Mosel testing program

0.10.1

REFERENCE MANUAL

FICO® Xpress Optimization

FICO

Mosel testing system 0.10.1 (FICO® Xpress 9.5)
Last Revised: September 2024

How to Contact the Xpress Team

Sales and Maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

Product Support

Customer Self Service Portal (online support): www.fico.com/en/product-support

Email: Support@fico.com (Please include *'Xpress'* in the subject line)

For the latest news and Xpress software and documentation updates, please visit the Xpress website at http://www.fico.com/xpress or subscribe to our mailing list.

Mosel testing program

Y. Colombani

Xpress Optimization, FICO, FICO House, Starley Way, Birmingham B37 7GN, UK
`http://www.fico.com/xpress`

Release 0.10.1

September 2024

Contents

1 Working with *moseltest* . . . 1
1.1 Input files and directory structure . . . 2
1.2 Working directory structure . . . 2
1.3 Other configuration settings . . . 3
1.3.1 Parallel execution . . . 3
1.3.2 Coverage statistics collection . . . 3
1.3.3 Report generation settings . . . 4
1.3.4 *xprmsrv* server startup . . . 4
1.3.5 Execution time limit . . . 4
1.3.6 Mosel environment variables . . . 4
1.3.7 Report format . . . 5
2 moseltest tags . . . 5
2.1 Conditional selection of tests . . . 5
2.2 Build sequence, build and run configurations . . . 6
2.3 Status values and test output . . . 7
2.4 Usage examples . . . 8
3 Return value . . . 9

1 Working with *moseltest*

The *moseltest* Mosel program is a general framework for testing Mosel using source Mosel files, C programs or Java programs. It can also be used for testing pre-compiled models, packages or applications (in the form of zip archives) and optionally collect coverage statistics during the execution of the tests. For its execution it only requires a valid Xpress installation and, if any tests are written in C or Java, the corresponding development tools:

- for Java: `java` and `javac`
- for C: `cc` and `make` (Unix); `cl` and `nmake` (Windows)

Public Git repository:

https://github.com/fico-xpress/mosel

1.1 Input files and directory structure

As input, the *moseltest* program expects either a directory containing the files to process or a single file name. This input information is passed via the `SRCDIR` model parameter (default value: `alltests`). For instance to run the model `mytest.mos`:

```
mosel moseltest SRCDIR=mytest.mos
```

When using a directory as the source for test files, any directory structure is accepted (*i.e.* all subdirectories are traversed and files are processed in alphabetical order).

To select tests to be actually processed from this directory it is possible to define an inclusion list (TINCL) and an exclusion list (TEXCL): these 2 parameters are either a list of file patterns (separated by commas) or a file name preceded by the symbol '@' (in this file each line is interpreted as a pattern, and empty or lines starting with '!' or '#' are ignored). For instance:

```
mosel moseltest SRCDIR=alltests TINCL='Book/*,unit/*' TEXCL='@toexclude'
```

Alternatively, a subset of the tests can be selected by the labels they define (see tag *labels* below): the parameters `ONLYLABELS` and `SKIPLABELS` define lists of labels (separated by commas). Tests defining any of the labels from `ONLYLABELS` are processed (all others are skipped) and tests defining any of the labels from `SKIPLABELS` are skipped.

A test can be packaged either as a *single file* (.mos, .c, .java or .mcf), as an *archive* (.tar, .tar.gz, .tgz or .zip) or as a *subdirectory* (the name of such a directory must end with `.dir`). In the case of an archive (or directory tree), either it contains only one executable file (*i.e.* a single .c, .mos, .java or .mcf file) or the main file to run must be named `main.mos`, `main.c`, `main.java`, or `main.mcf`.

Source files (extension `mos`, `c` or `java`) are compiled then executed. Configuration files (extension `mcf`) are loaded and the file to execute must be defined by the *model* tag (see Section 2).

For instance, a test model `mytest.mos` requiring a data file `mydata.txt` may be stored:

- as an archive `mytest.zip|tar|tgz|tar.gz` containing the 2 files (*i.e.* `mytest.mos` and `mydata.txt`)
- in a directory named `mytest.dir` together with its data file

1.2 Working directory structure

The program *moseltest* saves its temporary files under a working directory the location of which can be defined by the model parameter `WKDIR` (default value: `'workdir'`). Unless the model parameter `KEEPDIR` is set to 'true', this temporary directory will be deleted at the end of the testing procedure. Tests are run by separate Mosel instances: the *workers*. Each worker is executed under its own working directory located under `WKDIR` (*e.g.* `WKDIR/wk_1`), this directory is populated with the following content before a test is run:

- the program to be tested both in source and compiled form (*e.g.* mos & bim file) or
- in the case of a model, a copy of the corresponding bim file, or
- in the case of an application, the expansion of the zip file and a copy of all files located under the directory 'model_resources', or
- the expansion of the specified archive/directory tree (if any)

- a valid makefile for the host operating system
- files `out.txt` and `err.txt` (output and error streams of the program).

The environment variables `CLASSPATH` and `MOSEL_DSO` are set as necessary, in particular modules and packages are searched for in the current directory as well as in a session specific modules directory (see tags *module* and *package* below). The location of this directory is recorded in the environment variable `DSODIR` (it is located under `WKDIR/dso`). The environment variable `MOSEL_DSO` also includes the content of the parameter `LIBPATH`. The environment variable `SRCDIR` contains the absolute path to the testing directory (or file) as set via the model parameter of the same name. Thanks to the tag *save* a test may save a copy of a generated file that can be used later by another test defining the *restore* tag. Files handled by this mechanism are collected under the data directory (located under `WKDIR/data`), its full path can be retrieved from the `DATADIR` environment variable.

1.3 Other configuration settings

1.3.1 Parallel execution

The model parameter `NBW` (default value: 1) defines the number of Mosel instances (or workers) to use for running the tests. Although the order of the tests is not modified, when running several workers, tests are executed concurrently: as a consequence the results are not necessarily available in the same order and a given test cannot rely on the result of its predecessor.

By default *moseltest* will delay displaying of results such that order in the report does not depend on the number of workers. This behaviour can be changed via the model parameter `ASYNCOUT` (default value: 'false'): when it is set to 'true', the result of each test is displayed as soon as it is available.

The *order of execution* of tests can be controlled at the level of a *group* of tests. A group consists of the tests located in a given directory (ignoring subdirectories). In a group, a test with a file name that starts with a digit will be executed sequentially before the other tests. For instance in the group `[0t.mos,1t.mos,a.mos,b.mos]` the tests `0t.mos` and `1t.mos` will be run sequentially and then `a.mos` and `b.mos` will be executed in parallel after `1t.mos` has completed.

Additionally the tags (see Section 2 below) 'after' and 'sync' may be used to adjust the order of execution of tests.

- The tag 'after' defines the file name (relative to the directory of the current test) of a test that must be completed before the current test can be started. In our example above, defining this tag with the value `'b.mos'` in the test `'b.mos'` would prevent `'a.mos'` and `'b.mos'` to be run concurrently.
- The tag 'sync' disables completely parallel execution of a group starting from the test that defines it (*i.e.* all tests are run in sequence independently of the number of workers). This tag also guarantees that all tests in this group will be run on the same worker.

1.3.2 Coverage statistics collection

moseltest may be used to generate coverage statistics of packages, models or Insight apps. This mode is enabled by the definition of the parameter `COVLST` that consists in a list of packages/models (no file extension) or applications (extension '.zip') separated by spaces. The required bim files must be compiled with tracing information (option '-G') and be saved under one of the directories included in `LIBPATH`. To locate the corresponding source files *moseltest* will rely on the parameter `COVSRC` that is initialised by default with `LIBPATH`. In the case of applications it will check first the directory `'source'` of the zip archive before looking into `COVSRC` directories.

1.3.3 Report generation settings

With the default reporting format (see 1.3.7 below) *moseltest* generates a report in the file specified via parameter `REPFILE` or if no specific name is set via this parameter using the base report name from parameter `BASEREP`, optionally generating a unique report file name by appending a timestamp to it (depending on setting of `REPDATED`, with optional formatting via `REPDATEFMT`), alternatively appending to or overwriting any existing file of the specified name (setting of `REPRESET`).

The report is formatted with the maximum line length specified via `LINELEN`. The parameter `SAVELOGS` indicates whether to keep logs of successful executions (otherwise only logs of those executions that have resulted in an unexpected error status are kept). If a directory for log files is specified via `LOGDIR` along with `SAVELOGS` set to 'true', then the logs will be copied as individual files per test into the location `LOGDIR` (similarly to the unique report file, the individual filenames can be configured via `REPDATED` and `REPDATEFMT` to include a timestamp), otherwise the contents of all files is collected in the `REPFILE`.

The setting of parameter `SHOWLOGS` decides whether default streams are copied to the console when running *moseltest*.

If the model parameter `LOGWKR` is set to 'true' and multiple workers are in use, the worker ID as well as the sequence number on this worker will be reported in the logs (*e.g.* `"3#2"` indicates that the test was run on worker 3 as the 2nd test for this worker since its last restart).

When generating coverage statistics (that is, if parameter `COVLST` is defined), a summary of the statistics is displayed at the end of execution of the testing procedure, and the parameter `COVREP` specifies which additional reports must be produced. If the bit 0 (value 1) is set, annotated Mosel source files will be produced. In this mode a copy of each source file with the extension '.cov' (*e.g.* `mymod.mos.cov`) is saved into the directory `COVDIR` (default value: `'coverage'`) – in these files each line is preceded by the number of times it has been executed. If the bit 1 (value 2) of `COVREP` is set, the file `moseltest.info` will be saved under `COVDIR`: this file can be used with the *genhtml* script from the *lcov* tool[1] to generate an HTML representation of the coverage statistics. If the bit 2 (value 4) of `COVREP` is set, the file `moseltest.xml` will be saved under `COVDIR`: this is a Cobertura XML report file.

1.3.4 *xprmsrv* server startup

If any tests require an *xprmsrv* server and no such server is running before the start of *moseltest* then the parameter `PIDFILE` needs to define the name of an empty file to which write access is possible. If such a `PIDFILE` is specified then *moseltest* will startup an *xprmsrv* server and use this file to store the process ID of the server in order to be able to stop it when the execution of *moseltest* is about to terminate. Additionally, the parameter `XSRVLOG` can be stated to select a file that will collect the output of the server.

1.3.5 Execution time limit

The parameter `MAXWAIT` specifies the maximum time in seconds allowed for every individual execution of a test; the execution is interrupted by *moseltest* after this delay.

1.3.6 Mosel environment variables

If the parameter `KEEPENV` is 'true' then the `MOSEL_DSO` and `MOSEL_BIM` settings are inherited from the current environment, otherwise they are re-initialized by *moseltest*.

[1] This tool is installed by default on most Unix and MacOS platforms; for Windows we suggest to use `genhtml.perl` from https://raw.githubusercontent.com/Farigh/lcov-for-windows/master/bin/genhtml

1.3.7 Report format

The parameter `RFMT` decides how test results are reported. With the default format (`RFMT=''`) the information is displayed as a human readable text. The program may alternatively generate TeamCity Service Messages (`RFMT='TC'`), TAP-14 (`RFMT='TAP'`) and TAP-13 (`RFMT='TAP13'`) reports as well as Junit XML reports (`RFMT='JUNIT'`). In all cases the report is sent to the standard output stream of the program and includes output and error logs of failing tests (or for all tests if `SHOWLOGS` is set). The parameter `MAXLOGLEN` (default: 60000) is the maximum number of characters to save for a given stream (the beginning of the stream is stripped if this limit is exceeded). A value of 0 removes the limit and a negative value disables inclusion of logs in the report.

The default output (`RFMT=''`) can be configured to use colors for displaying the test results by setting values 1 or 2 for the parameter `COLOR` to obtain darker or brighter color shades respectively.

2 *moseltest* tags

By default, any specified test program is run and an exit status of 0 is interpreted as a success. This behaviour can be changed using tags embedded into the source program file. Tag lines are of the form:

```
!*! <tagname><sep><value>
```

Where *tagname* is the name of the tag, *sep* a separator (either a space, ':' or '=') and *value* the associated value. These tag lines can be directly incorporated into Mosel files (they are valid comments) but for C and Java files they have to be included in comments. For instance:

```
/*
!*! skip_host:winhost
!*! build tagada.bim
*/
```

For tags expecting a list (such as *labels*) several instances of the tag can be stated: its actual value will be the concatenation of all values.

The tags supported by *moseltest* fall under the following three categories:

- test selection (Section 2.1)
- build sequence and configuration (Section 2.2)
- test results (Section 2.3)

2.1 Conditional selection of tests

`labels:` space-separated list of labels associated to this test. The *moseltest* parameter `ONLYLABELS` defines a list of labels that tests must define to be processed (*i.e.* a test is kept if it defines any of the labels from this list). Similarly, parameter `SKIPLABELS` defines a list of labels for tests that must be skipped (*i.e.* a test defining any flag of this list will be skipped).

`only_host:` is the opposite of *skip_host*: the test is run only on the specified hosts

`only_sys:` is the opposite of *skip_sys*: the test is run only on the specified systems (same predefined names as *skip_sys*.

`required:` minimum or maximum version number of a component that is required for running a test. Multiple conditions can be stated for the same or different components. The

operators '>' and '<' mean '>=' and '<=' respectively. The extension needs to be specified if a component has both, a DSO and a BIM portion of the same name. Format: `mosel|module[.dso]|package[.bim] <|> MM[.mm[.rr]]`

`skip:` boolean value indicating whether the test should be skipped. This can be used to disable a test unconditionally

`skip_host:` space-separated list of host names on which the test must not be run (*i.e.* it is skipped when the procedure is run on any of these hosts). Several *skip_host* can be specified: their values are concatenated.

`skip_sys:` list of system names or processor types on which the test must not be run (*i.e.* it is skipped when the procedure is run on any of these systems). Several *skip_sys* can be specified: their values are concatenated. Valid system names are: `aix darwin linux hp-ux sunos windows` The architecture may be appended to the system name. For instance `linux` will cause the test to be skipped on all Linux machines but `linux64` will keep the test on Linux 32bit.
The processor type prefixed by a `'-'` may also be appended to the system name (*e.g.* `linux-x86_64`).

2.2 Build sequence, build and run configurations

`after:` name of a test that must have terminated before the current test can be started. The path to the file is relative to the location of the current test and cannot appear after the current test in the alphabetical order (*i.e.* test `'a.mos'` cannot request to be run after `'b.mos'`)

`build:` a list of objects to generate before running the test. This list may contain modules (.dso), packages (.bim) or executables (in this case the `make` command is used to build them). Several *build* can be specified: their values are concatenated.

`componly:` Boolean indicating whether execution of the test should be skipped. By default, each test is run after having being compiled. If this tag is set to 'true', the procedure is validated after a successful compilation.

`copy:` list of file names to be copied into the working directory before running the test.

`model:` name of a compiled model (no file extension) or an application (ends with '.zip'). This tag can only be used in a configuration file (extension .mcf), the "test" is the specified model or application that must be stored in one of the directories stated via `LIBPATH`.

`module:` name of a module. The "test" is in fact the source of a module. *moseltest* will compile this module, rename it according to this tag and save it into the session specific modules directory such that it becomes available for the following tests.

`newinst:` Boolean indicating whether a new Mosel instance must be started for this particular test.

`package:` name of a package. The "test" is in fact the source of a package. *moseltest* will compile this package, rename it according to this tag and save it into the session specific modules directory such that it becomes available for the following tests.

`parms:` list of parameters passed to the test model. Several *parms* tags might be stated, the actual parameter string is built by concatenating the different tags. Any parameter settings that are specified via the `TESTPARAMS` parameter of *moseltest* are merged with the settings made via *parms* tags.

`restart:` Boolean indicating whether the Mosel instance must be restarted after this test. All tests are run on the same Mosel instance. When this tag is set to 'true', the current instance is terminated and a new one is started for the following tests.

`restore:` name of a file to restore from a previous *save* operation. The file must have been saved by a previous test. The file copy is performed before execution of the test. The file is taken from the directory identified by `DATADIR`. Several *restore* tags may be used.

`runafter:` the name of a file (.mos, .c, .java) that must be run after the main test. This program may be used to validate the execution (*e.g.* check the content of a result file). A non-zero exit value will be interpreted as a failure.

`runbefore:` the name of a file (.mos, .c, .java) that must be run before the main test program. This program may be used to prepare the running environment for the test (*e.g.* grab some data from an external source) or to check whether the environment is suitable for the test. An exit status of '2' will make *moseltest* skip the current test and any other non-zero value will be interpreted as a failure.

`save:` name of a file to be saved for future use. The operation is performed after execution of the model when it is successful. The file is copied to the directory identified by `DATADIR`. Several *save* tags may be stated

`setenv:` define an environment variable for the process running the Mosel instance. The syntax of this definition is:

```
[sysname.]varname=value
```

Note that 'value' may contain Unix variable references (*e.g.* `${XPRESS}`) that are expanded before definition. If a 'sysname' is used (see *skip_sys* for the list of system names) the variable will be defined only for the corresponding system. When environment variables are defined, the Mosel instance is systematically restarted and terminated at the end of the run (independently of the tags *restart* and *newinst*).

`sync:` Boolean activating the synchronous mode on the current group of tests. If a test defines this tag, all following tests in the same directory will be run sequentially after the current one on the same worker.

2.3 Status values and test output

`compstat:` expected compilation status (integer). By default *moseltest* expects the compilation of a model source to succeed. This tag will be used to test the compiler when a compilation failure is expected.

`errexpect:` similar as *outexpect* but expressions are searched in the error stream.

`exitcode:` expected exit code (integer). By default a test is successful after the model execution returns the execution exit code 0. This tag specifies an alternative exit code.

`noerrmsg:` Boolean indicating that any message sent to the error stream will cause a test failure (by default the error stream is ignored if execution succeeded).

`outexpect:` an extended regular expression that will be searched in the output stream of the test program after its execution. Several *outexpect* tags may be specified: they are searched in the order of their declaration (*i.e.* after the first expression has been found, the following one is searched in the remaining part of the document, *etc.*). The test will be successful only if all regular expressions have been found.

`runstat:` expected execution status (integer). By default a test is successful after the model execution returns the execution status value 0. This tag specifies an alternative execution status.

2.4 Usage examples

Here are some examples of how to use the *moseltest* tags:

- A test must be run exclusively under Windows 32bit and Linux 64bit: the model file must include the following tag:

```
!*! only_sys windows32 linux64
```

- A test does not work on HPUX:

```
!*! skip_sys hp-ux
```

- A test performs 3 consecutive optimisations. The first optimisation should result in the output "Solution: 123", the second one in "Solution: 456" and the last one "Solution: 789". The following tags make it possible to identify a valid execution:

```
!*! outexpect Solution: 123
!*! outexpect Solution: 456
!*! outexpect Solution: 789
```

- A collection of tests require the package 'toolbox': instead of including the package source in each test, the source file can be put in the test directory named such that it will be processed before the other files (*e.g.* `aaa_mypackage.mos`). This source file must include the following tag:

```
!*! package toolbox
```

  Because of this tag, after compilation the package is copied into the dso directory (under `WKDIR/dso`) as `toolbox.bim` and it becomes available for the following tests

- A test is expected to fail (Mosel status 11 — for runtime error) and the error stream must report error "invalid parameter". This can be achieved with the following tags:

```
!*! runstat 11
!*! errexpect invalid parameter
```

- A test requires a data file stored on a remote host — the model `getit.mos` performs the retrieval; it also depends on module `useful.dso` and package `plusplus.bim`, both sources are included in the test archive. Moreover, evaluation of a successful execution is done by analysing a resulting output file. This verification is operated by model `verif.mos`. The main model of this test must define the following tags (the other files are not tagged):

```
!*! build useful.dso plusplus.bim
!*! runbefore getit.mos
!*! runafter verif.mos
```

- Contents of an .mcf file to run model `mymodel.bim` (located on the `LIBPATH`) setting 2 runtime parameters and checking the produced output (the model displays the parameter values).:

```
!*! model:mymodel
!*! parms:Q=3,G='hello'
!*! outexpect G=hello
!*! outexpect 3
```

- Contents of an .mcf file to run the compiled .bim contained in the specified Insight app archive, the output file `usrinputdump.txt` produced by this run is stored for later use by other tests:

```
!*! model:explapp.zip
!*! save usrinputdump.txt
```

  A second .mcf file for the same Insight app may specify a different parameter configuration by which the app reads in the previously saved file that gets copied into the test working directory via *restore*:

```
!*! model:explapp.zip
!*! restore usrinputdump.txt
!*! parms:MODE=USRRUN
```

- Run a test with Mosel versions 5.2-5.6 only, if *mmxprs* version is at least 3:

```
!*! required:mosel > 5.2, mosel < 5.6
!*! required:mmxprs > 3
```

- A test requires an additional file to be copied into the working directory.

```
!*! copy myfile.dat
```

- A test that defines labels 'needodbc' and 'needmysql':

```
!*! labels needodbc needmysql
```

  can be selected using parameter `'ONLYLABELS=needodbc'` (this setting will select all tests with the label 'needodbc'), stating `'ONLYLABELS=needodbc SKIPLABELS=needmysql'` will exclude this test (it will keep only tests defining 'needodbc' and not 'needmysql').

3 Return value

moseltest returns an exit code of 0 when all tests have been passed. Otherwise, a non-zero value is returned and by default a report is generated including the output and error streams of all tests that have failed.