# ecBuild, ctest, debug and adding test

JEDI Academy
June, 2018

# ecBuild

## https://github.com/ecmwf/ecbuild


- A CMake-based build system, consisting of a collection of CMake macros and functions that ease the managing of software build systems.

- There is no actual build required for this software. It is a collection of text files, scripts and CMake files.

- ecBuild/cmake must be called from an out-of-source build directory and forbids in-source builds.

# Usage

ecbuild [option...] [--] [cmake-argument...] <path-to-source>

- -prefix=<prefix>

  Set the install path to <prefix>.

--build=<build-type>

  Set the build-type to <build-type>: debug; release; bit

--log=<log-level>

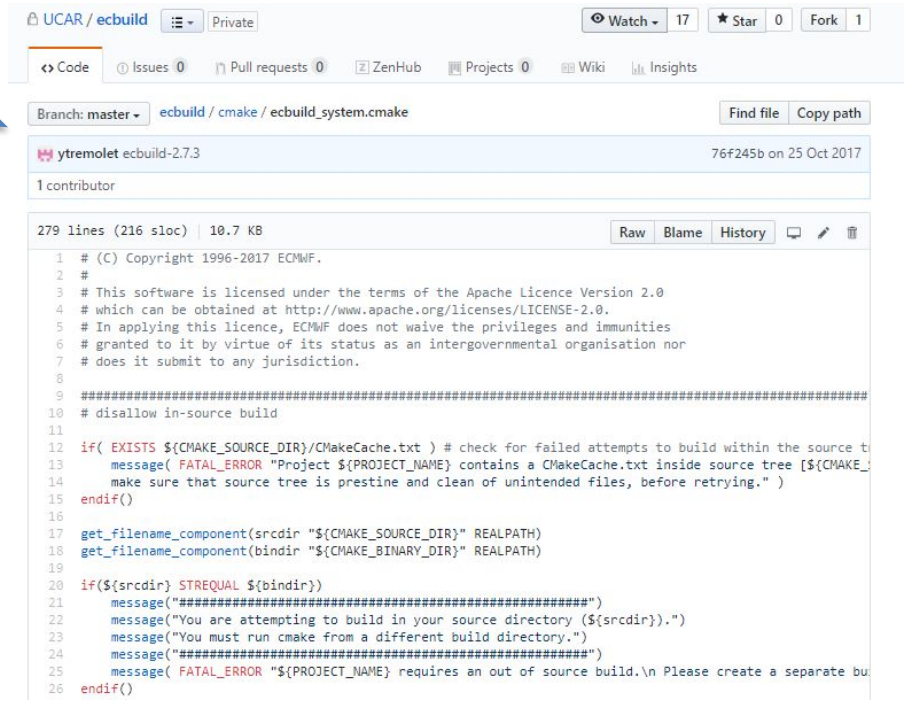
  Set the ecbuild log-level: DEBUG; INFO; WARN, ……

# Structure of CMakeLists.txt using ecBuild

1. # header

```
cmake_minimum_required( VERSION 3.3.2 FATAL_ERROR )
project(oops C CXX Fortran)
set( CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_CURRENT_SOURCE_DIR}/cmake" )
include( ecbuild_system NO_POLICY_SCOPE )
ecbuild_requires_macro_version( 2.7 )
```

2. # open project
3. # sources
4. # finalize project



# Structure of CMakeLists.txt using ecBuild


1. # header
2. # open project

```
ecbuild_declare_project()   # initialize an ecBuild project
```

3. # sources
4. # finalize project

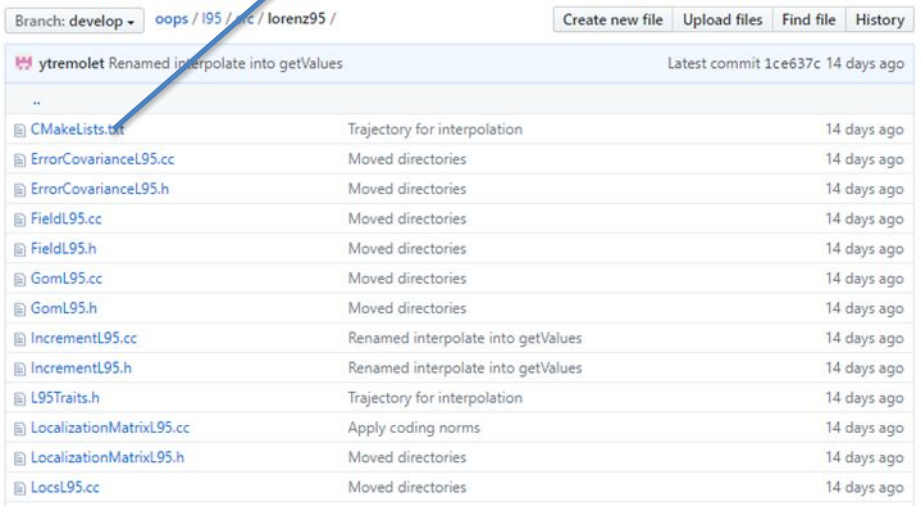
# Structure of CMakeLists.txt using ecBuild

1. # header
2. # open project
3. # sources

   *add_subdirectory (lorenz95 )*
   *add_subdirectory ( executables )*

4. # finalize project

```
ecbuild_add_library ( TARGET     lorenz95
                      SOURCES
                           ErrorCovarianceL95.cc
                           ErrorCovarianceL95.h
                           FieldL95.cc
                           FieldL95.h
                           ..........
                           LocalizationMatrixL95.cc
                           LocalizationMatrixL95.h
                      LIBS       oops
                      INSTALL_HEADERS LISTED
                      LINKER_LANGUAGE ${OOPS_LINKER_LANGUAGE}
)
```



# Structure of CMakeLists.txt using ecBuild

1. # header
2. # open project
3. # sources

```
add_subdirectory (lorenz95 )
add_subdirectory ( executables )
```

4. # finalize project

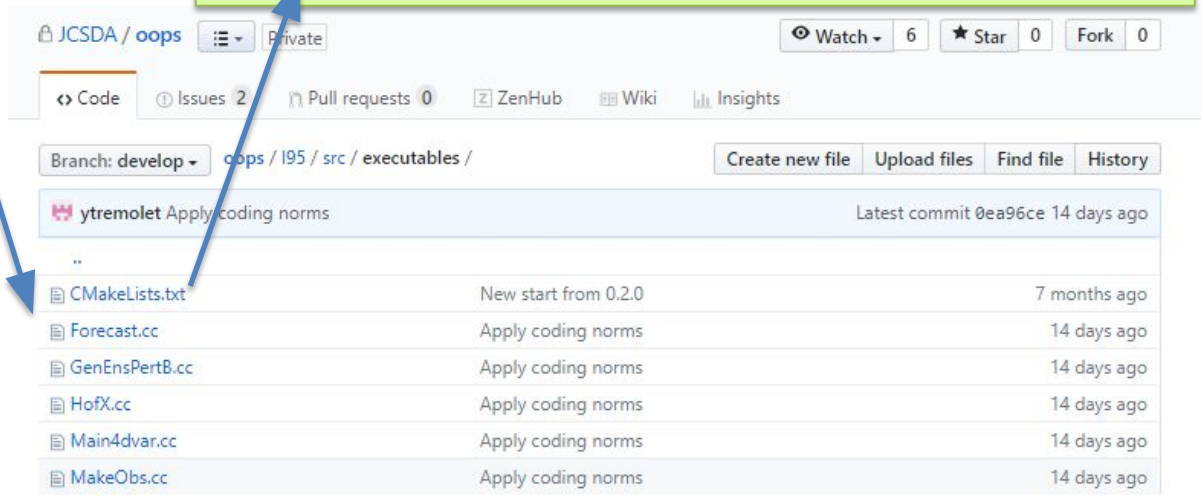
```
ecbuild_add_executable ( TARGET  l95_forecast.x
                         SOURCES Forecast.cc
                         LIBS    lorenz95 )

ecbuild_add_executable ( TARGET  l95_genpert.x
                         SOURCES GenEnsPertB.cc
                         LIBS    lorenz95 )

ecbuild_add_executable ( TARGET  l95_4dvar.x
                         SOURCES Main4dvar.cc
                         LIBS    lorenz95 )

ecbuild_add_executable ( TARGET  l95_makeobs.x
                         SOURCES MakeObs.cc
                         LIBS    lorenz95 )

ecbuild_add_executable ( TARGET  l95_hofx.x
                         SOURCES HofX.cc
                         LIBS    lorenz95 )
```



# Structure of CMakeLists.txt using ecBuild


1. # header
2. # open project
3. # sources
4. # pkgconfig exports (*optional*)

5. # finalize project

# Structure of CMakeLists.txt using ecBuild


1. # header
2. # open project
3. # sources
4. # finalize project

```
ecbuild_install_project( NAME ${PROJECT_NAME} )

# print the summary of the configuration
ecbuild_print_summary()
```

# Structure of CMakeLists.txt using ecBuild


```
# header
cmake_minimum_required( VERSION 3.3.2FATAL_ERROR )
project( oops C CXX Fortran )
set( CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_CURRENT_SOURCE_DIR}/cmake" )
include( ecbuild_system NO_POLICY_SCOPE )
ecbuild_requires_macro_version( 2.7 )

# open project
ecbuild_declare_project()

# sources
add_subdirectory( lorenze95 )
add_subdirectory( executables )

# finalize project
ecbuild_install_project( NAME ${PROJECT_NAME} )
ecbuild_print_summary()
```

# Add Dependencies


```
# header
cmake_minimum_required( VERSI
project( oops C CXX Fortran )
set( CMAKE_MODULE_PATH ${CMAKE_M
include( ecbuild_system NO_POLIC
ecbuild_requires_macro_version( 2

# open project
ecbuild_declare_project()

# Dependencies
.....
.....

# sources
add_subdirectory( lorenze95 )
add_subdirectory( executables )

# finalize project
ecbuild_install_project( NAME ${PR
ecbuild_print_summary()
```

```
# Boost, use cmake command

set( Boost_MINIMUM_VERSION "1.47" )
find_package( Boost ${Boost_MINIMUM_VERSION} REQUIRED
              COMPONENTS unit_test_framework )
include_directories( ${Boost_INCLUDE_DIR} )
```

```
# LAPACK, use ecbuild_add_option

ecbuild_add_option( FEATURE LAPACK
                    DESCRIPTION "Linear Algebra PACKage"
                    REQUIRED_PACKAGES "LAPACK REQUIRED" )
if (NOT HAVE_LAPACK)
  ecbuild_error("LAPACK_LIBRARIES not defined")
endif()
```

```
# eckit, Use another CMake project as a
# dependency by either building it from source or searching for it.

ecbuild_use_package( PROJECT eckit VERSION 0.18.0 REQUIRED )
include_directories( ${ECKIT_INCLUDE_DIRS} )
```

# ecBuild Bundle

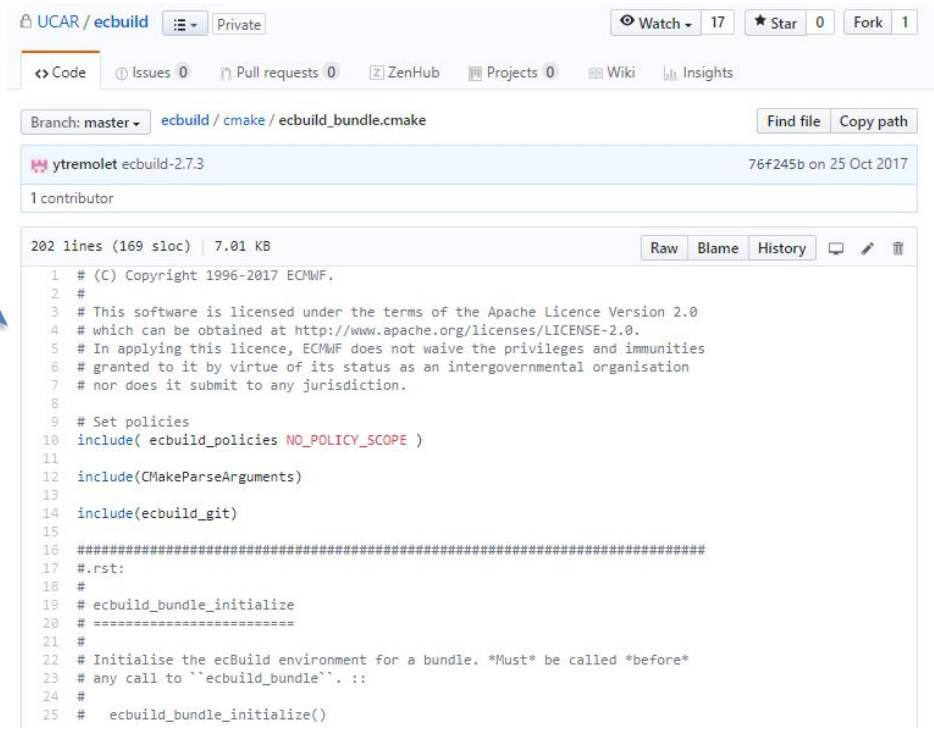
- A bundle is used to build a number of projects together.
- Each subproject needs to be declared with a call to *ecbuild_bundle*
- The order of projects is important and needs to respect dependencies: *( if project B depends on project A, A should be listed before B in the bundle)*
- Subprojects are configured and built in order.

# Structure of CMakeLists.txt using ecBuild

1. ### header

```
cmake_minimum_required( VERSION 3.3.2 FATAL_ERROR )
project(oops C CXX Fortran)
set( CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_CURRENT_SOURCE_DIR}/cmake" )
include( ecbuild_bundle )
ecbuild_bundle_initialize()
ecbuild_requires_macro_version( 2.7 )
```

2. ### Sources
3. ### finalize



# Structure of CMakeLists.txt using ecBuild


1. ### header

```
cmake_minimum_required( VERSION 3.3.2 FATAL_ERROR )
project(oops C CXX Fortran)
set( CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_CURRENT_SOURCE_DIR}/cmake" )
include( ecbuild_bundle )
ecbuild_bundle_initialize()
ecbuild_requires_macro_version( 2.7 )
```

2. ### Sources

```
ecbuild_bundle( PROJECT eckit    GIT "https://github.com/ECMWF/eckit.git"     TAG 0.18.5 )
ecbuild_bundle( PROJECT fckit    GIT "https://github.com/ECMWF/fckit.git"     TAG 0.5.0  )
ecbuild_bundle( PROJECT crtm     SOURCE "${HOME}/jedi/crtm"                   BRANCH develop   UPDATE )
```

……

```
ecbuild_bundle( PROJECT fms      GIT "https://github.com/JCSDA/fms.git"       BRANCH develop   UPDATE )
ecbuild_bundle( PROJECT fv3      GIT "https://github.com/JCSDA/fv3.git"       BRANCH develop   UPDATE )
ecbuild_bundle( PROJECT fv3-jedi GIT "https://github.com/JCSDA/fv3-jedi.git" BRANCH develop   UPDATE )
```

3. ### finalize

```
ecbuild_bundle_finalize()
```

# Basic ctest usage in JEDI

- https://jointcenterforsatellitedataassimilation-jedi-docs.readthedocs-hosted.com/en/latest/developer/building_and_testing/unit_testing.html#
- List all available tests
  - ctest –N

```
Singularity JCSDA-singularity-master-latest.simg:~/jedi/build/ufo-bundle> ctest -N
Test project /home/xinzhang/jedi/build/ufo-bundle
  Test   #1: eckit_test_config_resource
  Test   #2: eckit_test_config_configuration
  Test   #3: eckit_test_container_sharedmemarray
  Test   #4: eckit_test_container_btree
  Test   #5: eckit_test_container_bloomfilter
  Test   #6: eckit_test_container_trie
  Test   #7: eckit_test_container_densemap
  Test   #8: eckit_test_container_cache_lru
  Test   #9: eckit_test_container_benchmark_densemap
  Test  #10: eckit_test_filesystem_multihandle
……
```

# Basic ctest usage in JEDI

- Run a group of tests, for example, run all tests with the names have “ufo”
  - ctest -R ufo

```
Singularity JCSDA-singularity-master-latest.simg:~/jedi/build/ufo-bundle> ctest -R ufo
Test project /home/xinzhang/jedi/build/ufo-bundle
    Start 225: test_ufo_geovals
1/9 Test #225: test_ufo_geovals ..................   Passed    1.38 sec
    Start 226: test_ufo_amsua
2/9 Test #226: test_ufo_amsua ....................   Passed    2.38 sec
    Start 227: test_ufo_radiosonde
3/9 Test #227: test_ufo_radiosonde ...............   Passed    0.37 sec
    Start 228: test_ufo_insitutemperature
……
    Start 231: test_ufo_aod
7/9 Test #231: test_ufo_aod ......................   Passed    0.89 sec
    Start 232: test_ufo_obsop_seaice_tlad
8/9 Test #232: test_ufo_obsop_seaice_tlad ........   Passed    1.88 sec
    Start 233: test_ufo_obsop_rsonde_tlad
9/9 Test #233: test_ufo_obsop_rsonde_tlad ........***Exception: SegFault  0.49 sec

89% tests passed, 1 tests failed out of 9

Label Time Summary:
boost        =   8.03 sec (9 tests)
executable   =   8.03 sec (9 tests)
mpi          =   0.37 sec (1 test)
ufo          =   8.03 sec (9 tests)

Total Test time (real) =   8.04 sec

The following tests FAILED:
        233 - test_ufo_obsop_rsonde_tlad (SEGFAULT)
Errors while running CTest
```

# Debug failed test

- Run specific failed test
  - export OOPS_DEBUG=1  ! Turn on the debug log
  - export OOPS_TRACE=1  ! Turn on the trace log
  - ctest -R test_ufo_obsop_rsonde_tlad
- Check the log
  - cat Testing/Temporary/LastTest.log

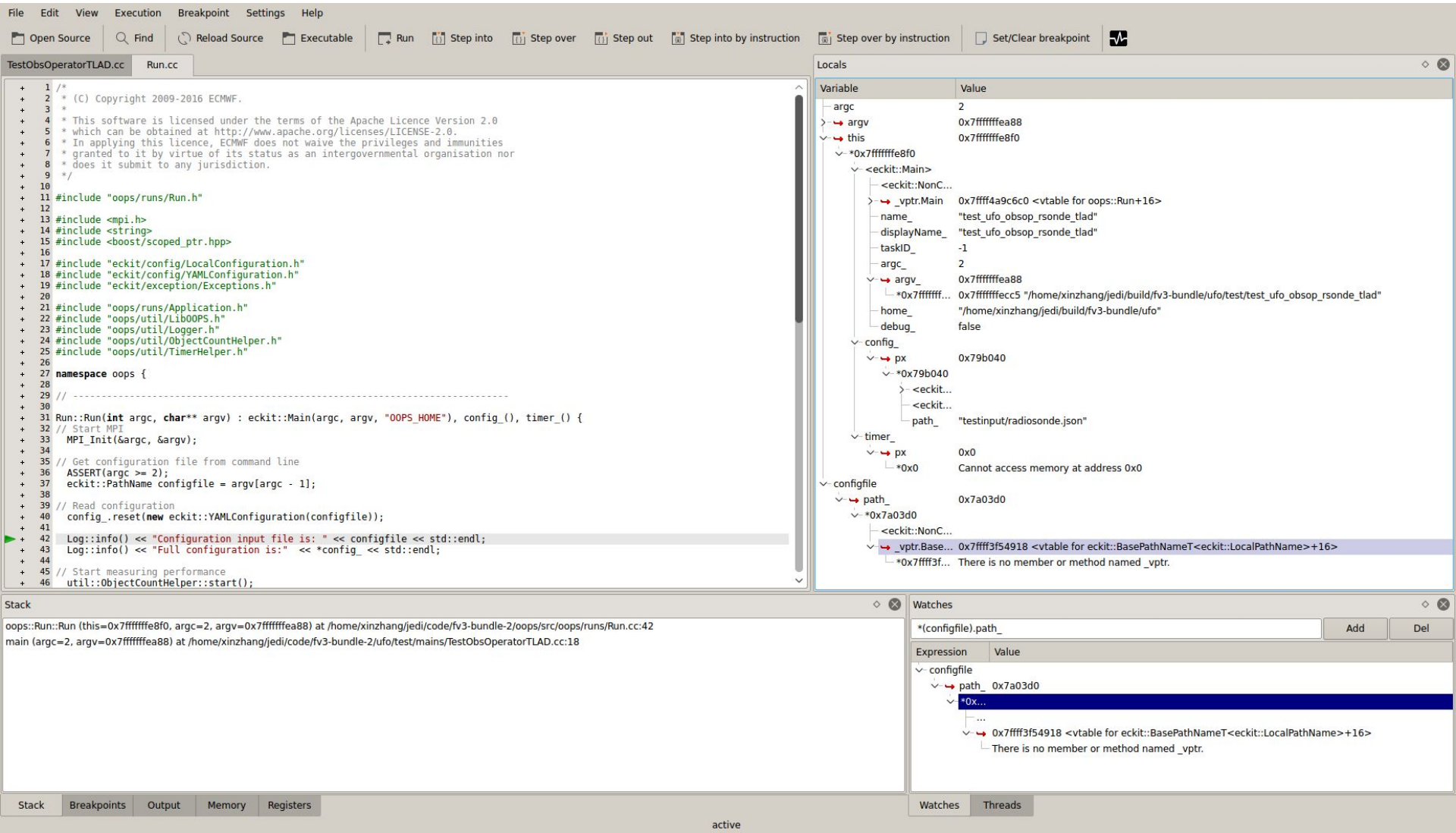
```
Start testing: Jun 03 03:05 UTC
----------------------------------------------------------
233/233 Testing: test_ufo_obsop_rsonde_tlad
233/233 Test: test_ufo_obsop_rsonde_tlad
Command: "/home/xinzhang/jedi/build/ufo-bundle/ufo/test/test_ufo_obsop_rsonde_tlad" "--" "testinput/radiosonde.json"
Directory: /home/xinzhang/jedi/build/ufo-bundle/ufo/test
"test_ufo_obsop_rsonde_tlad" start time: Jun 03 03:05 UTC
Output:
----------------------------------------------------------
Configuration input file is: testinput/radiosonde.json
Full configuration is:YAMLConfiguration[path=testinput/radiosonde.json, root={test_framework_runtime_config => --log_level=test_suite ,
window_begin => 2010-01-01T00:00:00Z , window_end => 2010-01-02T00:00:00Z , LinearObsOpTest => {testiterTL => 10 , toleranceTL =>
1.0e-8 , toleranceAD => 1.0e-12} , Observations => {ObsTypes => ({ObsType => Radiosonde , ObsData => {ObsDataIn => {obsfile =>
Data/diag_t_01_wprofiles.nc4}} , GeoVaLs => {norm => 8471.883687854357 , random => 0 , filename => Data/diag_t_01_wprofiles.nc4} ,
rmsequiv => 254.50117867804789 , tolerance => 1.0e-8 , ObsBias => {}})}}]
……
```

# Debug with kdbg in Singularity

- Build with debug option
  - ecbuild - -build=**debug** ../../fv3-bundle

- Run with kdbg
  - cd /home/xinzhang/jedi/build/ufo-bundle/ufo/test
  - kdbg test_ufo_obsop_rsonde_tlad -a testinput/radiosonde.json
  - Kdbg does not support parallel applications ( or we are not aware of )

# Debug with kdbg in Singularity



# Adding a new test

1. Create a File for your Test Application
2. Define A Test Fixture
3. Define Your Unit Tests
4. Register your Unit Tests with Boost
5. Create an Executable
6. Create a Configuration File
7. Register all files with CMake and CTest

https://jointcenterforsatellitedataassimilation-jedi-docs.readthedocs-hosted.com/en/latest/developer/building_and_testing/adding_a_test.html

# Adding a new test

1. Create a File for your Test Application
   - The goal of the JEDI Testing Framework is to have the test directory mirror the source directory such that each of the main C++ classes defined in the source directory has a corresponding test.
   - if have a new class
     - src/mydir/MyClass.h
   - create a file that will contain the test application
     - test/mydir/MyClass.h .

# Adding a new test

2. Define A Test Fixture
   - test fixtures are generally used to create objects as directed by the relevant sections of the configuration file, for use with the unit tests
   - it would be advisable to begin by defining a test::MyClassFixture class in test/mydir/MyClass.h to facilitate the creation of useful objects as specified in the configuration file. For many more examples see the various files in oops/src/test/interface.

# Adding a new test


## 3. Define Your Unit Tests

– define the unit tests themselves as functions within test/mydir/MyClass.h

```
template <typename MODEL> void testStateConstructors() {
  typedef StateFixture<MODEL>   Test ;
  typedef oops::State<MODEL>    State_;

  const double norm = Test ::test().getDouble("norm-file");
  const double tol = Test ::test().getDouble("tolerance");
  const util::DateTime vt(Test_::test().getString("date"));

  // Test main constructor
  const eckit::LocalConfiguration conf(Test ::test(), "StateFile");
  boost::scoped_ptr<State_> xx1(new State_(Test_::resol(), conf));

  BOOST CHECK(xx1.get());
  const double norm1 = xx1->norm();
  BOOST CHECK CLOSE(norm1, norm, tol);
  BOOST_CHECK_EQUAL(xx1->validTime(), vt);

  [...]
```

# Adding a new test

4. Register your Unit Tests with Boost
   - In order for Boost to run your tests, you have to generate a Boost test suite. This is achieved by means of the register_tests() method of test::MyClass

```
template <typename MODEL> class Increment : public oops::Test {
 public:
  Increment() {}
  virtual ~Increment() {}
 private:
  std::string testid() const { return "test::Increment<" + MODEL::name() +
">";}

  void register_tests() const {
    boost::unit_test::test_suite * ts =
BOOST_TEST_SUITE("interface/Increment");

    ts->add(BOOST_TEST_CASE( &testIncrementConstructor<MODEL>));
    ts->add(BOOST_TEST_CASE( &testIncrementCopyConstructor<MODEL>));
    ts->add(BOOST_TEST_CASE( &testIncrementTriangle<MODEL>));
    ts->add(BOOST_TEST_CASE( &testIncrementOpPlusEq<MODEL>));
    ts->add(BOOST_TEST_CASE( &testIncrementDotProduct<MODEL>));
    ts->add(BOOST_TEST_CASE( &testIncrementAxpy<MODEL>));

    boost::unit_test::framework::master_test_suite() .add(ts);
  }
};
```

# Adding a new test

5. Create an Executable
   - Executables for each test are generally located in the test/executables directory of each JEDI repository, though sometimes this directory is called test/mains.
   - Create an oops::Run object
   - Create an oops::Application object (in our example, this would be test::MyClass)
   - Pass the Application object to the execute() method of the Run object

```
#include "oops/runs/Run.h"
#include "../mydir/MyClass.h"

int main(int argc,  char ** argv) {
  oops::Run run(argc, argv);
  test::MyClass tests;
  run.execute(tests);
  return 0;
};
```

# Adding a new test

6. Create a Configuration File
   - Along with the executable, the configuration file is the way to tell JEDI what you want it to do.
   - the proper place to put it is in the test/testinput or test/mydir/testinput directory.
   - call our configuration file test/testinput/myclass.json.

# Adding a new test

7. Register all files with CMake and CTest
   - In steps 1-6 above we have created or modified three files, namely the source code for our tests, test/mydir/MyClass.h, the executable test/executables/TestMyClass.cc, and the configuration file test/testinput/myclass.json
   - editing the file test/CMakeLists.txt.
   - add your input file, test/testinput/myclass.json to json/yaml list
   - register your test with CTest. We can do this with a call to ecbuild_add_test() in the test/CMakeLists.txt file.

```
ecbuild_add_test( TARGET  test_myrepo_myclass
                  BOOST
                  SOURCES executables/TestMyClass.cc
                  ARGS    "../testinput/myclass.json"
                  LIBS    myrepo )
```