TestCenter Manual

TestCenter Manual

Copyright © 2003-2020 MeVis Medical Solutions

Published 2020-11-12

Table of Contents

1. Introduction 6
    1.1. Supported Testing Concepts 6
    1.2. Requirements 6
2. Design 8
    2.1. TestCases 8
        2.1.1. Defining a Test Case 8
        2.1.2. Implementing the Test Functions 9
        2.1.3. setUp and tearDown Functions 10
        2.1.4. Logging and Status Detection 11
        2.1.5. Handling Changes to the Environment 11
        2.1.6. Test Case Suites 12
    2.2. Master-Slave Approach 12
    2.3. TestSupport Python Package 13
    2.4. Handling Input/Output Data 14
        2.4.1. Handling Ground Truth Data 14
        2.4.2. Handling Result Data 15
        2.4.3. Result Handling 15
3. Running The TestCenter 16
    3.1. Local Testing From Within MeVisLab 16
        3.1.1. TestCaseManager 16
        3.1.2. Module Tests 16
    3.2. Local Testing from the Command Line 17
4. TestCenter Reports 19
    4.1. Static HTML Pages 19
5. TestCenter Configuration File 22
6. Tips and FAQ 23
    6.1. Tips 23
    6.2. FAQ 23

List of Figures

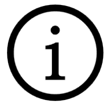
4.1. Example Report: Index Page 19
4.2. Example Report: LocalImageTest 20
4.3. Example Report: LocalFileNameTest — All Tests 20
4.4. Example Report: LocalFileNameTest — Failed Tests Only 21

List of Tables

2.1. Keys for Test Case Definition 9
3.1. TestCenter Command Line Options 18

Chapter 1. Introduction

The TestCenter is a framework for testing different aspects of MeVisLab. Tests are specified in the Python scripting language. In the following chapter, testing concepts and requirements of the TestCenter are introduced.



Tip

For a hands-on introduction to developing test cases, see the Getting Started, “Using the TestCenter”.

1.1. Supported Testing Concepts

This section will introduce the testing concepts that are supported in the TestCenter. They are *unit testing*[1], *regression testing*[2], and the so-called *formal testing* .

- Unit tests are performed to gain confidence into the validity of the implemented algorithms by verifying them in tests. Unit tests have to be designed and written specifically for each algorithm and function such that they detect as many problems as possible. The result is a live specification for the algorithm that can easily be verified.
- Formal tests are related to MeVisLab. They test whether a group of MeVisLab modules meets certain formal aspects, for example having meta information like module documentation
- Regression testing is done by repeatedly performing the available tests (unit and formal). It is useful to detect changes that break algorithms. With the integration of the TestCenter into the build process, the test can be executed for each build on every supported platform.

The tests can be of the functional or generic type. The test type needs to be given in the test definition.

- *Functional tests* are used for unit tests. With them, a specific feature is tested, for example an SQL database interface.
- *Generic tests* are used for formal tests of a larger set of modules. This kind of test is executed generically for all those modules, for example to verify that the modules' field names follow the specification.

1.2. Requirements

The following requirements were found to be important for the TestCenter design:

- Easy creation and maintenance of tests: Especially for unit testing it is essential that writing new tests is simple and that new tests automatically integrate with the testing framework to get executed.
- Tests must be able to be run in two possible scenarios:
  - Local execution is useful for all testing done during development of test cases or modules.
  - Some kind of centralized execution, for example as part of the build system, is useful for automatic testing to perform regression tests.
- The framework is required to detect crashes and infinite loops so that the testing process can be restarted. Otherwise each time such an incident happens, the process will stop and not produce any results.

[1]see http://en.wikipedia.org/Unit_testing
[2]see http://en.wikipedia.org/Regression_testing

- After testing, the generated results must be handled:
  - In case of local testing, they should be written to static HTML files so that users can view and archive them.
  - In case of automatic testing, there is a large amount of results with the additional requirement that regression information must be made visible. The results must be stored in a central location and there must be an interface that shows an overview of results over time and that allows to compare test runs.

Chapter 2. Design

In the following chapter, the design of the TestCenter is discussed.

- Section 2.1, “TestCases”
- Section 2.2, “Master-Slave Approach”
- Section 2.3, “TestSupport Python Package”
- Section 2.4, “Handling Input/Output Data”

2.1. TestCases

The test cases are the key elements of the testing framework. To add a new one, two steps are necessary:

- Defining the basic parameters of the test case, like test type, name and script file. With this information, the TestCenter knows about the test case.
- Defining the actual test, for example adding the instructions that should be executed to the script file defined above.

2.1.1. Defining a Test Case

Test cases use the MeVisLab infrastructure. Similar to the database for modules, there is a database for test cases. This database is filled with entries by searching for `*.def` files in the `TestCases` subdirectories of all packages. Each `*.def` file may contain the definition of multiple test cases.

The two test types are defined by blocks of source like the following:

```
FunctionalTestCase FunctionalExample {
  scriptFile = $(LOCAL)/FunctionalExample.py
}

GenericTestCase GenericExample {
  scriptFile = $(LOCAL)/GenericExample.py
}
```

The given example creates a functional and a generic test case. The information provided is collected in an entry that can be accessed using the `testCaseInfo` method of the TestCaseDatabase.

The `scriptFile` key specifies which file contains the actual test functions (see Section 2.1.2, “Implementing the Test Functions”). The script files are used to load the actual test cases, which behave like macro modules in the MeVisLab context. This has the benefit that a network named after the test case and located in the same folder (for example `$(LOCAL)/GenericExample.mlab`) is loaded as part of the context, too.

Table 2.1. Keys for Test Case Definition

Key	Description
author	Comma-separated list of author names. For functional tests, these authors are used to send email notifications for failed automatic tests.
comment	A short information what the test case is going to test.
timeout	Seconds the test case should take at most. If it takes longer, it is terminated. The default value is given in the TestCenter configuration file, see Chapter 5, *TestCenter Configuration File*.
scriptFile	The file containing the actual test specification.
testGroups	Comma-separated list of group names the test belongs to. This information can be used to run the test cases of only one group (for example all test cases of the `examples` group). There are two special test groups: `automatic` and `manual`. All test cases that do not belong to the `manual` group are part of the `automatic` group by default. Test cases that for some reason cannot be used in automatic testing should be a part of the `manual` group.
dataDirectory	Path to a directory containing the ground truth data for the test case. The default value is "$(LOCAL)/data". This directory should used as if it were read-only. See Section 2.4.1, "Handling Ground Truth Data" for further information.
showTestFunctionSortingLiterals	If set to True/Yes/1, the sorting literals (the string between TEST and the first underscore before the actual test name) is shown in the function list in the TestCaseManager and in the resulting HTML report for an easy referencing.
preferredRenderer	Optional key that indicates what type of renderer should be used for testing. Possible values are `software` and `hardware`. When the key is set to `software` the test case is run using software rendering. When it is set to `hardware` hardware rendering is used for the test case. If software rendering has been forced via the environment variable `MLAB_FORCE_MESA` a warning about conflicting settings is logged. When the key is not set rendering uses the system default.

All test cases must reside in a package's `TestCases` directory. It is recommended to use a substructure similar to the modules directory (for example `TestCases/FunctionalTests/[module_type]/[module_name]`).

The location of generic tests is important as the set of modules being tested by them depends on the package they are in.

- In general, a generic test is run on all modules in that package.
- If the generic test is located in a group's `Foundation` package, all modules of that group will be tested.
- Generic tests located in the `MeVis/Foundation` package will be executed for every module available.

2.1.2. Implementing the Test Functions

Test cases are scripted in Python. Each test case consists of an arbitrary number of test functions. These functions all have their own status. The worst status of a test case's functions will determine the overall status, see Section 2.1.4, "Logging and Status Detection".

Similar to the Google test framework, test functions are given a special prefix like `TEST`.

There are three major types of test functions that can be used: simple, iterative and field-value test functions. In addition to this, multiple simple test functions can be combined to a group. The following sections describe the available test function prefixes and types.

2.1.2.1. Simple Functions

`TEST`: The simple test function consists of a single function that is run once.

2.1.2.2. Advanced Functions

For the more advanced test function types, the concept of virtual test functions was introduced. A simple test function maps to a single virtual test function, like each iteration of an iterative test function is mapped to one. The virtual functions are named after the non-virtual parent function. For simple test functions, both names match. For field-value tests, the name of the field-value test case executed in the test function is appended. The iterative tests are more complicated: In case of returning a list, the index of the parameter will be appended to the function name. If a dictionary is returned, the keys are appended to the name and the values of the keys are used as parameters.

The following advanced test types are available:

- `ITERATIVTEST`: Iterative test functions run a function for every specified input. They return a tuple consisting of the function object called and the inputs iterated over. The following example would call the function `f` three times with the parameter being `1` in the first call, `2` in the second and `3` in the third.

  ```
  def ITERATIVETEST_ExampleTestFunction ():
    return [1, 2, 3], f

  def f (param):
    print param
  ```

  The iterative test functions are useful if the same function should be applied to different input data. These could be input values, names of input images, etc.

- `FIELDVALUETEST`: The field-value test functions integrate the field-value tests concept into the TestCenter. The path to the XML file containing the field-value test specification is returned and an optional list of contained field-value test cases. The following example would execute all field-value test cases defined in the `fieldvaluetestcase.xml` file located in `$(LOCAL)/data`.

  ```
  def FIELDVALUETEST_ExampleTestFunction ():
    return "$(LOCAL)/data/fieldvaluetestcase.xml"
  ```

  The field-value test case concept was introduced to be able to easily set a parameterization in the network and verify the values of different fields.

- `GROUP`: This test function combines multiple simple test functions to a named group. These functions just return the function objects of the simple test cases that should be member of the group. There is no technical reason for having these test groups but they improve the understanding of complex test cases with many test functions.

- `UNITTEST`: This test function wraps existing Python tests implemented with the `unittest` module. It returns a TestSuite that can contain test functions and other TestSuites. For each unit test function, a test function is added to a group with the name of the wrapper function. This test function is named after the test suite and unit test function.

  ```
  from backend import getSuite
  def UNITTEST_ExampleUnitTestWrapper ():
    return getSuite()
  ```

All function names follow the `PREFIX[order]_[name]` syntax. The `[order]` substring can be an arbitrary number of numbers that will define the order of the test functions. The `[name]` part is later used to identify the method and should therefore be selected with care.

2.1.3. setUp and tearDown Functions

Two special functions are available that can be used to maintain global information for a test case. They are called `setUpTestCase` and `tearDownTestCase` and are called prior to the execution of the first

test function and after the last, respectively. The outputs of `setUpTestCase` or `tearDownTestCase` are appended to the first / last test function.

Note

Please note that this is different from what is common in other unit-testing frameworks; there, the `setup` function would usually be called prior to *every* test function and `teardown` after *every* function.

2.1.4. Logging and Status Detection

A testing framework must somehow determine whether a test has failed or succeeded.

The MeVisLab debug console shows messages with a type. All sources of output are mapped to this console: standard output, standard error and other special methods provided inside MeVisLab for logging. Based on this information, the TestCenter determines the current test function status.

Aside of these status types, there are the TestCenter-specific types “Timeout” and “Crash”. They are used for situations where the testing fails. For more information on failure handing, see Section 2.2, “Master-Slave Approach”.

The test case status is determined by the worst status of one of its test functions. In addition to this, the TestCenter collects all messages printed in the debug console to display them in the reports for detailed introspection on what happened.

The following example shows a simple test function that verifies a condition and prints an error message if the condition was not met. This results in the test function having the status “Error”.

```
import mevis
def TEST_Simple_Test ():
  if not condition:
    mevis.MLAB.logError("The condition wasn't met!")
  else:
    mevis.MLAB.log("Everything is fine.")
```

As the mechanism leads to messages being collected from resources besides the test functions, it allows detecting all possible problems occurring while testing. For example, if an ML module fails and prints messages of type “ML Error”, this will result in the test function being marked as failed.

Messages coming from the TestCenter itself may look different to messages from modules. This way, irrelevant parts of the messages can be filtered out more easily. The distinction is done using special logging methods inside the TestCenter which are part of the TestSupport Python package (see Section 2.3, “TestSupport Python Package”). Example:

```
from TestSupport import Logging
def TEST_Simple_Test ():
  if not condition:
    Logging.error("The condition wasn't met!")
  else:
    Logging.info("Everything is fine.")
```

Tip

Other examples for info output methods would be `print "The variable has changed."` and `MLAB.log("The variable has changed.")`, but using the Logging Python module makes it easy to distinguish the test as message source.

2.1.5. Handling Changes to the Environment

During a test, the environment is usually changed. For example, a test function would set field values and touch triggers to start a computation based on the set values. This is a problem when the next test function is influenced by these changes.

- Unit testing frameworks prevent this problem by reloading the environment for every single test function. In case of the TestCenter, this is not an option as MeVisLab and the test network might take tremendous effort to load.
- However, test functions also cannot rely on the state of previous test functions as there is a mechanism to detect crashes and retry the according test function. This second attempt would have a different state than expected as the previous functions are not run again.
- Manually resetting all values a computation relies on would be possible but also a daunting task.

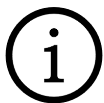
In the TestCenter, the problem is solved by the TestSupport.Fields Python module. It provides methods to set a field value in a way that makes resetting the original values possible.

The foundation is the ChangeSet concept (see the TestCenterReference, ChangeSet.py). It relies on the idea that a subset of input field values are set to parameterize the following computation. Setting the input values to their original values should be sufficient to (re)set the environment to an expected state. This is only heuristic!

Note

The ChangeSets only work for field value changes made using the according methods. They have no effect on indirect changes, for example changes due to the computation started by touching trigger buttons.

The TestCenter itself maintains a stack of such ChangeSets to split the TestCase execution into logical blocks. A first ChangeSet is pushed while loading a TestCase and popped while destructing it. When running a test function, a second ChangeSet is pushed before and popped afterwards.



Tip

You can also use the stack to push your own ChangeSets (see the `pushChangeSet` and `popChangeSet` methods of the TestSupport.Base Python module, see Section 2.3, "TestSupport Python Package"). Make sure to clean up the stack afterwards, otherwise error messages will be printed.

The output values could also be input values of the next computation. However, this requires a special handling like a manual reset to circumvent the ChangeSet concept.

2.1.6. Test Case Suites

Test cases can be grouped into test case suites. This allows to run several related tests via the TestCaseManager. Test case suites can be declared in any `*.def` file below the `TestCases` subdirectory of any packages. Each `*.def` file may contain the definition of multiple test case suites and test cases.

The following code shows how to define a TestCaseSuite:

```
TestCaseSuite ExampleTestCaseSuite {
  testCases = "ExampleTest1, ExampleTest2, ..."
}
```

The available test case suites are listed in the TestCaseManager and can be selected and executed. The result report will show an overview of all executed tests and links to the individual tests.

2.2. Master-Slave Approach

As the automatic testing should work without being interrupted even in cases of crashes or infinite loops, a separate MeVisLab instance is run for executing the tests. This setup is quite common and called master-slave approach.

- The master can be started via MeVisLab, TestCenter Coordinator or command line.

- The slave is a MeVisLab instance that automatically loads the TestCenter macro module, and is controlled by the master from the outside so that it can be interrupted and restarted if necessary.

It is important to note the difference between crash and timeout:

- If the slave process crashes, the master will notice this and react appropriately. As the crash could be due to a side effect of a previous test case or function, the master will retry the crashed test function once. Afterwards the remaining test functions are called (with consideration to the timeout value). After each restart, the `setUpTestCase` method is called for the test case.

- The timeout is a value given in the test case definition or via the default of twenty seconds. It specifies the time after which the test case should have finished, or the master will assume the slave was trapped in an infinite loop. The solution to this situation is to restart the slave process and proceed to the next test case. No retry is attempted as the timeout value may be very large and this would lengthen the test process enormously.

One of the main problems with the master-slave approach is that it requires some sort of communication between the two entities. This is solved by using an asynchronous socket-based communication. It has the benefit that crash and timeout detection comes for free when using non-blocking communication. Each operation is coupled with a timeout such that a failure will not block the session.

Basic Description of the Socket Implementation

The message length must be known to the receiver so that it can detect the end of transmission. Therefore, at least two messages are sent. The first message is of fixed length (16 bytes) and contains the right-aligned length of the string that should be sent as second message. This string is sent split into blocks of a size of max. 8192 bytes.

This low-level protocol is used to send commands encapsulated in XML nodes between the master and the slave. The command type is specified as type attribute in an XML node with parameters given as sub nodes. The slave will process the command and return an XML node as response, containing a status and the results.

The following list gives all commands and their supported parameters:

- *Quit*: Used to make slave terminate itself. Requires no parameters.

- *DoTest*: Lets the slave execute a given test case. A sub node with the tag “TestCase” is given that contains information on the current test case (name and type of test case and the module to be tested in case of a generic test case) and the list of test functions that need to be run.

- *SetPackageList*: Contains a sub node with the tag “Packages” with information on all packages that are assumed to be available. This is required for some tests that must deceide whether certain modules are available by taking into account only the specified packages.

- *BuildTestAgenda*: As the master must not run inside MeVisLab, the slave must gather which test cases and modules have to be taken into account. The master sends the request with a subnode with the tag “Configuration” that contains information on what the settings are for the current test run. This includes information on preselected packages, test cases, and modules, for example. The slave will return a list of XML nodes that contain the TestCase information used when sending the `DoTest` command.

For local testing (see Section 3.1, “Local Testing From Within MeVisLab”) it is possible to run test cases in the actual MeVisLab instance. In that case, the slave Python script is imported and function calls are used instead of the socket based communication.

2.3. TestSupport Python Package

For repeated and central tasks in the testing process, the TestSupport Python package was added to the TestCenter to provide methods to simplify test creation and prevent errors and different solutions

for the same problems. This section will only give a short overview of all available methods. For further information, see the TestCenter Reference.

The following list offers short descriptions for some of the TestSupport package Python modules.

- The *Async* Python module provides methods for test cases that need to handle background tasks.
- The *Fields* Python module provides methods to manipulate fields.
- The *Logging* Python module is required to create so-called internal messages in the report. In addition, there are methods to display images or links to files in the report.
- The *Macros* Python module offers an API (similar to Google Test) to evaluate expressions and compare values.
- The *ScreenShot* Python module adds support for creating screenshots of panels and images.

2.4. Handling Input/Output Data

2.4.1. Handling Ground Truth Data

In almost all cases, testing requires verifying computed results against some sort of ground truth data. This data could be anything from simple numbers to images. While simple data types are part of the test case scripting, files must be made available for the testing process.

2.4.1.1. Using local data

Each test case can have a directory for all required data. By default, this directory with name “data” is located in the local folder of the test case. By using the “dataDirectory” keyword in the definition file, this default value can be changed.

Having ground truth data in the test case folder has one major drawback: it requires the data to be checked in to the version control system and to be part of the installer. This is not an option if the data gets larger than a few megabytes. Therefore there is an option to use external data storage like a network share.

2.4.1.2. Using an external data storage

Note

Never try to connect to a network share directly via a fixed URL like `\\fileserver\testcenter-data` as this is not supported on all platforms and will cause problems if the resource is not available.

To connect external data storage, use the `ctx.expandFilename` method. It expands variables either predefined or as set in the MeVisLab preferences file. This way, the path to the share can be given in the `*.prefs` file and the share can be loaded on all platforms. Having the share defined in the `*.prefs` file instead of a dozen test files also makes changing the path much easier, for example if the share has to move.

The following `*.prefs` file would define two shares for TestCenter data that for usage on windows:

```
Settings {
  TESTCENTER_EXTERNAL_DATA_MMS = \\fs.mms.example\tc
  TESTCENTER_EXTERNAL_DATA_FME = \\fs.fme.example\tc
}
```

With these two variables set, the filename `${TESTCENTER_EXTERNAL_DATA_MMS}/testcasename/filename` can be used to get access to the given filename using the `expandFilename` method to get the actual path.

Note

`expandFilename` will only expand known variables. If a test case is instanciated on a site that does not have access to the resource, the variable is not specified and the result will be an invalid path and an error. To prevent problems, use the `Base.getExternalDataDirectory` method that returns NULL if the variable is not defined, and the path otherwise.

2.4.2. Handling Result Data

Test cases must be able to create results. The result files must be saved and added to the report. The location for saving files should be retrieved using the `getResultDirectory` method. Using other directories is possible, but not recommended. Only the retrieved result directories will be cleaned up when starting the next test run.

The files created in test functions will not appear in the report automatically. They must be added explicitly using one of the following two methods:

- `showFile`: adds download links for files
- `showImage`: directly inserts images into the report

Both methods will add special messages to the logfile. The result handlers will detect these messages, copy the linked files, and add them to the report.

2.4.3. Result Handling

The master will collect all the results the slave created while testing and store them for later usage. As local and automatic testing have different requirements for presenting the results, the master uses an XML file as an intermediate representation of the results. Results are saved to a file after all tests have been finished.

The structure of the result file is mostly taken from the responses sent by the slave. It contains information on all executed test cases, all modules being tested, and the results of the actual tests. This information is used to create a detailed report.

Note

The further processing of the XML file is not up to the TestCenter but the calling mechanism. For local testing there are methods that convert to static HTML pages. For automatic testing, those results are written to a database. The presentation of the results is therefore up to another entity. For more information see Chapter 4, *TestCenter Reports*.

Chapter 3. Running The TestCenter

The following chapter gives an overview of how the TestCenter is run depending on the use case.

- Section 3.1, “Local Testing From Within MeVisLab”
- Section 3.2, “Local Testing from the Command Line”

3.1. Local Testing From Within MeVisLab

Testing from within MeVisLab will mostly be relevant when developing test cases, new features, or new modules. This follows the unit-testing approach to first write a test case that will later verify the correctness of the implemented features.

Inside MeVisLab there are two options to run the TestCenter: the TestCaseManager (actually a macro module) and the Module Tests functionality as an integration of the TestCenter in the standard MeVisLab user interface.

3.1.1. TestCaseManager

Tip

For a hands-on introduction to working with the TestCenter, see the Getting Started, “Using the TestCenter”.

The testcase-centric *TestCaseManager* is started via **File** → **Run TestCaseManager**, or by adding the `TestCenterManager` macro module to the network. It is used to create, edit and run functional test cases and set some basic TestCenter configuration. For the configuration file, see Chapter 5, *TestCenter Configuration File*.

The TestCenterManager offers an option to use the same instance of MeVisLab (leave **Secure testing** unchecked). This is much faster than starting a separate instance but has the drawback that crashes will take down the testing MeVisLab instance.

The TestCaseManager incorporates its own result viewer that uses the Webkit integration in MeVisLab to show the resulting static HTML files, see Section 4.1, “Static HTML Pages”. Its advantage is that it can handle special schemes (part of the links), for example to open files at the according line.

For standard browsers, these schemes will cause display problems. To avoid this, go to **TestCaseManager** → **Test Reports**, select a report and click the button **Create External** to create HTML output suitable for standard browsers.

3.1.2. Module Tests

The Module Tests functionality is module-centric. It runs the test cases available for a set of modules. This includes the generic test cases that apply to the given modules and the functional test cases that are associated with them. The connection between a module and a test case is done using the `associatedTests` keyword in the `*.def` file of a module.

The Module Tests functionality can be used on three levels inside MeVisLab:

- For a single module via the context menu (**Debugging** → **Run Tests**).
- For a selection of modules the associated tests can also be called via the context menu (**Run Tests On Selection**). This is also available from the **Extras** menu. This is equivalent to the option described above.

- Via the main menu (**Extras** → **Run Module Tests**), associated test cases for a filtered subset of modules can be run. Complex filters are available.

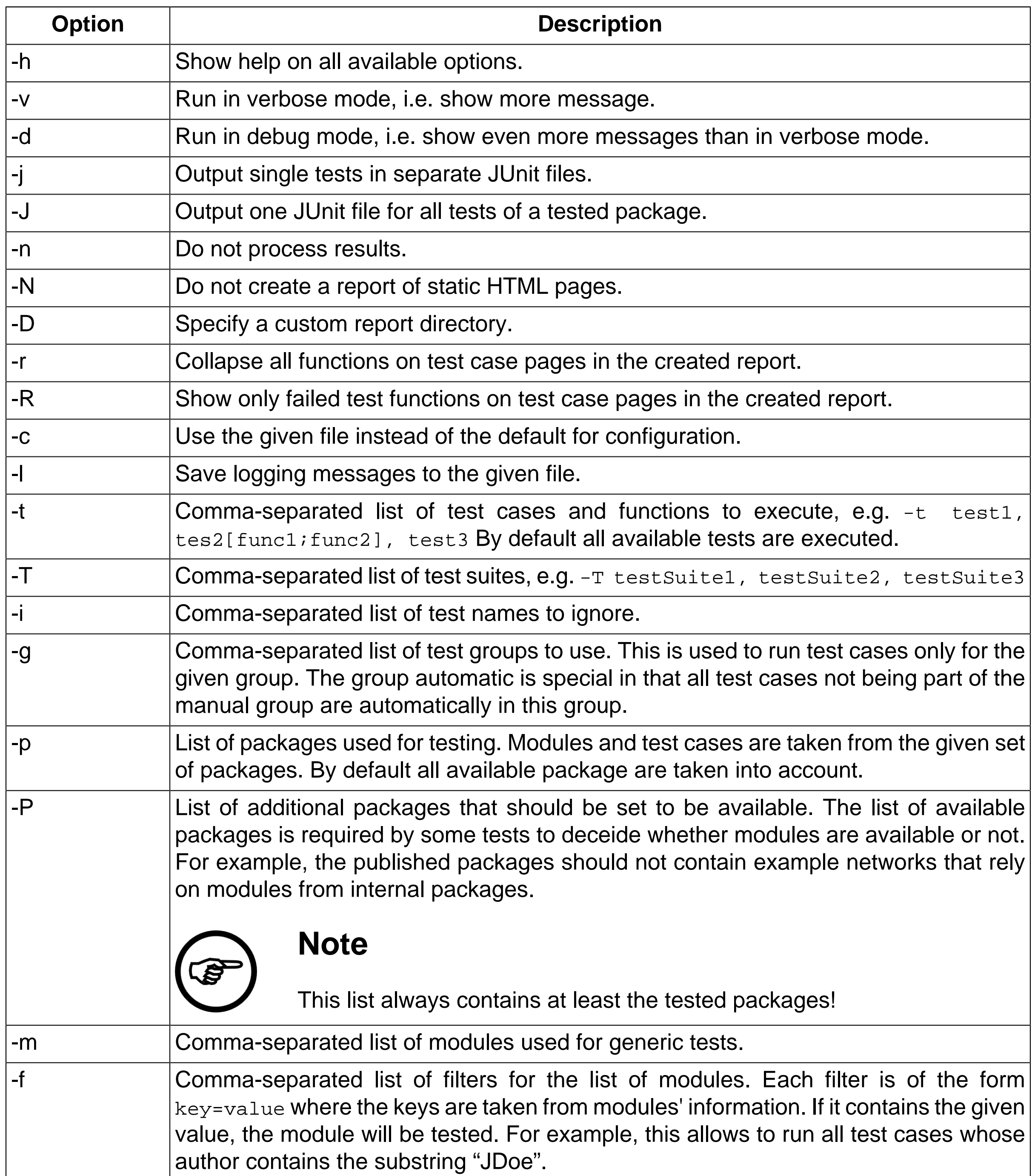
The results generated with the Module Tests functionality are converted to static HTML pages and displayed with the result viewer integrated into the TestCaseManager module.

3.2. Local Testing from the Command Line

The file `TestCenter.py` is used as script file for the hidden TestCenter module. This module is loaded via the `runmacro` option of the MeVisLab executable and will take care of loading the slave that will then connect to the master. You may start it with a batch file containing the following command where `%MLAB_ROOT%` is an environment variable set by the MeVisLab installation process. Use `$MLAB_ROOT` on unix derivatives: `"%MLAB_ROOT%\MeVis\ThirdParty\Python\MeVisPython.exe" "%MLAB_ROOT%\MeVisLab\Standard\Modules\Macros\Tests\TestCenterAdvanced\TestCenter.py"`

`TestCenter.py` can also be used from the command line, in which case it is used as a standalone Python script file. It offers a lot of helpful options to specify the tests to be executed (see Table 3.1, “TestCenter Command Line Options”).

Table 3.1. TestCenter Command Line Options



Option	Description
-h	Show help on all available options.
-v	Run in verbose mode, i.e. show more message.
-d	Run in debug mode, i.e. show even more messages than in verbose mode.
-j	Output single tests in separate JUnit files.
-J	Output one JUnit file for all tests of a tested package.
-n	Do not process results.
-N	Do not create a report of static HTML pages.
-D	Specify a custom report directory.
-r	Collapse all functions on test case pages in the created report.
-R	Show only failed test functions on test case pages in the created report.
-c	Use the given file instead of the default for configuration.
-l	Save logging messages to the given file.
-t	Comma-separated list of test cases and functions to execute, e.g. `-t test1, tes2[func1;func2], test3` By default all available tests are executed.
-T	Comma-separated list of test suites, e.g. `-T testSuite1, testSuite2, testSuite3`
-i	Comma-separated list of test names to ignore.
-g	Comma-separated list of test groups to use. This is used to run test cases only for the given group. The group automatic is special in that all test cases not being part of the manual group are automatically in this group.
-p	List of packages used for testing. Modules and test cases are taken from the given set of packages. By default all available package are taken into account.
-P	List of additional packages that should be set to be available. The list of available packages is required by some tests to deceide whether modules are available or not. For example, the published packages should not contain example networks that rely on modules from internal packages. **Note** This list always contains at least the tested packages!
-m	Comma-separated list of modules used for generic tests.
-f	Comma-separated list of filters for the list of modules. Each filter is of the form `key=value` where the keys are taken from modules' information. If it contains the given value, the module will be tested. For example, this allows to run all test cases whose author contains the substring “JDoe”.

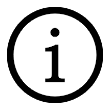
The command line interface was mostly helpful in the TestCenter development and is rarely necessary for normal testing. However, it could still useful for testing a special subset of test cases (e.g., all test cases of a test group), modules (e.g., all modules related to one author), the TestCenter itself, or associated tools.

Chapter 4. TestCenter Reports

The following chapter describes the basic features of the reports created by TestCenter results.

4.1. Static HTML Pages

The static HTML pages are created for local testing. They are written to the folder specified for **Report Directory** in **TestCaseManager** → **Configuration**. The pages are self-contained so that the complete folder can be moved without invalidating the pages. Even the XML result file created while testing is contained, so reports can be recreated later.



Tip

It is possible to write new scripts for further automatic processing of the XML result files, for example for merging or removing specific results from the reports.

In case of more than one test case, an index page with a list of all available test cases is created. Each test case has its own page that shows the results of all its test functions.

Figure 4.1. Example Report: Index Page

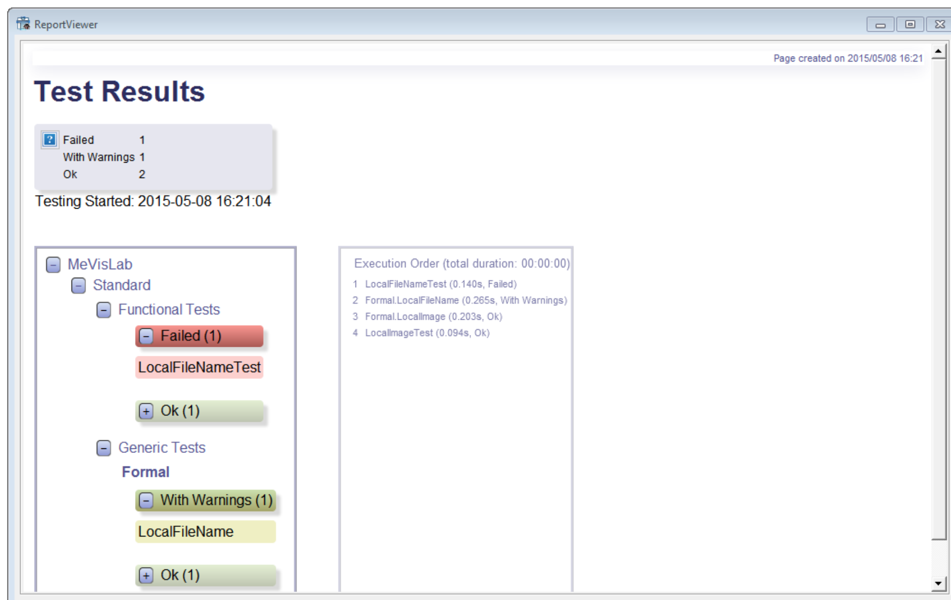
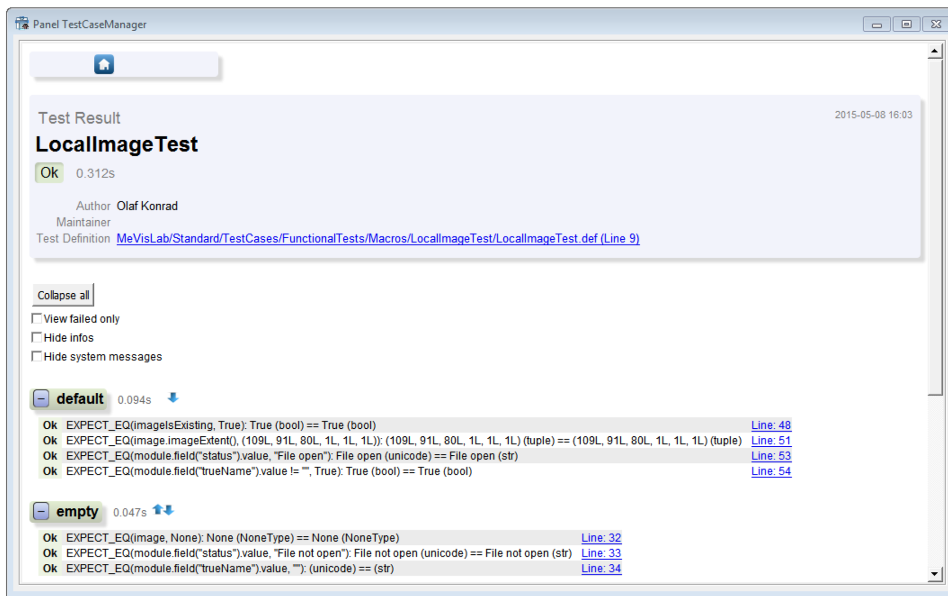


Figure 4.2. Example Report: LocalImageTest



On top of each test page there is a section with information on the test case and the general outcome. There is a configuration section that allows to configure what is displayed in the report.

Figure 4.3. Example Report: LocalFileNameTest — All Tests

It is possible to show only the failed tests. Details on the error are given in the report.

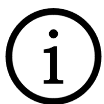
Figure 4.4. Example Report: LocalFileNameTest — Failed Tests Only

When clicking **Collapse all**, all output of all test functions is collapsed. The button then changes to **Expand all**, which will expand the outputs of all test functions.

Additionally there are checkboxes to hide info and system messages if available.

Below the header section, there is a block for each test function below a colored box that indicates the status. Clicking on the minus/plus sign before the test function name will collapse or expand the test function. Left of the name of the test function is displayed the duration of execution for this function, followed by elements to navigate to the next or previous test function.

The body of the test function block contains a list of all available messages generated by the function. For reports displayed with the TestCaseManager result viewer, the line numbers are clickable and will open the corresponding file at the correct location. The message line shows a tooltip with information on the time the message was generated.



Tip

For HTML output for external browsers, see Section 3.1.1, “TestCaseManager”.

Chapter 5. TestCenter Configuration File

The TestCaseManager configuration is written to the file `tc_cfg_[operatingsystem].xml`.

Example configuration file:

```
<Configuration>
 <mlab>
  <slavelogfile>mevislab_slave.log</slavelogfile>
  <executable usedebug="False">C:/Programme/MeVisLab2.1</executable>
  <arguments>-nosplash,-noide,-quick</arguments>
  <resultdir>C:\DOKUMENTS\USER\LOCAL\Temp\TestCenter</resultdir>
  <resultfile>tc_result.xml</resultfile>
  <timeout>20</timeout>
 </mlab>
 <report appendTestCaseName="False" appendTimestamp="False"
                   collapseFunctions="False" showFailedOnly="False">
  <directory>C:\Dokuments\user\TestCenterReports</directory>
  <name>Report</name>
  <timestamp>_%y_%m_%d__%H_%M</timestamp>
 </report>
</Configuration>
```

The `directory` variable defines the report directory (can be set in the TestCaseManager).

The `resultsdir` variable defines the folder where the results (images and XML result file) are temporarily saved before being processed to HTML or SQL by the conversion scripts.

Chapter 6. Tips and FAQ

6.1. Tips

Some hints regarding test data (or other external files)

1. In the first step for local testing, use the variable `dataDirectory` to store the base dir for all test data, so it can be changed easily, see Section 2.4.1.1, “Using local data”.
2. In a second step, if possible, place the test data on central test data servers to make sure your test runs on other computers as well, see Section 2.4.1.2, “Using an external data storage”.
3. If your test accesses files upon opening the test case, make sure that you check files for existence before trying to read them. Use the `getExternalDataDirectory` method for this, which either returns the path or NULL. Otherwise the resulting error messages will prevent the TestCaseManager from correctly loading the test (very annoying for your colleagues, especially if you do not follow rules 1 and 2).

Tip

When your tests run on external data on an inhouse server but the test case itself is delivered with MeVisLab, please check in a small test data sample. This way external MeVisLab users will also be able to run your test case and learn about its functionality.

Image sizes and the `OffscreenRender` module

The TestSupport.createOffscreenScreenShot(size=) function has a size parameter with a default of 512x512 pixels. If you need a different size, make sure the dimensions have a value of 2^x (power of two). The `OffscreenRender` module does not support arbitrary sizes on all graphics cards. Values of 64,128,256,512,1024 are good candidates.

How to debug functional test cases python code from MeVisLab

Active the Debugging mode in MATE with **Debug** → **Enable Debugging**, set a breakpoint in your Python test code and simply run the test from the TestCaseManager with **Secure Testing** disabled.

6.2. FAQ

Problem: My test does not load!

Answer: There is an error or warning message when the network is loaded or the Python script evaluated. Check the MeVisLab debug console for warning/error messages and fix them.

Problem: I would like to create screenshots of viewers but would like to be able to work while the test is running, so some viewers might be hidden by other applications.

Answer: The TestCenter supports offscreen rendering for this purpose, see the TestCenter Reference and examples (and the next question).

Problem: I'm trying to use the offscreen rendering feature to create images of a 3d scene in a `SoExaminerViewer`, but they are all empty. What is the problem?

Answer: There are two things to be aware of: For successful offscreen rendering, the `SoExaminerViewer` needs a) to have been opened at least once before the screenshot is taken (call `showViewerWindow()` in the test's `setUpTestCase()` method for each viewer), and b) an external camera (e.g. `SoPerspectiveCamera`) connected to it. Of course, you are also required to trigger “viewAll” or “applyCameraOrientation”, but this is the same as for onscreen viewing.

Problem: I would like to save computed images or base objects along with the report. How can I do that?

Answer: For each test function, you can ask the test support for the temporary test output directory using `Base.getResultDirectory()`. Save all test results to that directory and, very important, add a `Logging.showFile()` statement for each file. This ensures that a) the data is copied along with the report to the final report destination defined in the TestCaseManager, and b) there is a link in the test report pointing to the file.

Problem: My test network for a module in the 'release' package fails in the nightly tests.

Answer: In your test network, try to use only modules from the same or a more “general” package (i.e. avoid 'students' or 'personal' modules if possible). For the release package, this is especially important since the tests should also be executable without any other FME package.

Problem: I would like to execute certain tests from the command line.

You can execute the TestCenter from command line by calling `TestCenter.py` located in the `%MLAB_ROOT%\MeVisLab\Standard\Modules\Macros\Tests\TestCenterAdvanced` directory. See Section 3.2, “Local Testing from the Command Line” and TestCenter.py for parameters.

Problem: For functional algorithm testing, I would like to check or adapt some of the parameters (e.g. input data, algorithm parameters) manually before running the test. Is it possible to create a UI for a test?

Answer: Within FME, there is now prototypical support for such a thing. See the MeVis-internal bug http://gandalf.mt.mevis.lokal/bugzilla/show_bug.cgi?id=40623 for details.