a regression testing system for a NWP needs to run concurrent tests that are multiprocess themselves while respecting shared resources
- Test flexibility - Regression tests must have no limit to what they can test, as any limit would be a reduction in test coverage of an application. So a regression testing system should be able to test not only the NWP model, but any related or unrelated application.

1.2 Portability - The Environment File and Environment Class

SMARTS provides portability by having a file that describes different machines, the Environment.yaml file. This file describes the resources for a specific machine, which includes the number of cpus for this machine, and the available compilers, libraries and environment variables.

The environment class reads the Environment.yaml file and allows SMARTS (and SMARTS' tests) to load compilers and libraries across different machines via the same command. This allows access to resources that might be specified in different manners. For instance, one machine might might use https://github.com/TACC/lmod to load compilers and libraries, while another uses environment variable (i.e. `module load pnetcdf` vs `export PNETCDF=/path/to/pnetcdf/`.

The environment class (along with the Environment.yaml file) allows a single test to load the same library via the same command, regardless of how its loaded on the machine.

Currently, SMARTS only has the ability to load compilers and libraries via Lmod or by environment variables.

For more information on the environment file and loading and unloading compilers see Chapters 4 and 5.

1.3 Efficiency

SMARTS has a number of different strategies to insure that tests are ran efficiently.

The first includes running independent tests concurrently and maximizing specified system resources. Based on the number or CPUs specified in the Environment.yaml file, SMARTS will launch independent tests concurrently if doing so does not exceed the number of specified CPUs.

The second is that SMARTS has the ability to stop the execution of a chain of dependent tests if single dependent test fails. For instance, the chain of tests for a restart test for an atmospheric model might be: compile with gnu-¿ smoke test (run for a single time step and output files)-¿restart test. Obviously, trying to run the smoke test and the restart test if the model fails to compile does not make much sense. Thus, SMARTS will not run tests if one of their dependencies fail.

Lastly, SMARTS has the ability to run jobs on HPC compute cluster. Running model simulations on local workstations or login nodes is inefficient and an unethical use of shared resources. Thus, SMARTS has the ability to launch jobs on HPC compute nodes. Currently, this is only on HPC's that use the PBS workload manager.

Note: Launching jobs on an HPC via PBS has only been tested on the Cheyenne super computer. If you use another PBS system and have difficulty launching HPC jobs please get in touch.

1.4 Test Flexibility

SMARTS and SMARTS' tests are both written in Python. However, this is not to say that SMARTS *can not* test applications that are not written in Python. Since Python has a number of modules that can launch Python and non-python application as subprocesses SMARTS test can be used to test any number of applications.

SMARTS only requires that tests have a small Python interface. This interface will allow SMARTS to run tests as a multiprocess and will allow tests to communicate PASS or FAIL results to SMARTS.

For more information on tests, see Chapter 3.

Chapter 2

Command Line Interface - smarts.py

SMARTS can currently only be launched from the command line using Python via `smarts.py`. In the future there may be other options for launching SMARTS.

The `smarts.py` command line program is a Python3 program and should be launched by using Python3. The `smarts.py` interface is built in the similar manner to the Git commandline tool in that there are commands and sub-commands. Different to the Git commandline tool, `smarts.py` has a number of options that are required to run.

Listing 2.1 shows the usage and help message for `smarts.py`. There are three required arguments, which are listed in the 'Required Arguments' section. These three requirements are:

- `-e/--env-file env.yaml` - The Envrionment.yaml file that describes the current machine. This file describes the machine and any specified compilers or libraries to SMARTS so that tests are able to load and unload tests across different machines. See Chapters 4 and 5 for more information on specifying and using the environment.yaml file.
- `-s/--src-dir dir` - The directory of changes to test. This directory path will be passed to each test (as the `src_dir` argument) and will allow tests to copy soruce code or executables for regression testing. This directory can be located anywhere on the filesystem. See Chapter 3 for more infromation on using the directory from the `-s` argument in tests.
- `-t/--test-dir dir` - The directory that contains the desired tests to run. This argument does two things. One, it shows SMARTS where tests can be loaded. SMARTS will look in this directory for valid tests and will only load tests from this directory. Secondly, it passes this path to each test which allows tests to have access to any supplementary files that they may need. For instance this could be a namelist or streams file that is specific for a spcific test.

All three of these requirments arguments must be specified or `smarts.py` will return an error. After these required arguments commands and their subcommands may be specified. The two commands and their subcommands are:

- `list` - The `list` can be used to display infromation of the current SMARTS system. At present, the list command can only be used to list tests and test information, but in the future it may be expanded to print other infromation. The subcommands for `list` are:
  - `list tests` - List the valid and invalid tests that are found the the directory passed to the `-t/--test-dir`. See 2.1 for more information.

  - `list test test-name[s]` - Print out the additional infromation on a specific test or test(s). See 2.1 for additional information.

- `run test-name[s]` - Run the specified tests. See 2.2 for more information.

Listing 2.1: smarts.py Help Message

```
usage: smarts [-h] [-e env.yaml] [-s dir] [-t dir] [-v level] {list,run} ...

A regression testing system for MPAS

optional arguments:
  -h, --help            show this help message and exit

Required arguments:
  -e env.yaml, --env-file env.yaml
                        The location of the env.yaml file
  -s dir, --src-dir dir
                        The directory that holds the code to test changes
                        (MPAS-Model)
  -t dir, --test-dir dir
                        The location of the test directory

Optional arguments:
  -v level, --verbose level
                        Output debug level

subcommands:
  command description

  {list,run}            Sub-command help message
    list                List SMART's tests, test suites and compilers
    run                 Run a test or a test-suite by name
```

2.1 List

An example of the `list test test-name` and its subcommands can be seen in Listing 2.2. The `list test test-name` command will display two groups of tests. One for valid tests and another for invalid tests. Valid tests are tests that contain all the necessary parts to be successfully loaded and launched by SMARTS. Invalid tests, on the other hand, are tests that SMARTS could not load for one reason or another. Tests that contain syntax errors or are missing critical parts of a SMARTS test will be placed in invalid tests, along with the reason they were invalid.

Listing 2.2: smarts.py List Tests Example

```
>>> python3.py smarts.py -e ./envs/cheyenne.yaml -s ~/mpas_model_changes -t
./mpas_tests list tests
Tests found in: /users/mcurry/smarts/smarts_mpas_test:
Valid tests:
  - bit_for_bit_tests -- Bit for Bit Test
  - compile_test -- Test that MPAS compiles
  - mpas_reg_tests -- Complete MPAS Regression Test
  - smoke_test -- Smoke Test
  - sst_update_test -- SST Update Test

```

```
Invalid tests: (These tests were not able to be loaded)
  x restart_test.restart_test -- EOL while scanning string literal (restart_test.
      py, line 12)
```

To find out more information on a test, one can run the `list test [test-name]` command and smarts.py will print out additional information test information. This command can be used with any number of test names. Listing 2.3 provides an example for this command.

Listing 2.3: smarts.py List Test Info

```
>>> python3.py smarts.py -e ./envs/cheyenne.yaml -s ~/mpas_model_changes -t
./mpas_tests list test mpas_reg_tests
    Run name: mpas_reg_tests
   Long name: Complete MPAS Regression Test
 Description: Run all MPAS regression tests
       ncpus: 1
Dependencies: ['compile_test', 'smoke_test', 'bit_for_bit_tests',
               'sst_update_test']
```

2.2 Run

The `run` can be used with one or more test names. Doing so, will launch the specified tests through SMARTS, if those tests are valid. If they are not, an error will be reported and no tests will be ran.

If all tests are valid and able to be loaded correctly, then SMARTS will create a new run directory will be created. Within this directory tests will be given their own directory, which will become their current working directory. Grabbing the results of one test (say a compiled executable from a compile test) can be done by using the current working directory path. All main test run directories will named `run-smarts-YYYY-MM-DD-hh.mm.ss` with the date and time pieces being the date and time whne SMARTS was launched.

If a test has a test (or multiple tests) listed in its dependencies attribute, and that test is not specified in the run command, then it will automatcially be loaded and ran. Tests that are dependencies for other tests will be ran before their dependents and dependents will not run if any of their dependencies fail.

For instance, in 2.3 specifying mpas_reg_testing to run will load and run all of the dependencies listed in the Dependencie's list. If any of these tests fail, then the result of mpas_reg_testing will be marked as `INCOMPLETE`.

Tests will only be loaded once per `smarts.py` command. It is not possible for SMARTS to run two of the same tests twice. This includes if a test is specified as a dependency and is specified to run via `smarts.py`.

Running the same test twice should be done with seperate commands to `smarts.py`.

Listing 2.4: smarts.py run tests example

```
>>> python3.py smarts.py -e ./envs/cheyenne.yaml -s ~/mpas_model_changes -t
./mpas_tests run mpas_reg_tests
TEST RESULTS
================================================
 - mpas_reg_tests - PASSED - "All␣regression␣tests␣passed"
 - compile_test - PASSED - "MPAS␣compiled␣succesfully"
 - smoke_test - PASSED - "Smoke␣test␣succesfull"
 - bit_for_bit_tests - PASSED - "Bit-for-bit␣identical"
 - sst_update_test - PASSED - "SST␣update␣runs␣as␣expected"
```

Chapter 3

Tests

The unofficial motto of SMARTS is: "If Python can do it, SMARTS can do it!" SMARTS' tests have the ability to tests virtually any kind of program or operation system function, as long as the said program can be ran and checked for success or failure via Python commands, it can be a test.

SMARTS tests can contain any number of Python functions, classes, modules or libraries and run any number of external programs or executables.

Python contains a number of different methods and modules for launching and managing executables and subprocess so launching external executables, such as an atmospheric model, is possible.

All test will at least have a set structure so the SMARTS TestManager can load, launch and receive the results of tests. Each tests will therefore have an interface in the form of a single class and a single python function.

This chapter describes that interfaces.

3.1 Test Structure

All SMARTS complaint tests will consist of (at least) a directory, a file and a class all with the same name. Listing 3.1 shows an example test directory and two example tests.

Listing 3.1: Example Test Structure

```
\test_directory
    \test1
        test1.py
    \test2
        test2.py
```

The name used for the test directory, test file and the test class are used as the test launch name as this is the name that the user will use to run the test via the `smarts.py` command line interface.

Each class will need to have the following attributes:

- `ncpus` - `int` - Required - The number of CPUs that this test will used. When this test is ran, SMARTS will subtract this number from the number of available CPUS and will only launch other tests if enough CPUs are available.
- `test_name` - `str` - Optional - This is the 'long name' of the test and is currently only shown when the `list test test-name` command is run. However, providing it offers others who are reading the test (or who have run `list test test-name` more details on the test).

- `test_description` - `str` - Optional - Similar to the `test_name` the `test_description` provides information to other users (and perhaps your future self) on the details of specific tests. While it is optional it is recommend that a good description of the test be provided.
- `dependencies` - `list of str` - Optional - If a test is dependent upon the result of another test, then that tests launch name should be added into the `dependencies` list. If a test list another test in its dependencies (or multiple tests) it will only be ran if all of its dependents complete successfully. Listing 3.3 shows an example test with a single dependency.

Each test file will need to define a class of the same name as the test test directory name and the test file name. The class will need to define a single function, `run`. The `run` function will be starting location of execution when SMARTS runs a test.

The `run` function will need to take the following arguments.

1. `self` - Reference to this test instance.
2. `env` - The environment class that contains information on the current environment. See chapters 4 and 5 for more information on the environment class.
3. `result` - The result object which is used to communicated results from the test to the `TestManager`.
4. `src_dir` - The directory that contains the code to be tested. If SMARTS was started from the `smarts.py` command line interface this is the directory that was passed via the `-s` command line option. Use this directory path to copy or link files into the current working directory.
5. `test_dir` - The path to the test direction. Similar to the `src_dir` argument, if SMARTS was started from the `smarts.py` command line interface the `test_dir` is the directory that was passed via the `-t` command line option. Use this directory path and the name of each test to retrieve supplementary test files. For instance, a namelist.atmosphere file used for a idealized simulation.
6. `hpc` - An instance of the HPC class, instantiated with the HPC interface queuing system that was specified in the Description section of the env.yaml file. This object can be used to schedule jobs upon the current HPC (if one is present). See Section‘ 3.3 on starting jobs via the HPC class.

Listing 3.2: Example test1.py

```
1  import os
2
3  class test1:
4      ncpus = 1
5      test_name = 'Test 1'
6      test_description = 'SMARTS example test'
7      dependencies = None
8
9      def run(self, env, result, src_dir, test_dir, hpc):
10         if True:
11             result.result = "PASSED"
12             result.message = "True is true!"
13         else:
14             result.result = "FAILED"
15             result.message = "It appears True is no longer fact"
```

Listing 3.3: Example Test With Dependencies

```
import os

class test2:
    ncpus = 2
    test_name = 'Test 2'
    test_description = 'SMARTS example test dependent on Test1'
    dependencies = ['test1']

    def run(self, env, result, src_dir, test_dir, hpc):
        ...
```

3.2 Test Results

Tests are useless if they cannot communicate their results to tester. Each test is passed an instance of the `Result` class. The `Result` class is defined in Listing 3.4. The `ResultClass` contains attributes `result` and `msg`. The `ResultClass` can be see in Listing 3.4.

Listing 3.4: Result class definition

```
class ResultClass:
    result = None
    msg = None
    directory = None
```

Each test is passed the `ResultClass` object which can be used to communicate the tests result with SMARTS. Setting `ResultClass.result` to either `"PASSED` or `"FAILED"` results in a pass or fail from that specific test respectively. The `ResultClass.msg` attribute is an optional attribute that can describe the reason for failure (or success) of a test.

These results will be communicated to the user in the tests results report when all tests finish completion, along with the error message.

After a test completes, SMARTS will unscheduled a test if one of their dependencies returns a `"FAILED"` result. Unscheduling a test results in its `ResultClass.result` being set to `"INCOMPLETE"`. If SMARTS updates a test (based on the `"FAILED"` result from another test it will recursively unscheduled any test that is dependent upon it.

Setting `Result.result` to `None` has the same effect as setting the test to `"FAILED"`. `Result.result` is set to `None` by default.

In the future there are plans to have more extensive reporting tools. The main being a way to place the results of tests and plots into a single LaTeX file and compiled into a PDF which can enable testers to view tests and plots created by tests in a single place.

3.3 Starting HPC Jobs

Tests in SMARTS have the ability to launch HPC jobs from within tests. Currently, only PBS HPC's are supported, but in the future SLURM will also be supported.

Jobs can be launched via the `HPC` instanced passed into the run function of every test. Upon reading the specified environment.yaml file on started, SMARTS will initialize the corresponding HPC class (at this point either PBS or None), which will be passed to each test.

To have a test determine if its on an HPC machine or not, it can compare the `HPC` instance to the currently available types (currently only `PBS`, but in the future `SLURM`). HPC can currently have the possible values:

Listing 3.5: HPC.launch_job

```
def launch_job(self,
               executables,  # List of executables to run
               name,         # Name of the HPC job
               wallTime,     # Walltime in HH:MM:SS
               queue,        # Desired queue
               nNodes,       # Number of nodes
               ncpus,        # Number of CPUS per node
               nMPI,         # Number of MPI tasks per node
               **kwargs):
```

- `None` - If the HPC is `None`, then the current machine is not an HPC.
- `"PBS"` - If the HPC is `PBS`, then the machine is a PBS machine.

Tests can then use the two methods provided to launch a PBS batch job. These methods are: `HPC.launch_script` and `HPC.launch_job`. `launch_script` provides a way to launch already created batch scripts while `launch_job` provides a method for directly specifying a job.

Both of the HPC job commands above are blocking. So, if a test calls either `launch_job` or `launch_script`, that test will block until the batch job is completed. Upon successfully completion, `HPC.launch_script` and `HPC.launch_job` will return True; if the job is not able to be successfully submitted to the queuing system the two functions will return False.

HPC and its two functions, `HPC.launch_script` and `HPC.launch_job` provide no functionality for checking the result of an HPC job, they only provide whether jobs were successfully submitted to the queue and finished. Just because `HPC.launch_script` or `HPC.launch_job` returns True does not necessary mean the test completed successfully, only that it was accepted, ran and finished on the HPC. It is up to the test itself to check the result of the job (i.e. by reading log or netcdf files etc.).

In order to help facilitate potential problems and to ensure correct job submission, they HPC logs all commands sent to the HPC and all received messages from STDIN and STDOUT. If HPC queuing errors occurred, it is best to check this log file for more information.

The name of the log file will be: `smarts-hpc.SCRIPT-NAME.log`. Where SCRIPT-NAME is the name of the script used to submit the job (with any extension).

3.3.1 HPC Launch Job

`HPC.launch_job` has the following arguments:

1. `Executables` - A list of executables to be preformed by the job. For instance: source an environment file and launch the init_atmosphere core. These executables will be ran in the order that they appear. For example: `executables=['ulimit -s unlimited', 'source /setup_cheyenne', 'mpiexec_mpt ./init_atmosphere']`.
2. `name` - The name to give the HPC job (In PBS this is the '-N' option.)
3. `wallTime` - The wall time in hh:mm:ss
4. `queue` - The desired queue to use.
5. `nNodes` - The number of nodes to use

6. `ncpus` - The number of cpus to use per node
7. `nMPI` - The number of MPI tasks to use per node
8. `**kwargs` - Optional keyword arguments
    - `shell` - The desire shell to use for the script. This line will appear at the top of th shell script. The default is: `#!/usr/bin/env bash`.
    - `pbs_options` - *Soon to be just options* - A dictionary of additional options to use in the script. All key, value pairs of the dictionary will be added to the script. The key will be the option argument and its corresponding option will be the value of the option argument. For instance: `options = { '-M' : 'email_address' }` will be translated and inserted to `#PBS -M email_address` for a PBS job script.
    - `script_name` - The desired script name. Default is `script.pbs`.

Given the arguments provided and optional keyword arguments provided to `HPC.launch_job`, `HPC.launch_job` creates a job script (script.pbs) and calls the corresponding queue submission command on that script.

3.3.2 HPC.launch_script

`HPC.launch_script` allows a test to run a batch script that was created before by the user. The function definition of `HPC.launch_script` can be seen in Figure 3.6. `HPC.launch_script` takes a single argument, `script`, which should point to a valid batch script. This script will be submitted the corresponding HPC workload manager.

`HPC.launch_script` has a single optional keyword argument, `cl_options` which will be a list of options to pass the command to submit the script. Options and their arguments (if they have any) should each be separate elements of the list. For instance: `cl_options = ['-M', 'email_address']`. Options are added to the submission command in the order they appear.

Listing 3.6: HPC.launch_script

```
1 def launch_script(self, script, **kwargs):
```

Chapter 4

Environment.yaml Machine Specification

Preforming regression tests across multiple machines and with different compilers and libraries is a necessary part of maintaining quality software. Especially if that software is intended to be used on a variety of machines with a variety of different compilers and libraries.

All machines vary in the amount of resources they have present and how they manage compilers and libraries. A single test itself should not have to worry about the details of how a specific library is loaded on each specific machine and instead should be able to solely focus on preforming tests.

The Environment.yaml and the Environment class work in tandem to load compilers and libraries across different machines to remove this burden from tests.

Each machine used for testing will have its own unique Environment.yaml file and are in the form of the machines name followed by the `.yaml` file extension.

So for instance, the Cheyenne super computer's Environment.yaml file would be `cheyenne.yaml`, while Casper's would be `casper.yaml`.

The information within an Environment.yaml file includes: the number of CPUs to use for testing, the type of HPC scheduler that it may use (if any), and the compilers, libraries, MPI implementations. It also contains information on how compilers and libraries can be loaded: either by using the LMOD module command or by setting environment variables.

An Environment.yaml file contains two required section, with one optional section.

- Required Sections
  - `Description` - Describes general information about the machine
  - `Modsets` - Describes different compiler, MPI installation and library combinations
- Optional Sections
  - `PBS_OPTIONS` - Optional default options to be passed to PBS jobs
  - `SLURM_OPTIONS` - Optional default options to be passed to Slurm jobs

4.1 Description

The Description sections describes the machine in general and includes information to inform SMARTS of the machines name, the number of CPU's to use, the HPC type (if any), if the machine is to use the LMOD program to load compilers and libraries.

An example Description section that contains all necessary parts can be found in Listing 4.1.

Listing 4.1: Examlpe Cheyenne Environment.yaml Description

```
Description:
    Name: Cheyenne
    Max Cores: 4
    Modules: True
    LMOD_CMD: /glade/u/apps/ch/opt/lmod/8.1.7/lmod/libexec/
        lmod
    HPC: PBS
```

The example yaml file in Listing 4.1 is the Description section for the Cheyenne super computer. According to the description above, SMARTS will know that: it can load libraries, compilers and MPI implementations via the module command (see specifying libraries below); that Cheyenne is a super computer and uses the PBS scheduler; and that SMARTS can use up to four CPUs on the login nodes on Cheyenne.

The `LMOD_CMD` attribute is the path to the lmod command. On machines that use lmod, this can be found by printing the value of the `LMOD_CMD` environment variable.

NOTE: The `Max Cores` option in the Description specifies the number of maximum cores that can be used in the location where SMARTS is ran. So, if SMARTS is ran on the login node of Cheyenne, and `Max Cores` is set to `4`, then SMARTS will only use that many CPUs; however, this does not mean that a test cannot use more than 4 CPUs. A test could, for instance, launch a job via the HPC's batch node and use more than the specified amount in `Max Cores`.

4.2 HPC Options

The `HPC_Options` section of the Environment.yaml file specifies default options that can be passed to an HPC instance. For instance, a user might want to always have the workload manager send an email on a job completion to their email, or they may want to always run under a specific account key. If they do, they can specify those options and arguments in the `HPC_Options`.

The `HPC_Options` can contain any options that would be use in the job script for that machine's workload manager.

Tests will need to retrieve the `HPC_Options` dictionary from their environment class instance and pass it to `HPC.launch_job`. Tests can also edit existing dictionary items or add new ones by editing the `HPC_Options` dictionary.

Listing 4.2: Examlpe HPC_Options

```
HPC_OPTIONS:
  M: email@gmail.com
  q: regular
  j: oe
```

4.3 Modsets

The Modsets section describes compilers, MPI implementations, libraries and environment variables. Because Fortran libraries most often need to be built with the compilers they are built with,

modsets are used to describe combination of a single compiler, an MPI implementation and any number of libraries and environment variables.

A installation of a compiler, MPI implementation, or library can be described as either a module or as a environment variable combination.

Listing 4.3 contains an example Modset section with a single modset for a `INTEL-19.0.1` compiler, this specific example is take from the Cheyenne.yaml environment file.

Listing 4.3: Example Cheyenne Intel Modset

```
Modsets:
  ###############
  # INTEL-19.0.2
  ###############
  INTEL-19.0.2:
    Name: intel-19.0.2
    Compiler:
      Name: intel
      Version: 19.0.2
      Module: intel
      Executables:
        - ifort
        - icc
    MPI:
      Module: mpt
      Version: 2.19
      Executables:
        - mpicc
        - mpif90
    Libs:
      - p-netcdf:
        Name: PNETCDF
        Value: /glade/work/duda/libs-intel19.0.2
      - c-netcdf:
        Name: NETCDF
        Value: /glade/work/duda/libs-intel19.0.2
      - pio:
        Name: PIO
        Value: /glade/work/duda/libs-intel19.0.2
      - external_libs:
        Name: MPAS_EXTERNAL_LIBS
        Ealue: "-L${NETCDF}/lib -lhdf5_hl -lhdf5 -ldl -lz"
      - external_includes:
        Name: MPAS_EXTERNAL_INCLUDES
        Value: "-I${NETCDF}/include"
      - JASPERLIB:
        Name: JASPERLIB
        Value: "/glade/u/home/wrfhelp/UNGRIB_LIBRARIES/lib"
      - JASPERINC:
        Name: JASPERINC
```

```
41          Value: /glade/u/home/wrfhelp/UNGRIB_LIBRARIES/include
42      - use_pio2:
43          Name: USE_PIO2
44          Value: 'true'
45      - precision:
46          Name: PRECISION
47          Value: single
```

Modsets can contain three sections: the `Compiler`, `MPI`, and `Libs` section. A modset is required to contain the `Compiler` and `Libs` sections, but the `MPI` section is optional.

4.3.1 Compilers

The compiler section of a modset describes information needed to load a specific compiler. It includes the compilers: name (`Name`), version (`Version`), a list of compiler executables (`Executables`) and either the module name (`Module`), or the path to the compilers installation directory (`Path`).

While the `Name` keyword is required in the Compiler section, it is not actively used by SMARTS, but provides readability for users and for tests.

The `Version` keyword serves two purposes.

First, if `Version` is specified with the `Module` keyword, the name specified with the `Module` and the version number are appended together to load a specific version of the compiler. This is equivalent to running `module load gnu/8.3.0` or `module load gnu/9.1.0`.

Second, the `Version` keyword is used in conjunction with executables found under the `Executables` keyword to confirm the correct compiler has been loaded. See Chapter 5 for more information on how the executables section is used when loading a modset.

In the `Intel-19.0.2` modset from Listing 4.3, the compiler is specified with the name intel (denoted by the `Module` keyword), and, when loaded, will be loaded by using the lmod module Python command to load intel/19.0.2. This occurs in the same way one would load the `intel/19.0.2` using the module command via the command line: `module load intel/19.0.2`. Lastly, when loaded, SMARTs will test to see if the correct version of `ifort` and `icc`, which are listed in the executables, have been loaded correctly.

There must only be one compiler section per modset.

4.3.2 MPI

The `MPI` section specifies an MPI installation and is specified in the same manner as the compiler section. Upon being loaded, the executables listed in `Executables` will be tested against the version listed in the `Version` keyword.

The `MPI` section is not required to be present in a modset.

4.3.3 Libs

The `Libs` section can contain information on the installation of libraries and environment variables. The implementation of the `Libs` is meant to be flexible and allow tests to access libraries and needed environment variables.

Individual entries to the `Libs` section can specified as either a module (Listing 4.4) or an environment variable (Listing 4.5). Libraries can either be specified as with the `Module` and optional `Version` keywords as seen in Listing 4.4 or as a environment variable using the `Name` and `Value` keywords as seen in Listing 4.5.

Listing 4.4: Example Library Module

```
- netcdf:
  Module: netcdf
  Version: 3.6.3
```

Listing 4.5: Example Library Environment Variable

```
- netcdf:
  Name: NETCDF
  Value: /glade/work/duda/libs-intel19.0.2
```

The `Libs` section is required.

Chapter 5

The Environment Class

The Environment Class (smarts/env.py) is the interface that tests will use to load compilers, MPI installations and libraries that are specified in a Environment.yaml file.

Internally, the Environment Class loads and stores the Environment.yaml file that was passed to SMARTS on launch. However, the Environment Class contains public function which can be used to find and load modsets.

Each `run` method of a test will be passed an instance of the Environment class that has been loaded with the Environment.yaml that was specified by the `smarts.py` command line. Each test can use this instance of the Environment to load modsets as they choose.

The Environment class contains two public routines which tests can use to load modsets: `list_modsets` and `load_modset`, which are described in the two sections below.

5.1 List Modsets

Listing 5.1: list_modset Definition

```
1 def list_modsets(self, name=None, *args, **kwargs):
2     """ Return a list of modsets found in the parsed environment.yaml file, if
3     name is specified then modset names that contain that name will be
4     returned.
5
6     Keyword arguments:
7     name -- Name to specify specific modset(s) name (String)
8     """
```

The `list_modsets` command can be used retrieve the names of available modsets. These names can be used later by the `load_modset` command to load specific modsets. As seen in Listing 5.1, if `list_modsets` is ran and `name=None` then all the Modsets contained will be returned; however, if `name` is set, the Modsets that contain the name specified in name will be retrieved.

For instance, if SMARTS was initiated with a Environment.yaml file that contained the modsets:

- `GNU-9.1.0`
- `GNU-8.3.0`
- `INTEL-19.0.2`

The following calls to `Environment.list_modset` would return the following:

- `env.list_modset() - ['GNU-9.1.0', 'GNU-8.3.0', 'INTEL-19.0.2']`
- `env.list_modset(name='GNU-9') - ['GNU-9.1.0']`
- `env.list_modset(name='GNU') - ['GNU-9.1.0', 'GNU-8.3.0']`

5.2 Load Modsets

Listing 5.2: load_modset Definition

```
def load_modset(self, modsetName, *args, **kawrgs):
    """ Completely load the modset, modsetName to be used by a single test.
    This function completely loads a modset (compiler, mpi implementation, and
    all libraries).

    To load a compiler, this function will alter the PATH environment variable
    for the current process (single test) and prepend the compiler path to it.
    If the compiler is specified as a module, it will be loaded via the
    lmod Python interface (`module python load ...`)

    MPI implementation will be loaded in a similar manner to compilers.

    Both MPI and Compilers will be checked to ensure that the correct version
    is installed by running the compiler executables specified in the
    executables section of the compiler or MPI env.yaml sections with
    `--version` and checking the versions in the env.yaml file match correctly.

    Libraries will be loaded by creating ENV_NAME as an environment variable
    and assigning to it the value specified in value.

    modsetName -- Name of the modset to be loaded (String)
```

From a modset name can be used in the `load_modset` command to load a specific compiler i.e.:

```
gnu_modsets = env.list_modsets(name="GNU-9.1.0")
env.load_modset(gnu_modsets[0])
```

When `load_modset` is called with a valid modset, it will load the `Compilers`, `MPI` and `Libs` section in the following order and in the following way:

1. **Compiler**

   Depending on how the compiler is specified, the compiler will either be loaded using the lmod command or by using environment variables.

   - **Module** - If the compiler is specified with the `Module` keyword in the Environment.yaml file then it will be loaded using the lmod `module load` command. Doing so will alter the environment of the test in the same way that using the `module load compiler` would alter an environment.
   - **Environment Variable** - If the path of the compiler is specified as an environment variable, then SMARTS will prepend the compiler's `bin` directory to the `PATH` environment variable. Which enables a Python's subprocess or multiprocess command to launch that compiler.

In both cases, SMARTS will check to see if the correct compiler version has been loaded. This is done by running all of the executables listed in the `Executables` section with `--version` and checking to see if the version specified in `Version` is in the output.

2. **MPI**

   If specified, the MPI section will be loaded in the same way as the compiler section above and will be tested to ensure they are loaded correctly.

3. **Libs**

   Libraries will be loaded in the order that they are listed. Depending on how the library is specified, as either a module (`Module`) or as an environment variable value pair (`Name`, the library will be loaded using the lmod command or by creating an environment variable and setting it equal to value listed under value.

   For environment variable value pairs, SMARTS will create new environment variables with `Name` (or overwrite one if the name is already present) and set it to

   As opposed to how the `Compilers` and `MPI` sections are loaded, libraries are not tested that they are loaded correctly.