# Open CASCADE Technology

# Automated testing system

# User's Guide

## CONTENTS

# 1. INTRODUCTION

This document provides OCCT developers and contributors with an overview and practical guidelines for work with OCCT automatic testing system.

Reading this section "Introduction" should be sufficient for developers to use the test system to control non-regression of the modifications they implement in OCCT. Other sections provide a 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 [1], 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 to the test system, e.g. for testing applications based on OCCT.

The tests are organized in three levels:

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

See 5.1 for the current list of available test groups and grids.

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 (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 for the tests 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 the 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 the developer who provides the modification.

If a modification affects the result of an existing test case, either the modification should be corrected (if it causes regression) or the 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 the 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. (Publicly available data files can be downloaded from http://dev.opencascade.org separately from OCCT code.)

For this it is recommended to add a file **DrawAppliInit** in the directory which is current at the moment of starting DRAWEXE (normally it is OCCT root directory, $CASROOT). This file is evaluated automatically at the DRAW start.

Example (Windows):

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

Note that variable CSF_TestDataPath is set to default value at DRAW start, pointing to folder $CASROOT/data.

In this example, subdirectory d:/occt/test-data is added to this path. Similar code could be used on Linux and Mac OS X except that on non-Windows platforms colon ‘:’ should be used as path separator instead of semicolon ‘;’.

All tests are run from DRAW command prompt (run **draw.bat** or **draw.sh** to start it).

### 1.3.2. Running tests

To run all tests, type command ‘testgrid’.

Example:

```
Draw[]> testgrid
```

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

Example:

```
Draw[]> testgrid blend simple
```

As the tests progress, the result of each test case is reported. At the end of the log a summary of test cases is output, including the 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_master_2012-06-04T0919
```

The tests are considered as non-regressive if only OK, BAD (i.e. known problem), and SKIPPED (i.e. not executed, typically because of lack of a data file) statuses are reported. See 3.5 for details.

The results and detailed logs of the tests are saved by default to a subdirectory of the current folder, whose name is generated automatically using the current date and time, prefixed by word "results_" and Git branch name (if Git is available and current sources are managed by Git).

If necessary, a non-default output directory can be specified using option **–outdir** followed by a path to the directory. This directory should be new or empty; use option **–overwrite** to allow writing results in existing non-empty directory.

Example:

```
Draw[]> testgrid –outdir d:/occt/last_results -overwrite
```

In the output directory, a cumulative HTML report **summary.html** provides links to reports on each test case. An additional report in JUnit-style XML format can be output for use in Jenkins or other continuous integration system.

Type 'help testgrid' in DRAW prompt to get help on options supported by testgrid command.

```
Draw[1]> help testgrid
testgrid: Run all tests, or specified group, or one grid
    Use: testgrid [group [grid]] [options...]
    Allowed options are:
    -parallel N: run N parallel processes (default is number of CPUs, 0 to disable)
    -refresh N: save summary logs every N seconds (default 60, minimal 1, 0 to disable)
    -outdir dirname: set log directory (should be empty or non-existing)
    -overwrite: force writing logs in existing non-empty directory
    -xml filename: write XML report for Jenkins (in JUnit-like format)
```

### 1.3.3. Running single test

To run a single test, type command ‘test’ followed by the names of group, grid, and test case.

Example:

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

Note that normally an intermediate output of the script is not shown. The detailed log of the test can be obtained after the test execution by running command "dlog get".

To see intermediate commands and their output during the test execution, add one more argument "-echo" at the end of the command line. Note that with this option the log is not collected and summary is not produced.

The tests are executed in the global scope, thus all artifacts produced by test case (shapes, Tcl variables, etc.) remain in the DRAW session after the test is completed, and can be inspected interactively. Some tests may be sensitive to presence of such artifacts, thus it is recommended to start new DRAW session for executing a test case to ensure it runs as expected.

### 1.3.4. Creating a new test

See section 3 for detailed description of the rules for creation of the new tests. The following short description covers the most typical situations:

1. Use prefix “bug” followed by Mantis issue ID and, if necessary, additional suffixes, for naming the test script and DRAW commands specific for this test case.
2. If the test requires C++ code, add it as new DRAW command(s) in one of files in QABugs package.
3. Add script(s) for the test case in grid (subfolder) corresponding to the relevant OCCT module of the group **bugs** ($CASROOT/tests/bugs). See 5.2 for the correspondence map.
4. In the test script:
    a. Load all necessary DRAW modules by command **pload**.
    b. Use command **locate_data_file** to get a path to data files used by test script. (Make sure to have this command not inside catch statement if it is used.)
    c. Use DRAW commands to reproduce the situation being tested.
    d. If test case is added to describe existing problem and the fix is not available, add TODO message for each error to mark it as known problem. The TODO statements must be specific so as to match the actually generated messages but not other similar errors.
    e. Make sure that in case of failure the test produces message containing word “Error” or other recognized by test system as error (see files **parse.rules**).
5. If the test case uses data file(s) not yet present in the test database, these can be put to subfolder **data** of the test grid, and integrated to Git along with the test case.
6. Check that the test case runs as expected (test for fix: OK with the fix, FAILED without the fix; test for existing problem: BAD), and integrate to Git branch created for the issue.

Example:

Added files:

```
> git status –short
A tests/bugs/heal/data/OCC210a.brep
A tests/bugs/heal/data/OCC210a.brep
A tests/bugs/heal/bug210_1
A tests/bugs/heal/bug210_2
```

Test script

```
puts "OCC210 (case 1): Improve FixShape for touching wires"

restore [locate_data_file OCC210a.brep] a

fixshape result a 0.01 0.01
checkshape result
```

# 2. ORGANIZATION OF THE 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 a list of paths separated by semicolons (“;”) on Windows or colons (“:”) on Linux or Mac. Upon DRAW launch, the 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 the list of test grids in it.
- Test grids (sub-directories), each containing a 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, the 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.

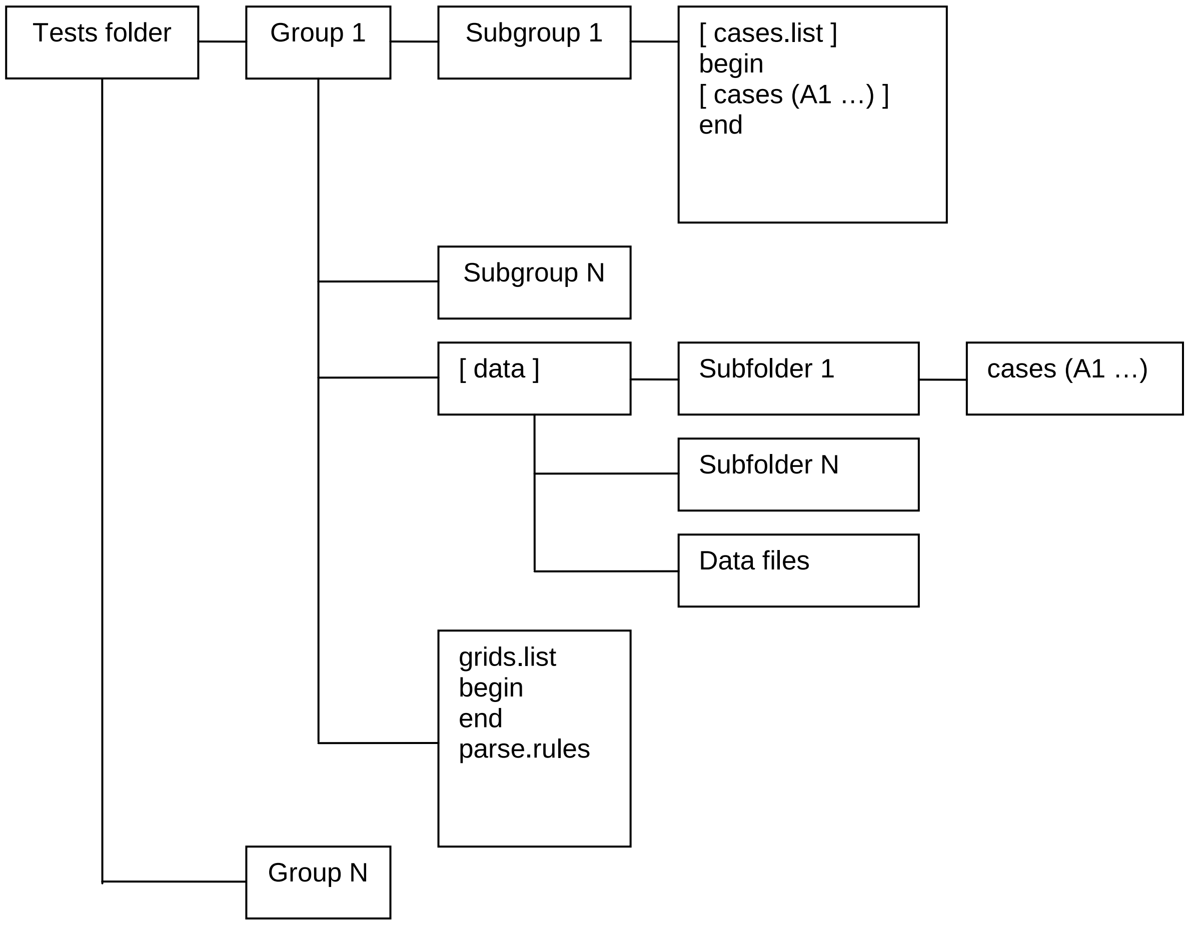

**Figure 1. Layout of tests folder**

## 2.2. Test groups

### 2.2.1. Group names

The names of directories of test groups containing systematic test grids correspond to the functionality tested by each group.

Example:

```
caf
mesh
offset
```

A test group **bugs** is used to collect the tests coming from bug reports. Group **demo** collects tests of the test system, DRAW, samples, etc.

### 2.2.2. File "grids.list"

The test group must contain a file "**grids.list**" which defines an 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. File “begin”

The file **begin** is a Tcl script. It is executed before every test in the 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 commands
set cpulimit 300 ;# set maximum CPU time allowed for script execution
```

### 2.2.4. File “end”

The file **end** is a TCL script. It is executed after every test in the 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 present in the output in order to signal that the test is finished without crash. See section 3 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. File “parse.rules”

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 are based on Tcl re_syntax with exception of '\b' escape symbol which is interpreted as word boundary following Perl **re** syntax (replaced by '\y' for Tcl).

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 3.5 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
```

#### 2.2.6. Directory “data”

The test group may contain subdirectory **data** where test scripts shared by different test grids can be put (see 2.3.4).

## 2.3. Test grids

### 2.3.1. Grid names

A test 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 the directory should correspond to its contents.

Example:

```
caf
   basic
   bugs
   presentation
```

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

### 2.3.2. File “begin”

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

Example:

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

### 2.3.3. File “end”

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

Example:

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

### 2.3.4. File “cases.list”

The grid directory can contain an optional file **cases.list** defining an alternative location of the test cases. This file should contain a single line defining the relative path to the 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 in common subdirectory of the test group (**data/simple** as in example).

If **cases.list** file exists then the 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 files **begin** and **end**.

### 2.3.5. Directory “data”

The test grid may contain subdirectory **data** where data files used in tests (BREP, IGES, STEP, etc.) of this grid can be put.

## 2.4. Test cases

The test case is a TCL script which performs some operations using DRAW commands and produces meaningful messages that can be used to check the validity of the result.

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 for the reserved names **begin**, **end**, **data**, **cases.list, parse.rules**). For test cases corresponding to issues in Mantis bug tracker, the name starts with **'bug'** followed by issue ID. 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 the test execution results in tabular format within HTML reports.

## 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 the frames of processing issues in OCCT Mantis tracker. Such tests in general should be added to group **bugs**, in the grid corresponding to the affected OCCT functionality (see 5.2). New grids can be added as necessary to contain tests of functionality not yet covered by existing test grids.

The test case name in the **bugs** group should start with "bug" followed by the ID of the corresponding issue in Mantis (without leading zeroes). It is recommended to add a suffix providing a hint on the tested situation. If more than one test is added for a bug, they should be distinguished by suffixes; either meaningful or just ordinal numbers.

Example:

```
bug12345_coaxial
bug12345_orthogonal_1
bug12345_orthogonal_2
```

If the new test corresponds to a functionality already covered by the existing systematic test grid (e.g. group **mesh** for BRepMesh issues), this test can be added (or moved later by OCC team) to that grid.

### 3.2. Adding data files required for a test

It is advisable to make self-contained test scripts whenever possible, so as they could be used 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 the test requires a data file, it should be put to subdirectory **data** of the test grid. It is recommended to prefix the data file with the corresponding issue id prefixed by "bug", e.g. "**bug12345_face1.brep**", to avoid possible conflicts with names of existing data files.

Note that when the test is integrated to the master branch, OCC team will move the data file to data files repository, so as to keep OCCT sources repository clean from data files.

When preparing a test script, try to minimize the size of involved data model. For instance, if the problem detected on a big shape can be reproduced on a single face extracted from that shape, use only that face in the test.

## 3.3. Adding new DRAW commands

If the test cannot be implemented using available DRAW commands, consider the following possibilities:

- If existing DRAW command can be extended to enable possibility required for a test in a natural way (e.g. by adding an option to activate a specific mode of the algorithm), this way is recommended. This change should be appropriately documented in a relevant Mantis issue.
- If the new command is needed to access OCCT functionality not exposed to DRAW previously, and this command can be potentially reused (for other tests), it should be added to the package where similar commands are implemented (use "getsource" DRAW command to get the package name). The name and arguments of the new command should be chosen to keep similarity with the existing commands. This change should be documented in a relevant Mantis issue.
- Otherwise the new command implementing the actions needed for this particular test should be added in QABugs package. The command name should be formed by the Mantis issue ID prefixed by "bug", e.g. "**bug12345**".

Note that a DRAW command is expected to return 0 in case of a normal completion, and 1 (Tcl exception) if it is incorrectly used (e.g. a wrong number of input arguments). Thus if the new command needs to report a test error, this should be done by outputting an appropriate error message rather than by returning a non-zero value.

## 3.4. Implementation of the script

The test should run commands necessary to perform the tested operations, in general assuming a clean DRAW session. The required DRAW modules should be loaded by “**pload**” command, if it is not done by “**begin**” script. The messages produced by commands in a 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 a regular expression to catch possible messages indicating the 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 it is recommended to use the word 'Error' in the error 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 a successful completion of the script. This is often done by the **end** script in the grid.

When the test script requires a data file, use Tcl procedure '**locate_data_file**' to get a path to it, instead of putting the path explicitly. This will allow easy move of the data file from OCCT sources repository to the data files repository without the need to update the test script.

Example:

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

When the test needs to produce some snapshots or other artifacts, use Tcl variable *imagedir* as the 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 (unless it is already defined) to system-specific temporary directory.

Use Tcl variable *casename* to prefix all files produced by the test. This variable is set to the name of the test case.

Example:

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

would produce:

```
A1.png
A1-axo.png
A1-front.png
```

Note that OCCT must be built with FreeImage support to be able to produce usable images.

In order to ensure that the test works as expected in different environments, observe the following additional rules:

- Avoid using external commands such as ‘grep’, ‘rm’, etc., as these commands may be absent on another system (e.g. on Windows); use facilities provided by Tcl instead.
- Do not put call to locate_data_file in catch statement – this can prevent correct interpretation of the missing data file by the test system.

## 3.5. 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 a line matching SKIPPED pattern is found (prior to any FAILED pattern). This indicates that the test cannot be run in the current environment; the most typical case is the absence of the required data file.
- **FAILED**: reported if a line matching pattern with status FAILED is found (unless it is masked by the preceding IGNORE pattern or a TODO statement, see below), or if message TEST COMPLETED is not found at the end. This indicates that the test has produced a bad or unexpected result, and usually means a regression.
- **BAD**: reported if the test script output contains one or several TODO statements and the corresponding number of matching lines in the log. This indicates a known problem (see 3.6). The lines matching TODO statements are not checked against other patterns and thus will not cause a FAILED status.
- **IMPROVEMENT**: reported if the test script output contains a TODO statement for which no corresponding line is found. This is a possible indication of improvement (known problem disappeared).
- **OK**: reported if none of the above statuses have been assigned. This means that the test has passed without problems.

Other statuses can be specified in **parse.rules** files, these will be classified as **FAILED**.

For integration of the change to OCCT repository, all tests should return either **OK** or **BAD** status.

The new test created for an unsolved problem should return **BAD**. The new test created for a fixed problem should return **FAILED** without the fix, and **OK** with the fix.

### 3.6. Marking BAD cases

If the test produces an invalid result at a certain moment then corresponding bug should be created in the OCCT issue tracker [2], and the problem should be marked as TODO in the test script.

The following statement should be added to such a test script:

```
puts "TODO BugNumber ListOfPlatforms: RegularExpression"
```

Here:

- **BugNumber** is the bug ID 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 the value of environment variable **os_type** defined in DRAW.

Example:

```
Draw[]> puts $env(os_type)
windows
```

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

Example:

```
puts "TODO #22622 Mandriva2008: Abort .* an exception was raised"
```

The parser checks the test output and if an output line matches the RegularExpression then it will be assigned a BAD status instead of FAILED.

A separate TODO line must be added for each output line matching an error expression to mark the test as BAD. If not all TODO messages 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 the final **"TEST COMPLETE"** message is missing) the expression **"TEST INCOMPLETE"** should be used instead of the 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. ADVANCED USE

### 4.1. Running tests on older versions of OCCT

Sometimes it might be necessary to run tests on older versions of OCCT (prior to 6.5.4) 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 to define necessary environment.

Note: in OCCT 6.5.3, file **DrawAppliInit** already exists in **$CASROOT/src/DrawResources**, new commands should be added to this file instead of a new one in the current directory.

Example (assume that d:/occt contains an up-to-date version of OCCT sources with tests, and the 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) $env(CASROOT)/data;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 either a failure or an 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 one or more additional data directories, 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/data
return ;# this is to avoid an echo of the last command above in cout
```

## 4.3. Parallel execution of tests

For better efficiency, on computers with multiple CPUs the tests can be run in parallel mode. This is default behavior for command testgrid: the tests are executed in parallel processes (their number is equal to the number of CPUs available on the system). In order to change this behavior, use option -parallel followed by the number of processes to be used (1 or 0 to run sequentially).

Note that the parallel execution is only possible if Tcl extension package Thread is installed. It is included in ActiveTcl package, but can be absent in some Linux distributions. If this package is not available, **testgrid** command will output a warning message.

## 4.4. Checking non-regression of performance, memory, and visualization

Some test results are very dependent on the characteristics of the workstation where tests are performed, and thus cannot be checked by comparison with some predefined values. These results can be checked for non-regression (after a change in OCCT code) by comparing them with the results produced by the version without this change. The most typical case is comparing the result obtained in a branch created for integration of a fix (CR***) with the results obtained on the master branch before that change is made.

OCCT test system provides a dedicated command "**testdiff**" for comparing CPU time of execution, memory usage, and images produced by the tests.

Synopsys:

```
testdiff        :
  Compare results of two executions of tests (CPU times, ...)
  Use: testdiff dir1 dir2 [groupname [gridname]] [options...]
  Where dir1 and dir2 are directories containing logs of two test runs.
  Allowed options are:
  -save filename: save resulting log in specified file (default name is
                  $dir1/diff-$dir2.log); HTML log is saved with same name
                  and extension .html
  -status {same|ok|all}: filter cases for comparing by their status:
          same - only cases with same status are compared (default)
          ok   - only cases with OK status in both logs are compared
          all  - results are compared regardless of status
  -verbose level:
          1 - output only differences
          2 - output also list of logs and directories present in one of dirs only
          3 - (default) output also progress messages
```

Example:

```
Draw[]> testdiff results-CR12345-2012-10-10T08:00 results-master-2012-10-09T21:20
```

# 5. APPENDIX

## 5.1. List of existing test groups

### 5.1.1. Group "3rdparty"

This group allows testing the interaction of OCCT and 3rdparty products.

DRAW module: VISUALIZATION.

| Grid | Commands | Functionality |
|---|---|---|
| **export** | vexport | export of images to different formats |
| **fonts** | vtrihedron<br>vcolorscale<br>vdrawtext | display of fonts |

### 5.1.2. Group "blend"

This group allows testing blends (fillets) and related operations.

DRAW module: MODELING.

| Grid | Commands | Functionality |
|---|---|---|
| **simple** | blend | fillets on simple shapes |
| **complex** | blend | fillets on complex shapes, non-trivial geometry |
| **tolblend_simple** | tolblend<br>blend | |
| **buildevol** | buildevol | |
| **tolblend_buildvol** | tolblend<br>buildevol | use of additional command tolblend |
| **bfuseblend** | bfuseblend | |
| **encoderegularity** | encoderegularity | |

### 5.1.3. Group "boolean"

This group allows testing Boolean operations.

DRAW module: MODELING (packages BOPTest, BRepTest).

Grids names are based on name of the command used, with suffixes:

**_2d –** for tests operating with 2d objects (wires, wires and 3d objects and ect.),

**_simple** – for tests operating on simple shapes (boxes, cylinders, toruses, ect.)

**_complex** – for tests dealing with complex shapes

| Grid | Commands | Functionality |
|---|---|---|
| **bcommon_2d** | bcommon | Common operation (old algorithm), 2d |
| **bcommon_complex** | bcommon | Common operation (old algorithm), complex shapes |
| **bcommon_simple** | bcommon | Common operation (old algorithm), simple shapes |
| **bcut_2d** | bcut | Cut operation (old algorithm), 2d |
| **bcut_complex** | bcut | Cut operation (old algorithm), complex shapes |
| **bcut_simple** | bcut | Cut operation (old algorithm), simple shapes |

| Grid | Commands | Functionality |
|---|---|---|
| **bcutblend** | bcutblend | |
| **bfuse_2d** | bfuse | Fuse operation (old algorithm), 2d |
| **bfuse_complex** | bfuse | Fuse operation (old algorithm), complex shapes |
| **bfuse_simple** | bfuse | Fuse operation (old algorithm), simple shapes |
| **bopcommon_2d** | bopcommon | Common operation, 2d |
| **bopcommon_complex** | bopcommon | Common operation, complex shapes |
| **bopcommon_simple** | bopcommon | Common operation, simple shapes |
| **bopcut_2d** | bopcut | Cut operation, 2d |
| **bopcut_complex** | bopcut | Cut operation, complex shapes |
| **bopcut_simple** | bopcut | Cut operation, simple shapes |
| **bopfuse_2d** | bopfuse | Fuse operation, 2d |
| **bopfuse_complex** | bopfuse | Fuse operation, complex shapes |
| **bopfuse_simple** | bopfuse | Fuse operation, simple shapes |
| **bopsection** | bopsection | Section |
| **boptuc_2d** | boptuc | |
| **boptuc_complex** | boptuc | |
| **boptuc_simple** | boptuc | |
| **bsection** | bsection | Section (old algorithm) |

### 5.1.4. Group "bugs"

This group allows testing cases coming from Mantis issues.

The grids are organized following OCCT module and category set for the issue in the Mantis tracker. See 5.2 for details.

### 5.1.5. Group "caf"

This group allows testing OCAF functionality.

DRAW module: OCAFKERNEL.

| Grid | Commands | Functionality |
|---|---|---|
| **basic** | | Basic attributes |
| **bugs** | | Saving and restoring of document |
| **driver** | | OCAF drivers |
| **named_shape** | | TNaming_NamedShape attribute |
| **presentation** | | AISPresentation attributes |
| **tree** | | Tree construction attributes |
| **xlink** | | XLink attributes |

### 5.1.6. Group "chamfer"

This group allows testing chamfer operations.

DRAW module: MODELING.

The test grid name is constructed depending on the type of the tested chamfers. Additional suffix "**_complex**" is used for test cases involving complex geometry (e.g. intersections of edges forming a chamfer); suffix "**_sequence**" is used for grids where chamfers are computed sequentially.

| **Grid** | **Commands** | **Functionality** |
|---|---|---|
| **equal_dist** | | Equal distances from edge |
| **equal_dist_complex** | | Equal distances from edge, complex shapes |
| **equal_dist_sequence** | | Equal distances from edge, sequential operations |
| **dist_dist** | | Two distances from edge |
| **dist_dist_complex** | | Two distances from edge, complex shapes |
| **dist_dist_sequence** | | Two distances from edge, sequential operations |
| **dist_angle** | | Distance from edge and given angle |
| **dist_angle_complex** | | Distance from edge and given angle |
| **dist_angle_sequence** | | Distance from edge and given angle |

### 5.1.7. Group "de"

This group allows testing Data Exchange components of OCCT (IGES and STEP translators).

| **Grid** | **Commands** | **Functionality** |
|---|---|---|
| **iges** | ReadIges, WriteIges | IGES import / export |
| **step** | ReadStep, WriteStep | STEP import / export |

### 5.1.8. Group "demo"

This group allows demonstrating how testing cases are created, and testing DRAW commands and the test system as a whole.

| **Grid** | **Commands** | **Functionality** |
|---|---|---|
| **draw** | getsource<br>restore | Basic DRAW commands |
| **testsystem** | **-** | Testing system |
| **samples** | **-** | OCCT samples |

### 5.1.9. Group "draft"

This group allows testing draft operations.

DRAW module: MODELING.

| **Grid** | **Commands** | **Functionality** |
|---|---|---|
| **Angle** | depouille | Drafts with angle (inclined walls) |

### 5.1.10. Group "feat"

This group allows testing creation of features on a shape.

DRAW module: MODELING (package BRepTest).

| **Grid** | **Commands** | **Functionality** |
|---|---|---|
| **featdprism** | | |
| **featlf** | | |
| **featprism** | | |

| Grid | Commands | Functionality |
|---|---|---|
| **featrevol** | | |
| **featrf** | | |

### 5.1.11. Group “geometry”

This group is for testing creation simple geometry objects and its modifications..

DRAW module: MODELING (packages GeometryTest, GeomliteTest)

| Grid | Commands | Functionality |
|---|---|---|
| **2dbeziecurve** | 2dbeziecurve | Creation 2dbeziecurve with following geometrical modifications |
| **2dbsplinecurve** | 2dbsplinecurve | Creation 2dbsplinecurve with following geometrical modifications |
| **beziecurve** | beziecurve | Creation beziecurve with following geometrical modifications |
| **bsplinecurve** | bsplinecurve | Creation bsplinecurve with following geometrical modifications |
| **circle** | circle | Creation circle with following geometrical modifications |
| **ellipse** | ellipse | Creation ellipse with following geometrical modifications |
| **hyperbola** | hyperbola | Creation hyperbola with following geometrical modifications |
| **iso** | uiso<br>viso | Creation isolines with following geometrical modifications |
| **law** | law | Creation law with following geometrical modifications |
| **line** | line | Creation line with following geometrical modifications |
| **parabola** | parabola | Creation parabola with following geometrical modifications |
| **project** | project | Creation projection with following geometrical modifications |
| **revsurf** | revsurf | Creation surface of revolution with following geometrical modifications |

### 5.1.12. Group "heal"

This group allows testing the functionality provided by ShapeHealing toolkit.

DRAW module: XSDRAW

| Grid | Commands | Functionality |
|---|---|---|
| **fix_shape** | fixshape | Shape healing |
| **fix_gaps** | fixwgaps | Fixing gaps between edges on a wire |
| **same_parameter** | sameparameter | Fixing non-sameparameter edges |
| **fix_face_size** | DT_ApplySeq | Removal of small faces |
| **elementary_to_revolution** | DT_ApplySeq | Conversion of elementary surfaces to revolution |

| Grid | Commands | Functionality |
|---|---|---|
| **direct_faces** | directfaces | Correction of axis of elementary surfaces |
| **drop_small_edges** | fixsmall | Removal of small edges |
| **split_angle** | DT_SplitAngle | Splitting periodic surfaces by angle |
| **split_angle_advanced** | DT_SplitAngle | Splitting periodic surfaces by angle |
| **split_angle_standard** | DT_SplitAngle | Splitting periodic surfaces by angle |
| **split_closed_faces** | DT_ClosedSplit | Splitting of closed faces |
| **surface_to_bspline** | DT_ToBspl | Conversion of surfaces to b-splines |
| **surface_to_bezier** | DT_ShapeConvert | Conversion of surfaces to bezier |
| **split_continuity** | DT_ShapeDivide | Split surfaces by continuity criterion |
| **split_continuity_advanced** | DT_ShapeDivide | Split surfaces by continuity criterion |
| **split_continuity_standard** | DT_ShapeDivide | Split surfaces by continuity criterion |
| **surface_to_revolution_advanced** | DT_ShapeConvertRev | Convert elementary surfaces to revolutions, complex cases |
| **surface_to_revolution_standard** | DT_ShapeConvertRev | Convert elementary surfaces to revolutions, simple cases |

### 5.1.13. Group "mesh"

This group allows testing shape tessellation (BRepMesh) and shading.

DRAW modules: MODELING (package MeshTest), VISUALIZATION (package ViewerTest)

| Grid | Commands | Functionality |
|---|---|---|
| **advanced_shading** | vdisplay | Shading, complex shapes |
| **standard_shading** | vdisplay | Shading, simple shapes |
| **advanced_mesh** | mesh | Meshing of complex shapes |
| **standard_mesh** | mesh | Meshing of simple shapes |
| **advanced_incmesh** | incmesh | Meshing of complex shapes |
| **standard_incmesh** | incmesh | Meshing of simple shapes |
| **advanced_incmesh_parallel** | incmesh | Meshing of complex shapes, parallel mode |
| **standard_incmesh_parallel** | incmesh | Meshing of simple shapes, parallel mode |

### 5.1.14. Group "mkface"

This group allows testing creation of simple surfaces.

DRAW module: MODELING (package BRepTest)

| Grid | Commands | Functionality |
|---|---|---|
| **after_trim** | mkface | |
| **after_offset** | mkface | |
| **after_extsurf_and_offset** | mkface | |
| **after_extsurf_and_trim** | mkface | |
| **after_revsurf_and_offset** | mkface | |
| **mkplane** | mkplane | |

### 5.1.15. Group "nproject"

This group allows testing normal projection of edges and wires onto a face.

DRAW module: MODELING (package BRepTest)

| Grid | Commands | Functionality |
|---|---|---|
| **Base** | nproject | |

### 5.1.16. Group "offset"

This group allows testing offset functionality for curves and surfaces.

DRAW module: MODELING (package BRepTest)

| Grid | Commands | Functionality |
|---|---|---|
| **compshape** | offsetcompshape | Offset of shapes with removal of some faces |
| **faces_type_a** | offsetparameter offsetload offsetperform | Offset on a subset of faces with a fillet |
| **faces_type_i** | | Offset on a subset of faces with a sharp edge |
| **shape_type_a** | | Offset on a whole shape with a fillet |
| **shape_type_i** | | Offset on a whole shape with a fillet |
| **shape** | offsetshape | |
| **wire_closed_outside_0_005** | mkoffset | 2d offset of closed and unclosed planar wires with different offset step and directions of offset ( inside / outside ) |
| **wire_closed_outside_0_025** | | |
| **wire_closed_outside_0_075** | | |
| **wire_closed_inside_0_005** | | |
| **wire_closed_inside_0_025** | | |
| **wire_closed_inside_0_075** | | |
| **wire_unclosed_outside_0_005** | | |
| **wire_unclosed_outside_0_025** | | |
| **wire_unclosed_outside_0_075** | | |

### 5.1.17. Group "perf"

This group allows testing of performance of OCCT visualization components.

DRAW module: VISUALIZATION

| Grid | Commands | Functionality |
|---|---|---|
| **multi_mesh_selection** | vdisplay verase vrotate vpan vmove vselect etc. | Operations with multiple mesh presentations |
| **multi_mesh_shading** | | |
| **multi_mesh_shrink** | | |
| **multi_mesh_wireframe** | | |
| **multi_object_hlr** | | Operations with multiple objects |
| **multi_object_selection** | | |
| **multi_object_shading** | | |
| **multi_object_wireframe** | | |
| **single_mesh_selection** | | Operations with single mesh |
| **single_mesh_shading** | | |
| **single_mesh_shrink** | | |
| **single_mesh_wireframe** | | |
| **single_object_hlr** | | Operations with single object |
| **single_object_selection** | | |
| **single_object_shading** | | |

| Grid | Commands | Functionality |
|---|---|---|
| **single_object_wireframe** | | |

### 5.1.18. Group "pipe"

This group allows testing construction of pipes (sweeping of a contour along profile).

DRAW module: MODELING (package BRepTest)

| Grid | Commands | Functionality |
|---|---|---|
| **Standard** | pipe | |

### 5.1.19. Group "prism"

This group allows testing construction of prisms.

DRAW module: MODELING (package BRepTest)

| Grid | Commands | Functionality |
|---|---|---|
| **seminf** | prism | |

### 5.1.20. Group "sewing"

This group allows testing sewing of faces by connecting edges.

DRAW module: MODELING (package BRepTest)

| Grid | Commands | Functionality |
|---|---|---|
| **tol_0_01** | sewing | Sewing faces with tolerance 0.01 |
| **tol_1** | sewing | Sewing faces with tolerance 1 |
| **tol_100** | sewing | Sewing faces with tolerance 100 |

### 5.1.21. Group "thrusection"

This group allows testing construction of shell or a solid passing through a set of sections in a given sequence (loft).

| Grid | Commands | Functionality |
|---|---|---|
| **solids** | thrusection | Lofting with results solid |
| **not_solids** | thrusection | Lofting with results shell or face |

### 5.1.22. Group "v3d"

This group allows testing visualization of shapes in 3d viewer.

| Grid | Commands | Functionality |
|---|---|---|
| **edge** | vdisplay verase vrotate vpan vmove vselect etc. | Display and manipulations of presentation of different shapes in 3d view |
| **edge_face** | | |
| **edge_solid** | | |
| **face** | | |
| **vertex** | | |
| **vertex_edge** | | |
| **vertex_face** | | |
| **vertex_solid** | | |
| **vertex_wire** | | |
| **wire** | | |
| **wire_solid** | | |

### 5.1.23. Group "xcaf"

This group allows testing extended data exchange packages.

| Grid | Commands | Functionality |
|---|---|---|
| **dxc** | | Subgroups are divided by format of source file, by format of result file and by type of modification of the document ( For example: word combination **brep_to_igs** means that the source shape (format brep) was added to the document, after that, the document was saved into igs format. Additional words **add_CL** mean initialization of colors and layers in the document before saving, and additional words **add_ACL** mean creation of an assembly and initialization of colors and layers in a document before saving) |
| **dxc_add_ACL** | | |
| **dxc_add_CL** | | |
| **igs_to_dxc** | | |
| **igs_add_ACL** | | |
| **brep_to_igs_add_CL** | | |
| **stp_to_dxc** | | |
| **stp_add_ACL** | | |
| **brep_to_stp_add_CL** | | |
| **brep_to_dxc** | | |
| **add_ACL_brep** | | |
| **brep_add_CL** | | |

### 5.1.24. Group "xml"

This group allows testing standard and XML persistence of OCAF component.

| Grid | Commands | Functionality |
|---|---|---|
| **ocaf_std** | Save<br>Open | Save and restore of OCAF and XDE documents to standard and XML format |
| **ocaf_xml** | | |
| **xcaf_std** | | |
| **xcaf_sml** | | |

## 5.2. Mapping of OCCT functionality to grid names in group "bugs"

| OCCT Module / Mantis category | Toolkits | Test grid in group bugs |
|---|---|---|
| **Application Framework** | PTKernel<br>TKPShape<br>TKCDF<br>TKLCAF<br>TKCAF<br>TKBinL<br>TKXmlL<br>TKShapeSchema<br>TKPLCAF<br>TKBin<br>TKXml<br>TKPCAF<br>FWOSPlugin<br>TKStdLSchema<br>TKStdSchema<br>TKTObj<br>TKBinTObj<br>TKXmlTObj | **caf** |

| OCCT Module / Mantis category | Toolkits | Test grid in group bugs |
|---|---|---|
| **Draw** | TKDraw<br>TKTopTest<br>TKViewerTest<br>TKXSDRAW<br>TKDCAF<br>TKXDEDRAW<br>TKTObjDRAW<br>TKQADraw<br>DRAWEXE<br>Problems of testing system | **demo** |
| **Foundation Classes** | TKernel<br>TKMath<br>TKAdvTools | **fclasses** |
| **Shape Healing** | TKShHealing | **heal** |
| **Mesh** | TKMesh<br>TKXMesh | **mesh** |
| **Data Exchange** | TKIGES | **iges** |
| | TKSTEPBase<br>TKSTEPAttr<br>TKSTEP209<br>TKSTEP | **step** |
| | TKXSBase<br>TKXCAF<br>TKXCAFSchema<br>TKXDEIGES<br>TKXDESTEP<br>TKXmlXCAF<br>TKBinXCAF | **xde** |
| **Modeling_algorithms** | TKGeomAlgo<br>TKTopAlgo<br>TKPrim<br>TKBO<br>TKBool<br>TKHLR<br>TKFillet<br>TKOffset<br>TKFeat<br>TKXMesh | **modalg*** |
| **Modeling Data** | TKG2d<br>TKG3d<br>TKGeomBase<br>TKBRep | **moddata*** |
| **Visualization** | TKService<br>TKV2d<br>TKV3d<br>TKOpenGl<br>TKMeshVS<br>TKNIS<br>TKVoxel | **vis** |

## 5.3. Recommended approaches to checking test results

### 5.3.1. Shape validity

Run command "checkshape" on the result shape of the test (final or intermediate) and make sure that parse.rules of the test grid or group will report bad shapes (usually recognized by word "Faulty") as error.

Example

```
checkshape result
```

### 5.3.2. Shape tolerance

The maximal tolerance of sub-shapes of each kind of the resulting shape can be extracted from output of **tolerance** command as follows:

```
set tolerance [tolerance result]
regexp { *FACE +: +MAX=([-0-9.+eE]+)} $tolerance dummy max_face
regexp { *EDGE +: +MAX=([-0-9.+eE]+)} $tolerance dummy max_edgee
regexp { *VERTEX +: +MAX=([-0-9.+eE]+)} $tolerance dummy max_vertex
```

### 5.3.3. Shape volume, area, or length

Use command **vprops**, **sprops**, or **lprops** to measure volume, area, or length of the shape produced by the test. The value can be extracted from the result of the command by **regexp**.

Example:

```
# check area of shape result with 1% tolerance
regexp {Mass +: +([-0-9.+eE]+)} [sprops result] dummy area
if { abs($area - $expected) > 0.1 + 0.01 * abs ($area) } {
    puts "Error: The area of result shape is $area, while expected $expected"
}
```

### 5.3.4. Memory leaks

The test system measures the amount of memory used by each test case, and considerable deviations (as well as overall difference) comparing with reference results will be reported by **testdiff** command.

For checking memory leak on a particular operation, typical approach is to run this operation in cycle measuring memory consumption at each step and comparing it with some threshold value.

The file **begin** in group **bugs** defines command **checktrend** that can be used to analyze sequence of memory measurements to get statistically based evaluation of the leak presence.

Example:

```
set listmem {}
for {set i 1} {$i < 100} {incr i} {
    # run suspect operation
    …
    # check memory usage (with tolerance equal to half page size)
    lappend listmem [meminfo h]
    if { [checktrend $listmem 0 256 "Memory leak detected"] } {
        puts "No memory leak, $i iterations"
        break
    }
}
```

### 5.3.5. Visualization

Take a snapshot of the viewer, give it the name of the test case, and save in the directory indicated by Tcl variable **imagedir**. Note that this variable is pointing to the log directory if command **testgrid** is running, or to temporary directory of the current folder if the test is run interactively.

```
vinit
vclear
vdisplay result
vsetdispmode 1
vfit
vzfit
vdump $imagedir/${casename}_shading.png
```

This image will be included in the HTML log produced by **testgrid** command and will be checked for non-regression through comparison of images by command **testdiff**.

## 6. REFERENCES

[1] DRAW Test Harness User's Guide

[2] OCCT MantisBT issue tracker, http://tracker.dev.opencascade.org