Mosel testing program

Y. Colombani

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

January 2022

1 Working with *moseltest*

The *moseltest* Mosel program is a general framework for testing Mosel using source Mosel files, C programs or Java programs. 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).

A test can be packaged either as a *single file* (.mos, .c or .java), 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 or .java file) or the main file to run must be named `main.mos`, `main.c` or `main.java`.
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 tests are run in a dedicated directory specified via the model parameter `WKDIR` (default: `workdir`). Running *moseltest* will create and populate the working directory with the following contents (please note

that unless model parameter KEEPDIR is set to 'true' the working directory will be deleted after each completed test run):

- the program to be tested both in source and compiled form (*e.g.* mos & bim file) 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.

1.3 Other configuration settings

Report generation settings: Unless Teamcity mode is enabled (see 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 date to it (depending on setting of REPDATES), 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).

***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.

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.

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

Teamcity mode: If Teamcity mode is enabled (by setting parameter TC to 'true') then no report file is generated, in its place output in Teamcity format is generated that is sent to the output stream. The length of the output sent to Teamcity is limited to the number of characters specified in TCMAXLOG, truncating from the end of the message string if required.

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
*/
```

The following tags are supported:

`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`).

`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*.

`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.

`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.

`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.

`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.

`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.

`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.

`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.

`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.

`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.

`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.

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

`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).

`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.

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

`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*).

2.1 Usage examples

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

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

```
!*! only_sys windows32 aix64
```

- 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 directory `DSODIR` 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
```

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.