Technical Testsuite v1.0
User Guide

July 12, 2016

Contents

1 Introduction 2
2 Requirements 2
3 Quick start 2
4 Testsuite folder organization 3
5 Command line arguments 5
6 Test specification (`testlist.xml`) 6
7 Checker scripts 8
8 Updating or generating reference files 10

1 Introduction

The main goal of the testsuite is to check technical correctness of a COSMO model version (see section Section 6.5 of the COSMO coding standards). The testsuite is implemented using the Python programming language. Based on a set of given model configurations (which can be defined by the user, for example COSMO-RU, COSMO-EU or COSMO-2), the test suite creates modifications of this configuration and launches a short simulation for each of the new configurations. The software than verifies that:

- the simulations runs without aborting or crashing
- the results are within rounding error with respect to a reference simulation
- the code gives bit identical results with different processor configurations or I/O configurations
- restart functionality is working and gives bit identical results

Additional user defined verifications (so called "checkers") can be specified.

2 Requirements

The testsuite is written in Python and thus requires and installation of Python 2.6 with the standard modules `os`, `sys`, `string`, `struct`, `re`, `optparse`, `xml.etree`, `copy`, `math`, `subprocess`, `logging` installed. Since the testsuite typically runs within a batch job on a compute node, the Python installation must be available on the compute nodes.

3 Quick start

The different tests to be run are defined in an xml input file (`testlist.xml`). Before launching the test, a cosmo executable needs be compiled and copied in the main folder. The main program (`testsuite.py`) can then be called from a compute node with a list of command line arguments. Here is a typical example:

```
./testsuite.py -n 16 --exe=cosmo --color --mpicmd='aprun -n' -v 1
```

where -n refers to the total number of processors, `--exe` to the name of the executable, `--color` will print the test results in color, `--mpicmd='aprun -n'` sets the command used to launch the executable and `-v 1` set the verbosity. A detailed list of command line arguments accepted by the testsuite can be listed out by calling `./testsuite.py -h` on the command line.
For each test a set of so called checkers (see section 7) are called and return one of the following results: MATCH, OK, FAILED or CRASH. Depending on the verbosity level, results of individual checkers are displayed or only a summary of the result is echoed. A test is considered passed if it achieved a result of either MATCH or OK.

4 Testsuite folder organization

Figure 1 shows an overview of the organization of the testsuite root folder. The testsuite root folder contains the main program (`testsuite.py`) script as well as additional source files in the `tools/` and `checker/` folders. In addition to the source files, the main directory also contains a `data/` directory, which contains a folder for each type of test (e.g. cosmo7, cosmo2) with reference input and output files.

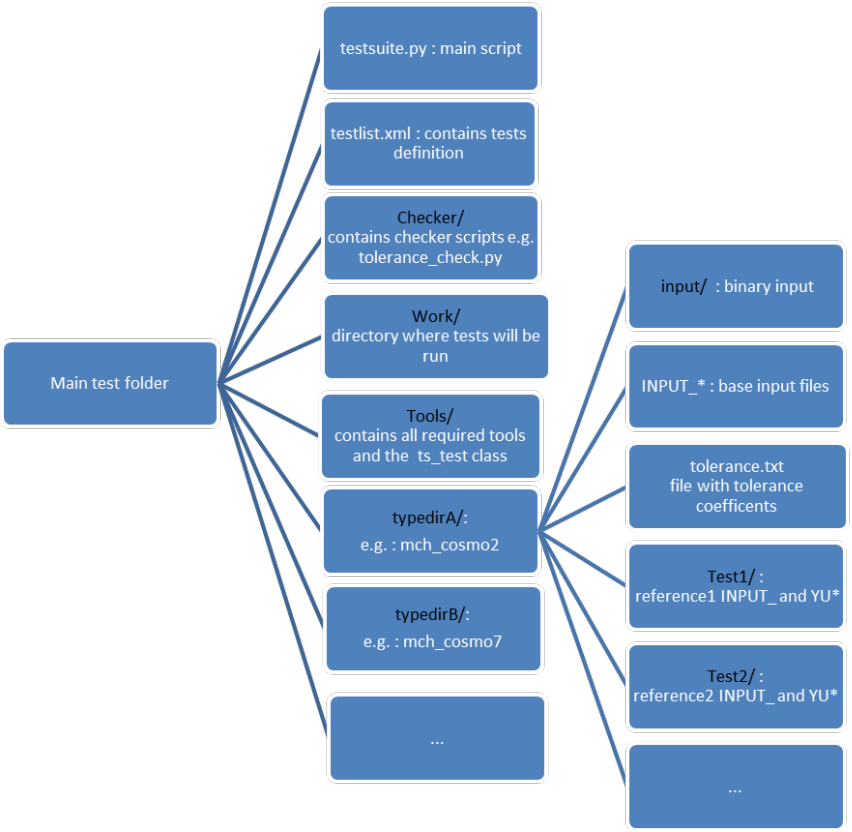

Figure 1: Test folder organization

When running the testsuite, for each of the tests defined in the `testlist.xml` input file, a separate directory `work/type/testname/` is created. First, the executable, namelists and auxiliary files are copied and the binary input linked into this folder. Then, the test simulation is run in this working directory. The working directories for the tests are not delete after

execution, if a test failed the user can thus go return manually to these directories for further investigation.

5 Command line arguments

Command line arguments are used to control the execution of the main `testsuite.py` script. Below is a complete list of available arguments, where the default values/options are given in square brackets:

`-h`, `--help`	Print the help message.
`-n` NPROCS	Number of processors (nprocx*nprocy+nprocio) to use [16].
`--nprocio`=NPROCIO	Set number of aynchronous IO processor [as specified in namelist]. If this argument is present it will override any values given in the namelist or `testlist.xml` file.
`-f`, `--force`	Do not stop upon error or fail [stop on error].
`-v` V_LEVEL	Verbosity level from 0 (quiet) to 3 (very verbose) [1].
`--mpicmd`=MPICMD	MPI run command (e.g. "mpirun -n") ["aprun -n"].
`--exe`=EXE	Executable file, [as specified in `testlist.xml`]. If this argument is present it will override any values given in the `testlist.xml`.
`--color`	Select colored output.
`--steps`=STEPS	Run only specified number of timesteps. If this argument is present it will override any values given in the `testlist.xml`.
`-w`, `--wrapper`	Use wrapper instead of executable for mpicmd (useful for OpenMPI on Mac computers).
`-o` STDOUT	Redirect standard output to selected file.
`-a`, `--append`	Appends standard output if redirection selected.
`--skip`=SKIP	Select which tests should be skipped (e.g. `--skip`=test_1,test_2).
`--update-namelist`	Use testsuite to update namelists. The tests will not be executed with this option.
`--update-yufiles`	Define new references by copying test output into the test reference folder. The tests will not be executed with this option.
`-l` TESTLIST, `--testlist`=TESTLIST	Select the xml testlist file [`testlist.xml`].
`--workdir`=WORKDIR	Name of working directory
`--tune-thresholds`	Enable automatic tuning of thresholds files
`--tuning-iterations`=ITER	Set the number of times tests should be executed. [10]
`--update-thresholds`	Update the thresholds based on the results of the current run.
`--reset-thresholds`	Sets all values in threshold files to 0.0 on the first run of a test

6 Test specification (`testlist.xml`)

The tests are defined in an XML file named `testlist.xml` which should be present in the testsuite root directory. It typically is of the following structure:

```
<?xml version="1.0" encoding="utf-8"?>
<testlist>

  <!-- ******* COSMO 7 ******* -->

  <test name="test_1" type="cosmo7">
    <description>Only dynamics</description>
    <checker>run_success_check</checker>
    <checker>tolerance_check</checker>
    <changepar file="INPUT_ORG" name="hstop">1</changepar>
    <autoparallel>1</autoparallel>
  </test>

  <test name="test_2" type="cosmo7">
    <description>Dynamics and physics</description>
    <namelistdir>cosmo7/test_1</namelistdir>
    <refoutdir>cosmo7/test_2</refoutdir>
    <checker>run_success_check</checker>
    <checker>tolerance_check</checker>
    <changepar file="INPUT_ORG" name="hstop">1</changepar>
    <changepar file="INPUT_ORG" name="lphys">.TRUE.</changepar>
    <autoparallel>1</autoparallel>
  </test>

 <\testlist>
```

The test list is embeded within a <**testlist**> tag. Each test is defined by a <**test**> tag, which has two compulsory attributes *name* and *type*. In the above example two tests are defined. The first test (`test_1`) will run with namelists and reference data from the `data/cosmo7/test_1` directory. Since the directory containing the namelists and reference data are named like the test, they do not need to be further specified. Two checkers will be run, one assuring that the simulation has run correctly and did not crash and one which will check that the results are within thresholds. The namelist parameter `hstop` is changed to stop the simulation after one hour. Test two uses the same namelists as test one but has its own reference files, thus the directories are specifically given with the optional `namelistdir` and `refoutdir` tags. Additionally, the namelist switch `lphys` is activated, in order to run with physics.

Below a list of the valid XML tags for defining a given test is given. Tags and/or attributes which are optional and need not mandatorily be specified for each test are indicated with an asterixs (*).

<description> A short description of the test which will be printed out by the testsuite.

<namelistdir*> Directory where to find the namelist (INPUT_*) files for this test. Defaults to data/type/name, where type and name correspond to the type and name of the test, respectively.

<refoutdir*> Directory which contains the reference output files (YU*). Defaults to the directory which contains the namelist (see above). If the path starts with "../", it will be relative to the running directory, i.e. "../test_3" is equivalent to "work/typedir/test_3". This can be use to compare against another test which has already run.

<depend*> Path to test directory on which this test depends. If the path starts with "../", it will be relative to the running directory.

<checker> Name of the checker to be called with this test. This tag can appear several times to apply more than one checker.

<executable*> Define the name of the executable. This tag is only considered if no executable name was given as a command line argument.

<changepar* file=*file* name=*name* occurence*=*occurence*>
Modify the parameter *name* in the namelist *file* to a new value for the current test. The optional *occurence* attribute can be used to modify a specific occurrence of the parameter in the namelist.

<autoparallel*> Number to select a domain decomposition (nprocx x nprocy) within a automatically generated list of decomposition possibilities. Running two tests with different autoparallel numbers will enable to compare runs with different processor configurations.

7 Checker scripts

A checker is an external script which is called from the main testsuite program to verify the correctness of a specific test. More than one checker can be applied to an individual test. For example, the user might want to check whether the run was successful, whether output was produced and whether results are within specified thresholds. The checkers can be written in any scripting language (bash, python ...). Communication of runtime variables between the testsuite and a checker is achieved via environment variables. A checker can access a set of environment variables defined by the testsuite.

`TS_BASEDIR`	Root directory of the testsuite.
`TS_VERBOSE`	Verbosity level requested by the user for running the testsuite.
`TS_RUNDIR`	Directory where the current test was run.
`TS_LOGFILE`	File which contains the standard output and standard error of the current test.
`TS_NAMELISTDIR`	Directory containing the namelist of the current test.
`TS_REFOUTDIR`	Directory containing the reference files for the current test.
`TS_TUNE_THRESHOLDS`	Shows whether or not tuning was activated
`TS_TUNING_ITERATIONS`	Number of times each test will be executed if tuning of tolerance files is activated.
`TS_RESET_THRESHOLDS`	Shows if the user activated the reset threshold function.

Each checker should return an exit code with the following definition:

0	MATCH	Results match bit-by-bit as compared to the reference.
10	OK	Results match the reference within defined thresholds.
15	SKIP	Test not applicable and thus skipped.
20	FAIL	Test failed, results are outside of thresholds.
30	CRASH	Test failed due to model crash.

The checkers to be called for a given test are defined within the `testlist.xml` file. All checkers are called irrespective of the exit code of preceding tests. The overall status of a test is defined as the maximum of the exit codes of all checkers called.

7.1 The run success checker

The `run_success_check.py` checker will check the successful execution of the test. It searches the standard output of the simulation for the typical `CLEAN UP` message issued at the end of a successful model simulation.

7.2 The grib output checker

The `existence_grib_out.sh` checker will check whether a simulation has correctly written at least one GRIB output file (e.g. `lfff00000000`) and that this file is of non-zero size.

7.3 The identical checker

When bit identical results are expected, as for example when comparing two different parallelizations on a given system, the `identical_check.py` checker should be called. This checker expects a `YUPRTEST` output file (see below) and absolute differences between the output of the test and the reference file are compared to within a zero tolerance threshold.
Note that bit identical results can only be expected when two simulations are run on the same system. A test using the `identical_check.py` should always have reference directory pointing to a previous test in the working folder, e.g. <`refoutdir`>`../test_3`<`refoutdir`>.

7.4 The tolerance checker

In order to validate COSMO results a specific output file named `YUPRTEST` output file is used. This file, which contains double precision values of the min, max and mean of the prognostic variables on each model level at specific time steps, is written out when the parameter `ltestsuite=.TRUE.` is activated in the `DIACTL` namelist group in the `INPUT_DIA` namelist file. When compiling the code on a different architecture, or when introducing some technical modification (e.g. optimization), one does not expect bit identical results with respect to the previous model version. The `tolerance_check.py` script will compare results to a reference but accounting for rounding errors and allowing to compare output files within some tolerance values.
The definition of meaningful thresholds is not a straightforward task as they should be low enough to detect bugs in the software, while allowing to tolerate acceptable rounding error differences. So far the following approach was adopted to determine these thresholds:

- two COSMO executable are generated with two different compilers
- at each time step a perturbation is added to the prognostic variables

$$f = f\,(1 + R\,\epsilon) \;\text{ with }\; \epsilon = 1^{-15} \;\text{ and }\; R \;\text{ random a perturbation} \tag{1}$$

The maximum errors obtained for different time steps by running and comparing these two executables N times (typically N=30) are used as threshold values.

7.5 Creation of Tolerance Files

The tolerance checker is able to tune these files automatically. The tuning process starts the same way a normal run of the tolerance checker would, with the comparison of the two `YUPRTEST` files. The compare phase is followed by the tuning of failed comparisons this is accomplished by increasing the corresponding levels threshold with a predefined factor. The checker always returns OK during the tuning process. The argument `--tune-thresholds` will enable automatic tuning mode which will tune the tolerance files.
The process of tuning can be furthermore configured by using the following arguments:

- `--tuning-iterations=N` - Specifies the number of tuning iterations. The default is 10.
- `--reset-thresholds` - The tolerance files corresponding to the current test will be reset on the first run. So that each value is set to 0.0 which results in a tuning from scratch.
- `--update-thresholds` - Update the thresholds based on the results of the current run.

7.6 Custom checkers

Users are encouraged to develop, document and share their own checker scripts which can be compiled and distributed with future versions of the testsuite.

8 Updating or generating reference files

When a new version of COSMO with significant changes is released, for instance with a new physics scheme or with new tuning parameter values, the reference files (e.g. `YUPRTEST`) need to be updated. To this end, the `testsuite.py` script can be called with the `--update_yufiles` option, which will copy the reference files from the test directories back into the reference directories. Care should be taken that this is only done under the following conditions:

- The user is sure that all tests run successfully and the eventual failing tests are due to model changes (which have been thoroughly checked and tested).
- The testsuite has been previously run fully (using the `--force` option) and without specifying only a limited number of tests (e.g. `--skip=` or `--only=`) or a limited number of timesteps (e.g. `--steps=`).