Open CASCADE Technology
6.7.0

Automated Testing System

December 16, 2013

Contents

1 Introduction

This document provides overview and practical guidelines for work with OCCT automatic testing system. Reading this section *Introduction* should be sufficient for OCCT developers to use the test system to control non-regression of the modifications they implement in OCCT. Other sections provide more in-depth description of the test system, required for modifying the tests and adding new test cases.

1.1 Basic Information

OCCT automatic testing system is organized around DRAW Test Harness DRAW Test Harness, a console application based on Tcl (a scripting language) interpreter extended by OCCT-related commands.

Standard OCCT tests are included with OCCT sources and are located in subdirectory *tests* of the OCCT root folder. Other test folders can be included in the scope of the test system, e.g. for testing applications based on OCCT.

Logically the tests are organized in three levels:

- Group: group of related test grids, usually relating to some part of OCCT functionality (e.g. blend)
- Grid: set of test cases within a group, usually aimed at testing some particular aspect or mode of execution of the relevant functionality (e.g. buildevol)
- Test case: script implementing individual test (e.g. K4)

Some tests involve data files (typically CAD models) which are located separately and are not included with OCCT code. The archive with publicly available test data files should be downloaded and installed independently on OCCT code from dev.opencascade.org.

1.2 Intended Use of Automatic Tests

Each modification made in OCCT code must be checked for non-regression by running the whole set of tests. The developer who does the modification is responsible for running and ensuring non-regression on the tests that are available to him. Note that many tests are based on data files that are confidential and thus available only at OPEN CASCADE. Thus official certification testing of the changes before integration to master branch of official OCCT Git repository (and finally to the official release) is performed by OPEN CASCADE in any case.

Each new non-trivial modification (improvement, bug fix, new feature) in OCCT should be accompanied by a relevant test case suitable for verifying that modification. This test case is to be added by developer who provides the modification. If a modification affects result of some test case(s), either the modification should be corrected (if it causes regression) or affected test cases should be updated to account for the modification.

The modifications made in the OCCT code and related test scripts should be included in the same integration to master branch.

1.3 Quick Start

1.3.1 Setup

Before running tests, make sure to define environment variable CSF_TestDataPath pointing to the directory containing test data files. (The publicly available data files can be downloaded from http://dev.opencascade.org separately from OCCT code.) The recommended way for that is adding a file *DrawInitAppli* in the directory which is current at the moment of starting DRAWEXE (normally it is $CASROOT). This file is evaluated automatically at the DRAW start. Example:

```
set env(CSF_TestDataPath) d:/occt/tests_data
return ;# this is to avoid an echo of the last command above in cout
```

All tests are run from DRAW command prompt, thus first run draw.tcl or draw.sh to start DRAW.

1.3.2 Running Tests

To run all tests, type command *testgrid* followed by path to the new directory where results will be saved. It is recommended that this directory should be new or empty; use option –overwrite to allow writing logs in existing non-empty directory.

Example:

```
Draw[]> testgrid d:/occt/results-2012-02-27
```

If empty string is given as log directory name, the name will be generated automatically using current date and time, prefixed by *results_*. Example:

```
Draw[]> testgrid {}
```

For running only some group or a grid of tests, give additional arguments indicating group and (if needed) grid. Example:

```
Draw[]> testgrid d:/occt/results-2012-02-28 blend simple
```

As the tests progress, the result of each test case is reported. At the end of the log summary of test cases is output, including list of detected regressions and improvements, if any. Example:

```
Tests summary

CASE 3rdparty export A1: OK
...
CASE pipe standard B1: BAD (known problem)
CASE pipe standard C1: OK
No regressions
Total cases: 208 BAD, 31 SKIPPED, 3 IMPROVEMENT, 1791 OK
Elapsed time: 1 Hours 14 Minutes 33.7384512019 Seconds
Detailed logs are saved in D:/occt/results_2012-06-04T0919
```

The tests are considered as non-regressive if only OK, BAD (i.e. known problem), and SKIPPED (i.e. not executed, e.g. because of lack of data file) statuses are reported. See `Grid's` *`cases.list`* `file` chapter for details.

The detailed logs of running tests are saved in the specified directory and its sub-directories. Cumulative HTML report summary.html provides links to reports on each test case. An additional report TESTS-summary.xml is output in JUnit-style XML format that can be used for integration with Jenkins or other continuous integration system. Type *help testgrid* in DRAW prompt to get help on additional options supported by testgrid command.

```
Draw[3]> help testgrid
testgrid: Run all tests, or specified group, or one grid
    Use: testgrid logdir [group [grid]] [options...]
    Allowed options are:
    -verbose {0-2}: verbose level, 0 by default, can be set to 1 or 2
    -parallel N: run in parallel with up to N processes (default 0)
    -refresh N: save summary logs every N seconds (default 60)
    -overwrite: force writing logs in existing non-empty directory
```

1.3.3 Running Single Test

To run single test, type command *test*' followed by names of group, grid, and test case.

Example:

```
Draw[1]> test blend simple A1
CASE blend simple A1: OK
Draw[2]>
```

Note that normally intermediate output of the script is not shown. To see intermediate commands and their output, type command *decho on* before running the test case. (Type 'decho off' to disable echoing when not needed.) The detailed log of the test can also be obtained after the test execution by command *dlog get*.

2 Organization of Test Scripts

2.1 General Layout

Standard OCCT tests are located in subdirectory tests of the OCCT root folder ($CASROOT). Additional test folders can be added to the test system by defining environment variable CSF_TestScriptsPath. This should be list of paths separated by semicolons (∗;∗) on Windows or colons (∗:∗) on Linux or Mac. Upon DRAW launch, path to tests sub-folder of OCCT is added at the end of this variable automatically. Each test folder is expected to contain:

- Optional file parse.rules defining patterns for interpretation of test results, common for all groups in this folder
- One or several test group directories.

Each group directory contains:

- File grids.list that identifies this test group and defines list of test grids in it.
- Test grids (sub-directories), each containing set of scripts for test cases, and optional files cases.list, parse.-rules, begin, and end.
- Optional sub-directory data
- Optional file parse.rules
- Optional files begin and end

By convention, names of test groups, grids, and cases should contain no spaces and be lowercase. Names begin, end, data, parse.rules, grids.list, cases.list are reserved. General layout of test scripts is shown on Figure 1.

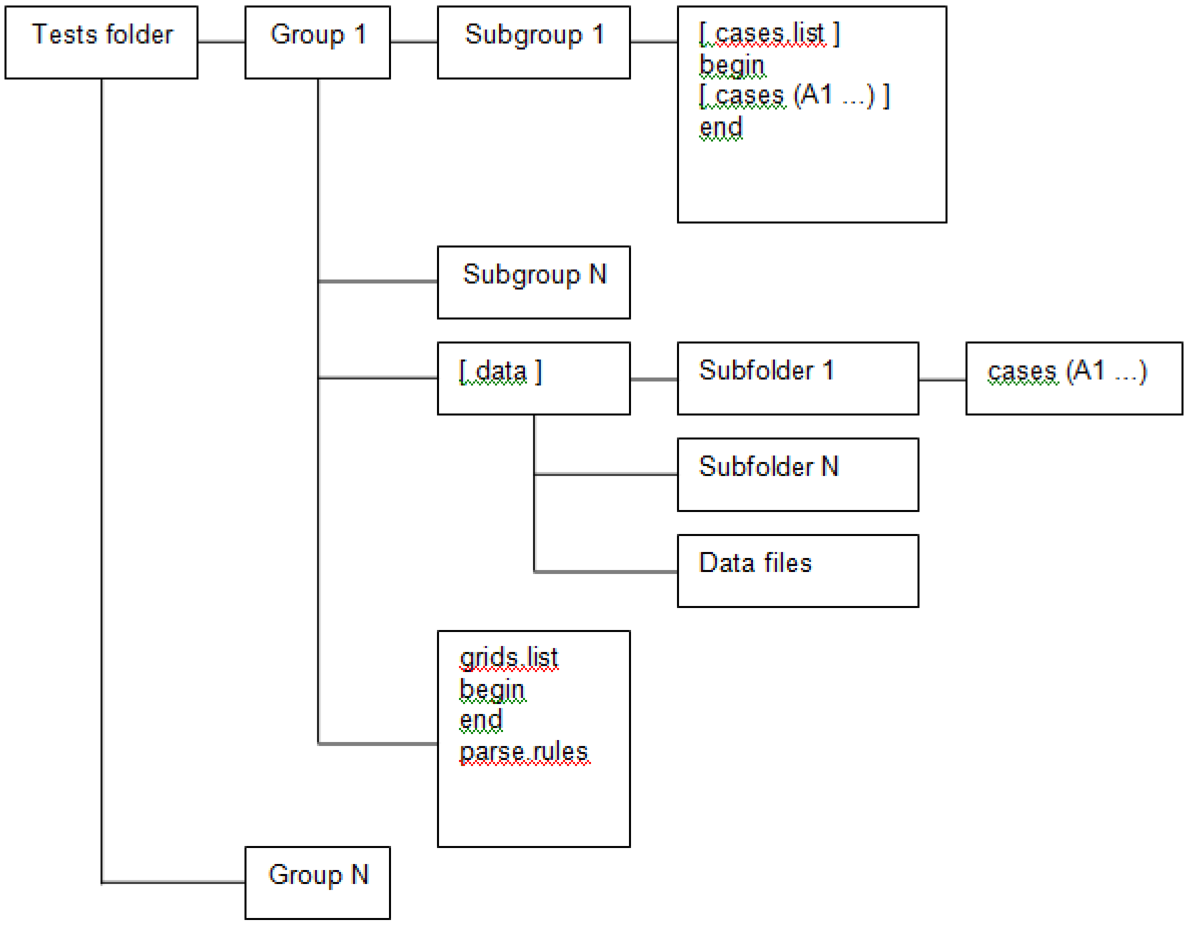

Figure 1. Layout of tests folder

2.2 Test Groups

2.2.1 Group Names

Test folder usually contains several directories representing test groups (Group 1, Group N). Each directory contains test grids for certain OCCT functionality. The name of directory corresponds to this functionality. Example:

```
  caf
  mesh
  offset
```

2.2.2 Group's ∗grids.list∗ File

The test group must contain file *grids.list* file which defines ordered list of grids in this group in the following format:

```
001 gridname1
002 gridname2
...
NNN gridnameN
```

Example:

```
001 basic
002 advanced
```

2.2.3 Group's ∗begin∗ File

The file *begin* is a Tcl script. It is executed before every test in current group. Usually it loads necessary Draw commands, sets common parameters and defines additional Tcl functions used in test scripts. Example:

```
pload TOPTEST ;# load topological command
set cpulimit 300 ;# set maximum time allowed for script execution
```

2.2.4 Group's ∗end∗ File

The file end is a TCL script. It is executed after every test in current group. Usually it checks the results of script work, makes a snap-shot of the viewer and writes *TEST COMPLETED* to the output. Note: *TEST COMPLETED* string should be presented in output in order to signal that test is finished without crash. See `Creation And Modification Of Tests` chapter for more information. Example:

```
if { [isdraw result]} {
    checkshape result
} else {
    puts ⋆Error: The result shape can not be built⋆
}
puts ⋆TEST COMPLETED⋆
```

2.2.5 Group's ∗parse.rules∗ File

The test group may contain *parse.rules* file. This file defines patterns used for analysis of the test execution log and deciding the status of the test run. Each line in the file should specify a status (single word), followed by a regular expression delimited by slashes (∗/∗) that will be matched against lines in the test output log to check if it corresponds to this status. The regular expressions support subset of the Perl re syntax. The rest of the line can contain a comment message which will be added to the test report when this status is detected. Example:

```
FAILED /\\b[Ee]xception\\b/ exception
FAILED /\\bError\\b/ error
SKIPPED /Cannot open file for reading/ data file is missing
SKIPPED /Could not read file .*, abandon/ data file is missing
```

Lines starting with a ∗#∗ character and blank lines are ignored to allow comments and spacing. See `Interpretation of test results` chapter for details.

If a line matches several rules, the first one applies. Rules defined in the grid are checked first, then rules in group, then rules in the test root directory. This allows defining some rules on the grid level with status IGNORE to ignore messages that would otherwise be treated as errors due to the group level rules. Example:

```
FAILED /\\bFaulty\\b/ bad shape
IGNORE /^Error [23]d = [\d.-]+/ debug output of blend command
IGNORE /^Tcl Exception: tolerance ang : [\d.-]+/ blend failure
```

2.3 Test Grids

2.3.1 Grid Names

Group folder can have several sub-directories (Grid 1... Grid N) defining test grids. Each test grid directory contains a set of related test cases. The name of directory should correspond to its contents.

Example:

caf basic bugs presentation

Where **caf** is the name of test group and *basic*, *bugs*, *presentation*, etc are the names of grids.

2.3.2 Grid's ∗begin∗ File

The file *begin* is a TCL script. It is executed before every test in current grid. Usually it sets variables specific for the current grid. Example:

```
set command bopfuse ;# command tested in this grid
```

2.3.3 Grid's ∗end∗ File

The file *end* is a TCL script. It is executed after every test in current grid. Usually it executes specific sequence of commands common for all tests in the grid. Example:

```
vdump $logdir/${casename}.gif ;# makes a snap-shot of AIS viewer
```

2.3.4 Grid's ∗cases.list∗ File

The grid directory can contain an optional file cases.list defining alternative location of the test cases. This file should contain singe line defining the relative path to collection of test cases.

Example:

../data/simple

This option is used for creation of several grids of tests with the same data files and operations but performed with differing parameters. The common scripts are usually located place in common subdirectory of the test group (data/simple as in example). If cases.list file exists then grid directory should not contain any test cases. The specific parameters and pre- and post-processing commands for the tests execution in this grid should be defined in the begin and end files.

2.4 Test Cases

The test case is TCL script which performs some operations using DRAW commands and produces meaningful messages that can be used to check the result for being valid. Example:

```
pcylinder c1 10 20 ;# create first cylinder
pcylinder c2 5 20 ;# create second cylinder
ttranslate c2 5 0 10 ;# translate second cylinder to x,y,z
bsection result c1 c2 ;# create a section of two cylinders
checksection result ;# will output error message if result is bad
```

The test case can have any name (except reserved names begin, end, data, cases.list, parse.rules). For systematic grids it is usually a capital English letter followed by a number.

Example:

```
A1
A2
B1
B2
```

Such naming facilitates compact representation of results of tests execution in tabular format within HTML reports.

2.5 Directory ∗data∗

The test group may contain subdirectory data. Usually it contains data files used in tests (BREP, IGES, STEP, etc.) and / or test scripts shared by different test grids (in subdirectories, see `Grid's` *`cases.list`* `file` chapter).

3 Creation And Modification Of Tests

This section describes how to add new tests and update existing ones.

3.1 Choosing Group, Grid, and Test Case Name

The new tests are usually added in context of processing some bugs. Such tests in general should be added to group bugs, in the grid corresponding to the affected OCCT functionality. New grids can be added as necessary to contain tests on functionality not yet covered by existing test grids. The test case name in the bugs group should be prefixed by ID of the corresponding issue in Mantis (without leading zeroes). It is recommended to add a suffix providing a hint on the situation being tested. If more than one test is added for a bug, they should be distinguished by suffixes; either meaningful or just ordinal numbers.

Example:

```
12345_coaxial
12345_orthogonal_1
12345_orthogonal_2
```

In the case if new test corresponds to functionality for which specific group of tests exists (e.g. group mesh for BRepMesh issues), this test can be added (or moved later by OCC team) to this group.

3.2 Adding Data Files Required for a Test

It is advisable that tests scripts should be made self-contained whenever possible, so as to be usable in environments where data files are not available. For that simple geometric objects and shapes can be created using DRAW commands in the test script itself. If test requires some data file, it should be put to subdirectory data of the test grid. Note that when test is integrated to master branch, OCC team can move data file to data files repository, so as to keep OCCT sources repository clean from big data files. When preparing a test script, try to minimize size of involved data model. For instance, if problem detected on a big shape can be reproduced on a single face extracted from that shape, use only this face in the test.

3.3 Implementation of the Script

Test should run commands necessary to perform the operations being tested, in a clean DRAW session. This includes loading necessary functionality by *pload* command, if this is not done by *begin* script. The messages produced by commands in standard output should include identifiable messages on the discovered problems if any. Usually the script represents a set of commands that a person would run interactively to perform the operation and see its results, with additional comments to explain what happens. Example:

```
# Simple test of fusing box and sphere
box b 10 10 10
sphere s 5
bfuse result b s
checkshape result
```

Make sure that file parse.rules in the grid or group directory contains regular expression to catch possible messages indicating failure of the test. For instance, for catching errors reported by *checkshape* command relevant grids define a rule to recognize its report by the word *Faulty*: FAILED /\bFaulty\b/ bad shape For the messages generated in the script the most natural way is to use the word *Error* in the message. Example:

```
set expected_length 11
if { [expr $actual_length - $expected_length] > 0.001} {
    puts *Error: The length of the edge should be $expected_length*
}
```

At the end, the test script should output *TEST COMPLETED* string to mark successful completion of the script. This is often done by the end script in the grid. When test script requires data file, use Tcl procedure *locate_data_file* to get path to the data file, rather than explicit path. This will allow easy move of the data file from OCCT repository to the data files repository without a need to update test script. Example:

```
stepread [locate_data_file CAROSKI_COUPELLE.step] a *
```

When test needs to produce some snapshots or other artifacts, use Tcl variable logdir as location where such files should be put. Command *testgrid* sets this variable to the subdirectory of the results folder corresponding to the grid. Command *test* sets it to $CASROOT/tmp unless it is already defined. Use Tcl variable casename to prefix all the files produced by the test. This variable is set to the name of the test case. Example:

```
xwd $logdir/${casename}.gif
vdisplay result; vfit
vdump $logdir/${casename}-axo.gif
vfront; vfit
vdump $logdir/${casename}-front.gif
```

could produce:

```
A1.gif
A1-axo.gif
A1-front.gif
```

3.4 Interpretation of Test Results

The result of the test is evaluated by checking its output against patterns defined in the files parse.rules of the grid and group. The OCCT test system recognizes five statuses of the test execution:

- SKIPPED: reported if line matching SKIPPED pattern is found (prior to any FAILED pattern). This indicates that the test cannot be run in the current environment; most typical case is absence of the required data file.
- FAILED: reported if some line matching pattern with status FAILED is found (unless it is masked by preceding IGNORE pattern or a TODO statement, see below), or if message TEST COMPLETED is not found at the end. This indicates that test produces bad or unexpected result, and usually highlights a regression.
- BAD: reported if test script output contains one or several TODO statements and corresponding number of matching lines in the log. This indicates a known problem (see 3.5). The lines matching TODO statements are not checked against other patterns and thus will not cause a FAILED status.
- IMPROVEMENT: reported if test script output contains TODO statement for which no corresponding line is found. This is possible indication of improvement (known problem disappeared).
- OK: If none of the above statuses have been assigned. This means test passed without problems.

Other statuses can be specified in the parse.rules files, these will be classified as FAILED. Before integration of the change to OCCT repository, all tests should return either OK or BAD status. The new test created for unsolved problem should return BAD. The new test created for a fixed problem should return FAILED without the fix, and OK with the fix.

3.5 Marking BAD Cases

If the test produces invalid result at a certain moment then the corresponding bug should be created in the OCCT issue tracker `http://tracker.dev.opencascade.org,` and the problem should be marked as TODO in the test script. The following statement should be added to such test script:

```
puts *TODO BugNumber ListOfPlatforms: RegularExpression*
```

where:

- BugNumber is an ID of the bug in the tracker. For example: #12345
- ListOfPlatform is a list of platforms at which the bug is reproduced (e.g. Mandriva2008, Windows or All).

Note: the platform name is custom for the OCCT test system; it can be consulted as value of environment variable os_type defined in DRAW.

Example:

```
    Draw[2]> puts $env(os_type)
    windows
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~{.tcl}

  * RegularExpression is a regular expression which should be matched against the line indicating the
    problem in the script output.
Example:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~{.tcl}
    puts *TODO #22622 Mandriva2008: Abort .* an exception was raised*
```

Parser checks the output of the test and if an output line matches the RegularExpression then it will be assigned a BAD status instead of FAILED. For each output line matching to an error expression a separate TODO line must be added to mark the test as BAD. If not all the TODO statements are found in the test log, the test will be considered as possible improvement. To mark the test as BAD for an incomplete case (when final TEST COMPLETE message is missing) the expression *TEST INCOMPLETE* should be used instead of regular expression.

Example:

```
puts *TODO OCC22817 All: exception.+There are no suitable edges*
puts *TODO OCC22817 All: \\*\\* Exception \\*\\**
puts *TODO OCC22817 All: TEST INCOMPLETE*
```

4 Extended Use

4.1 Running Tests on Older Versions of OCCT

Sometimes it might be necessary to run tests on previous versions of OCCT (up to to 6.5.3) that do not include this test system. This can be done by adding DRAW configuration file DrawAppliInit in the directory which is current by the moment of DRAW startup, to load test commands and define necessary environment. Example (assume that d:/occt contains up-to-date version of OCCT sources with tests, and test data archive is unpacked to d:/test-data):

```
set env(CASROOT) d:/occt
set env(CSF_TestScriptsPath) $env(CASROOT)/tests
source $env(CASROOT)/src/DrawResources/TestCommands.tcl
set env(CSF_TestDataPath) d:/test-data
return
```

Note that on older versions of OCCT the tests are run in compatibility mode and not all output of the test command can be captured; this can lead to absence of some error messages (can be reported as improvement).

4.2 Adding Custom Tests

You can extend the test system by adding your own tests. For that it is necessary to add paths to the directory where these tests are located, and additional data directory(ies), to the environment variables CSF_TestScriptsPath and CSF_TestDataPath. The recommended way for doing this is using DRAW configuration file DrawAppliInit located in the directory which is current by the moment of DRAW startup.

Use Tcl command ∗_path_separator∗ to insert platform-dependent separator to the path list. Example:

```
set env(CSF_TestScriptsPath) \
  $env(TestScriptsPath)[_path_separator]d:/MyOCCTProject/tests
set env(CSF_TestDataPath) \
  d:/occt/test-data[_path_separator]d:/MyOCCTProject/tests
return ;# this is to avoid an echo of the last command above in cout
```