ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

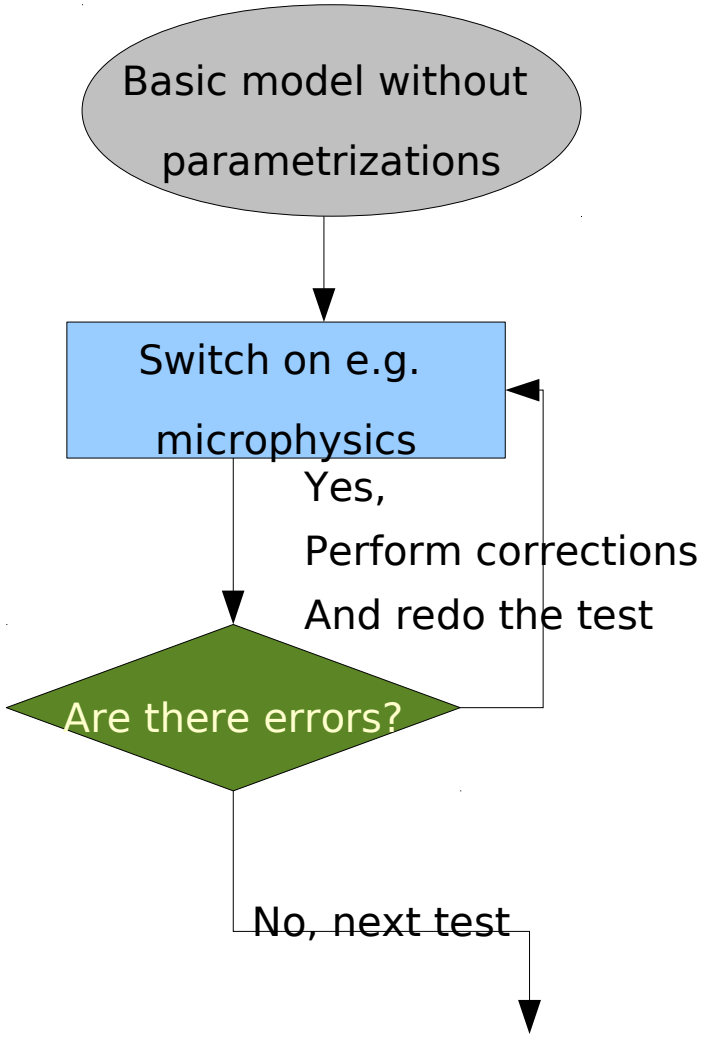
Technical Testsuite v2.0 for COSMO

N. Lardelli, X. Lapillonne, O. Fuhrer

Why a Technical Testsuite?

- Implementation errors are sometimes difficult to detect and to find. Need for some systematic method to test the code before doing a full meteorological/climatological verification.

- For executables compiled from very large codes (like **cosmo**), a “tassonomic” approach of debug is more effective.



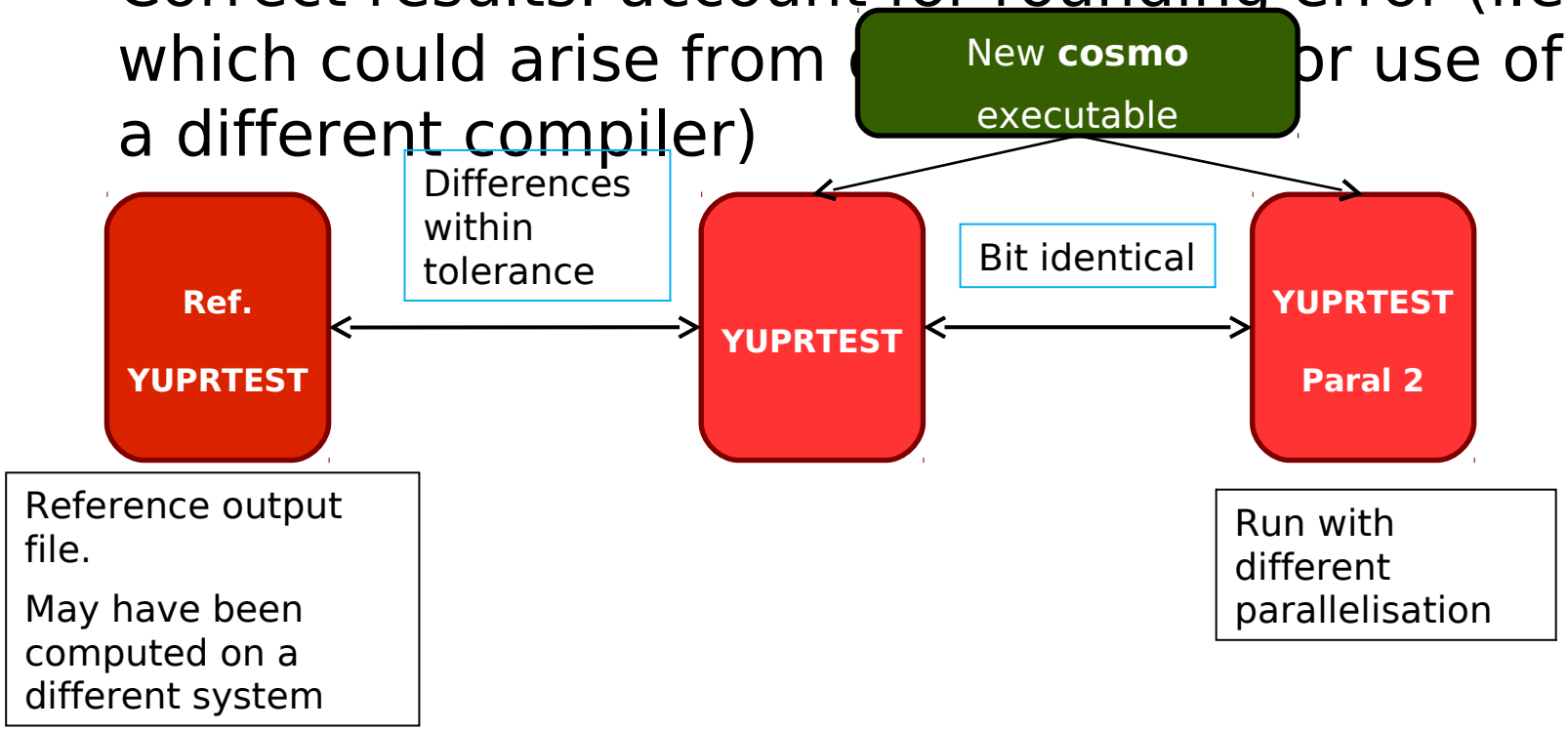
What are the requirements?

- The code is running and gives correct results with various configurations (e.g. only dynamics, dynamics + physics, members configurations ...)
- The code stops executing correctly
- The code gives bit identical results with different processor configurations (including with or without I/O PE)
- Restart functionality is working, and gives bit identical results
- Additional user defined verification can be easily added

Verifying Cosmo results

- ASCII output file (YUPRTEST) : double precision mean, max and min values at each vertical level of the prognostic fields

- Correct results: account for rounding error (i.e. which could arise from [illegible] or use of a different compiler)

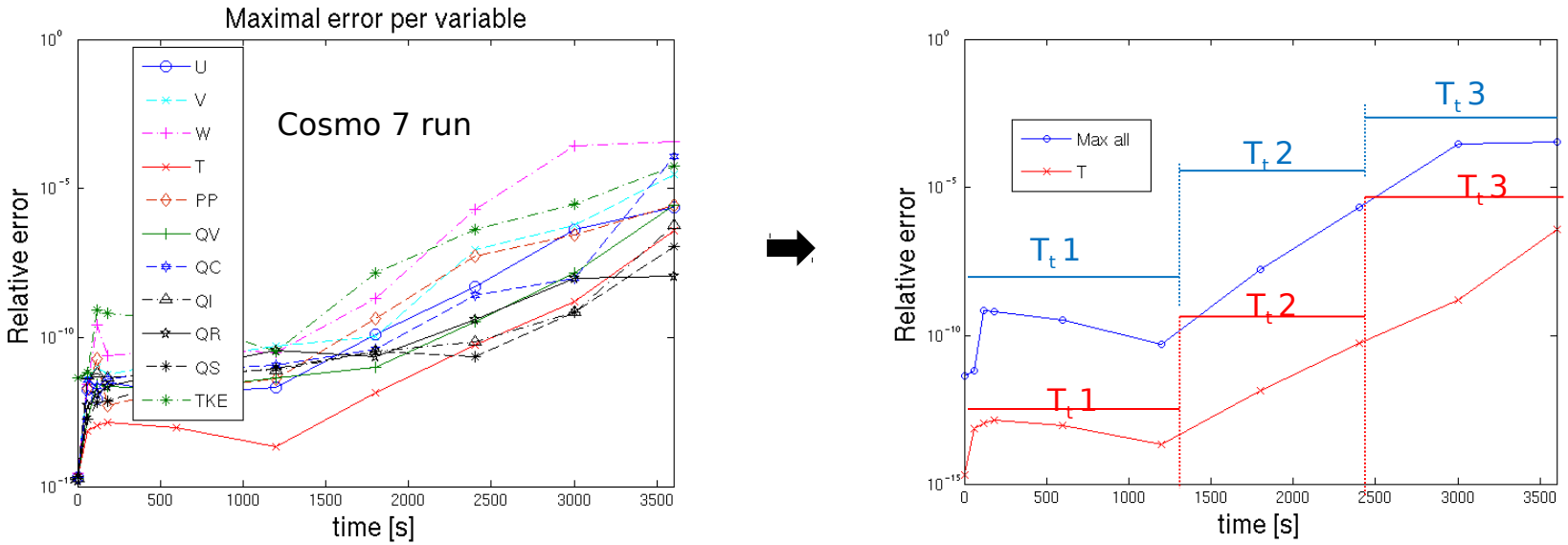


Setting the tolerance factor

How to account for rounding error propagation ?

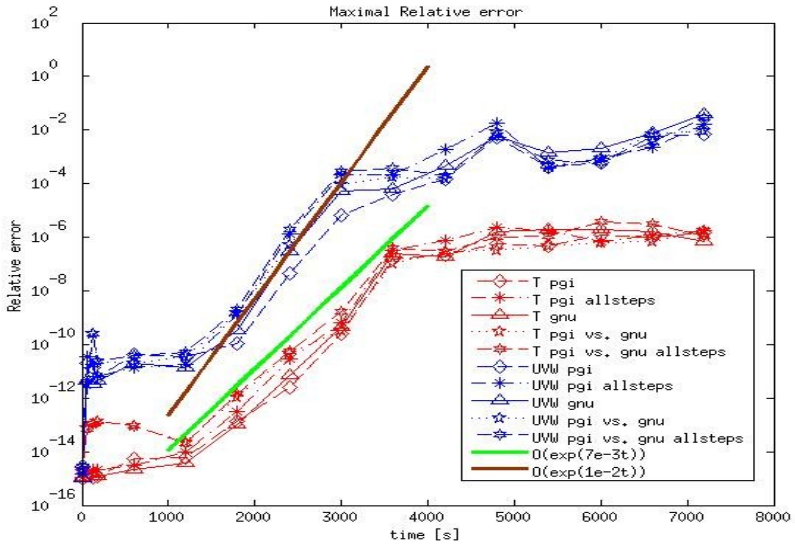
Methodology:

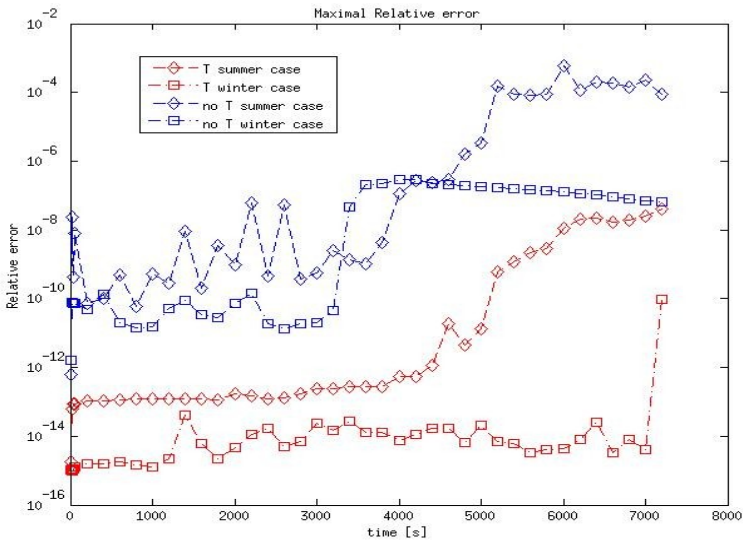
- 2 perturbed cosmo executables compiled with different compilers
- At each step a perturbation is added to the prognostic fields:
  $f = f * (1+R*\varepsilon)$, with R random array and $\varepsilon = 1^{-15}$
- Run 30 experiments, compute maximal differences for each prognostic variables



- Reduce tolerance number parameters : two groups of variables, T and All prognostics
- Set threshold for this 2 groups, for different time intervals
- Threshold can be set differently for different cases (e.g. cosmo2 or cosmo7)

Threshold under different conditions





- The behaviour of the error growth is similar under different compilers, but different with different scales and weather situations

- It may be wise to have an error growth study for every weather situation, topology and grid size we want to use as basis for the tests

Running testsuite.py

- The different tests are defined in an input file “testlist.xml”
- The script ./testsuite.py can be called with

```
lapixa/testsuite_nicolo> ./testsuite.py -n 16 --color -f --exe=cosmo_gnu --steps=10 --mpicmd='aprun -n' -v 0

////////////////////////////////////////////////////////
// BEGINING TESTSUITE
//

------------------------------------------------------------
Starting cosmo7/TEST_1,  Only Dynamics
*** cosmo7/TEST_1 : OK

------------------------------------------------------------
Starting cosmo7/TEST_2,  Dynamics + Physics
*** cosmo7/TEST_2 : OK

------------------------------------------------------------
Starting cosmo7/TEST_3,  Dynamics + Physics + Observations
*** cosmo7/TEST_3 : OK

------------------------------------------------------------
Starting cosmo7/TEST_3p,  Parallel Test
*** cosmo7/TEST_3p : MATCH

------------------------------------------------------------
Starting cosmo7/TEST_3pio, Parallel Test no IO processors
*** cosmo7/TEST_3pio : MATCH
```

Test is passed for OK or MATCH results

Other possible outcome are FAIL or CRASH

Test definition

- The tests are defined in an xml file: testlist.xml
- Parsed with the help of xml.etree.ElementTree
  ary

```
<test name="TEST_3" type="cosmo7">
  <description> Dynamics + Physics + Observations </description>
  <namelistdir>cosmo7/TEST_3/</namelistdir>
  <refoutdir>cosmo7/TEST_3/</refoutdir>
  <checker>run_success_check</checker>
  <checker>tolerance_check</checker >
  <!-- set hstop to 2 to be able to check restart later -->
  <changepar file="INPUT_ORG" name="hstop"> 2 </changepar>
  <autoparallel>1</autoparallel>
 </test>
<!-- ------------------------------------------------------------ -->
 <test name="TEST_3p" type="cosmo7">
  <description> Parallel Test </description>
  <namelistdir>cosmo7/TEST_3/</namelistdir>
  <refoutdir>../TEST_3/</refoutdir> <!-- previous TEST_3 run -->
  <depend>../TEST_3</depend>
  <checker>run_success_check</checker>
  <checker>identical_check</checker >
   <changepar file="INPUT_ORG" name="nstop"> 20 </changepar>
   <autoparallel>2</autoparallel>
 </test>
```

Path to input and refrence files

Checkers definition

changepar can be used to modify the reference namelist

Set automatic parallelization

For the parallel test, the reference directory is the previous run

Calling identical check

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

testsuite.py command line arguments

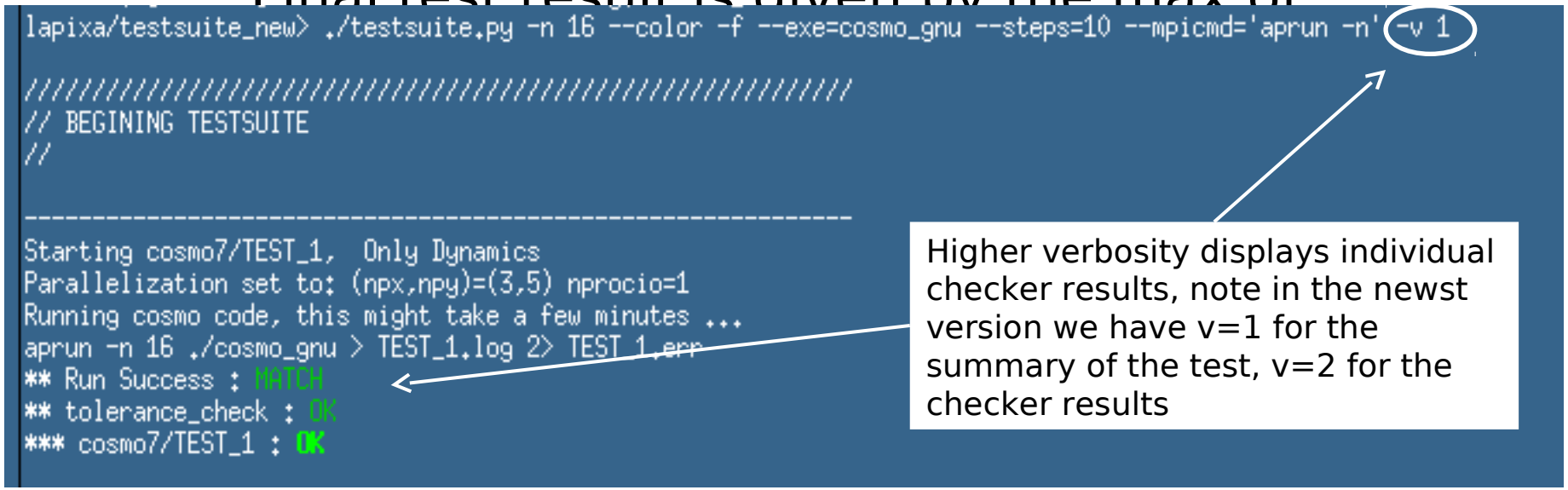
```
lapixa/testsuite_new> ./testsuite.py -h
Usage: testsuite.py [options]

Desc. : this script run a series of tests defined in testlist.xml. For each test a set of checks are carried out.

Options:
  -h, --help            show this help message and exit
  -n NPROCS             Number of processors. The parameters nprocx, nprocy and nprocio are then set automatically by the script.
  --nprocio=NPROCIO     Set number of aynchronous IO processor, def=From Namelist
  -f, --force           Do not stop upon error.
  -v V_LEVEL            Verbose level -1 to 2, def=0
  --mpicmd=MPICMD       mpiexe command, def=aprun.
  --exe=CEXE            executable file, def=From Namelist
  --color               Select colored output
  --steps=STEPS         Run only specified number of timesteps.
  -w, --wrapper         Use wrapper instead of executable for mpicmd.
  -a, --append          Appends standard output if a redirection of the standardoutput is required.
  -o STDOUT             Redirect standard output to selected file.
  --skip=SKIP           Select which test with the given prefix need to be skipped.
  --update_namelist     Use Testsuite for generation of new namelists.
  -l TESTLIST, --testlist=TESTLIST
                        Select the testlist file
```

- In case of execution of “python ./testsuite.py”, default values are used

- In the current version auxiliary parameters still as hardcoded variables

The checkers

- For each test a set of checkers can be called
- Checker : script (could be written in any language) that return one of the following exit code:
  0 : MATCH, 10 : OK , 20 : FAIL, 30 : CRASH
- Final test result is given by the max of



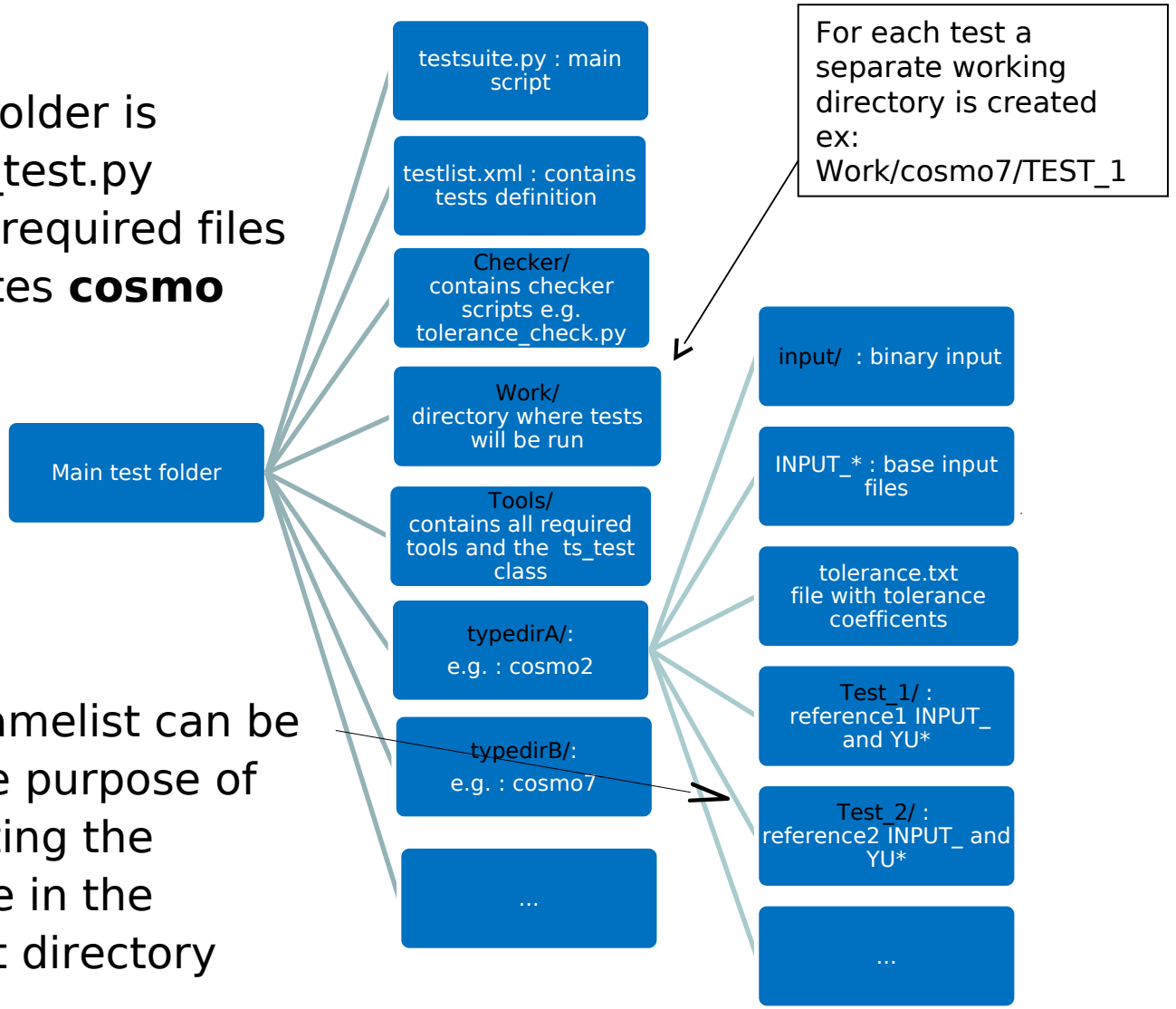
Implementation of new checkers

- The script can access run time environment variables (TS_BASEDIR, TS_NAMELISTDIR, TS_VERBOSE …) set by the class ts_test.py
- The idea is that each user can add his own custom checker (ex: checking that a specific output file exists)
- Either one defines new implementations of checker_base_wrapper.py:
  - Explicit implementation of the static methods name() and check()
  - Filename ending in “_check.py”
- Or one lets auto_check.py take care of a generic checker routine:
  - The routine doesn't have to end in “_check.py”
  - The routine takes no argument and return a value among {0,10,20,30}

The testsuite directory

- After the folder is created ts_test.py copies the required files and executes **cosmo** locally

- The option --update_namelist can be used for the purpose of soley updating the namelist file in the original test directory



Conclusions

- There are evident error growth behavior in the comparison of results between different compilers that can be used to distinguish erroneous part in the **cosmo** executable

- The current version of Testsuite allows users to add new checkers fast

- A complete study of error growth should be performed for every test definition in order to have tuned parameters for tolerance

Status with respect to COSMO coding document: *6.5 Standard Test Suite*

All versions have to pass a standard test suite, which checks some technical issues. The idea is to define such a test suite, that can easily be run at every center. Issues to be checked are for example:

- Portability (not in testsuite.py)
- Independence of processor configurations (MPI and OpenMP) (ok)
- Reproducibility of results with older versions (ok)
- Restart functionality (ok)
- I/O with Grib/NetCDF (possible)
- Tests with array bound checking (not in testsuite.py, user responsibility)
- Possibility to run with input data from different models (GME, IFS, ERA, etc.) (ok, needs reference input files)

- Timings / efficiency (possible, but difficult to get a portable solution)