Chapter

35 *The TTCN Exerciser*

A TTCN Exerciser is built from the TTCN to C compiler. For more information, see chapter 28, *The TTCN to C Compiler (on UNIX)* and chapter 33, *The TTCN to C Compiler (in Windows)*.

For information about TTCN Exerciser limitations, see "TTCN Exerciser" on page 60 in chapter 2, *Release Notes, in the Release Guide*.

Introduction

Though the GCI interface is generic and may be used for executing tests in an almost arbitrary environment, there is often a significant effort required to get an adaptation up and running. This has prompted the development of a pre-built kernel which can be used to test executable test suites without requiring them to have complete adaptation layers for some platforms.

The TTCN Exerciser contains functionality for simulation of test suites without the requirement of having an SDL system or IUT to test. Furthermore it allows for one or several PCOs to communicate with a real IUT through a subset of the GCI interface. It is also prepared for adaptation with C++ and object oriented implementations under test by providing an “External C interface”.

Functionality Overview

The TTCN Exerciser provides several features that may come in handy for test development, execution and also adaptor production. The main features of the TTCN Exerciser are as follows:

- Simulated PCO communication

  Allows for a PCO not to be connected to an IUT. A message can be input in a PCO input queue at any time.

- Discrete time simulation

  Test the tester with user-defined timing. An explicit “timeout” command is available to allow for the next pending timer to expire at will. This mechanism has a timer resolution down to individual nanoseconds.

- Realtime simulation

  This mode uses real time as provided by the system clock of the host where the tester is executed. Practical limitations to the time resolution yields an accuracy of approximately 1 millisecond. The theoretical limit of the kernel is 1 nanosecond.

- TTCN test case validation

  Runs “random walks” of test cases in order to detect paths where no verdict is assigned, or where some input causes a deadlock, as well

as several other dynamic error conditions. This can be very useful for getting a degree confidence in the correctness and completeness of the test suites.

- Custom PCO definition

  Provides the ability to define custom PCOs for connection to real IUTs while still having access to all the TTCN Exerciser features.

- Source-level debugging of TTCN

  This includes setting breakpoints, highlighting lines in the TTCN Table Editor as it is running, and more.

- Concurrent TTCN implementation

  The kernel internally implements concurrent TTCN, meaning that there is no extra effort in running concurrent test cases.

- Dynamic MSC generation with the MSC generator

  For more information, see chapter 33, *The TTCN to C Compiler (in Windows)* and chapter 28, *The TTCN to C Compiler (on UNIX)*.

- Command scripting and logging

  Providing repeatability of simulated scenarios, as well as shortcuts for doing common operations.

- Dynamic error detection

  The TTCN Exerciser helps detect a number of dynamic error conditions that a static syntax and semantics analyzer will not detect.

Kernel Operation Modes

In order to appreciate the way the TTCN Exerciser operates, it is important to get an insight in the different global modes it may be in. Each of these modes are outlined in this section. The full global kernel state and behavior is a combination of these modes, and to get the desired behavior, it is good to know how to toggle between the modes at will.

Execution Modes

These are the three major modes of operation of the TTCN Exerciser:

- Simulation mode

  This mode has all PCOs simulated, and input is required from the test operator or user-defined scripts. In essence, there is no IUT and all messages that are sent are only indicated in the various test logs. All messages that are received must be specified by the test operator or a test execution script.

- Target execution mode

  This mode has all the PCOs defined by an adaptor writer (implemented in an adaptation effort). This mode provides source-level debugging of TTCN test suites and an option of discrete time simulation.

- Mixed mode

  This mode is available when some of the PCOs are connected to an actual implementation under test. It enables the features of both the simulation mode and the features of the target execution mode.

Timer Modes

These are the two timer modes:

- Realtime

  This mode uses the system clock to provide timing information. This is the normal mode to run in for target testers. In the case a tester is paused or a breakpoint is reached, the timer mode will be toggled to discrete time simulation. You can resume it by setting the timer mode to `realtime`. See "Realtime" on page 1329 for more information. The realtime mode is recognized by all log messages being prefixed with an asterisk (`*`).

- Discrete time simulation

  This mode only increments time when a `timeout` command is received. It is the default mode for simulations. In effect, the mode keeps track of running timers and their expiration time. This mode is automatically toggled to when the execution is stopped or paused, as well as for many of the kernel commands. See "Discrete" on page 1328 for more information. The discrete time simulation mode is recognized by all log messages being prefixed with a hyphen (`-`).

Control Modes

There are two control modes:

- Simulator UI control

  This mode is for using the TTCN-SDL Co-simulator user interface protocol in communications with the operator. The mode appends a set of tags to all messages that are emitted. It is assumed that the tester is started by the TTCN-SDL Co-simulator user interface, but the mode is also useful if logs are to be processed by other tools.

- Command line control

  This mode is for running in a command-line environment. An ETS built with the TTCN Exerciser can be run directly from a command-line, thereby providing means for automated, batch style testing.

PCOs

You can operate PCOs by either implicitly enqueueing messages in the PCO by using the receive command, or by customizing the PCO providing your own definition of the PCO behavior.

Customizing the Behavior of PCOs

The behavior of PCOs can be customized for achieving communications with “real” implementations under test. This is similar to the adaptation process. For more information, see chapter 37, *Adaptation of Generated Code*. The difference is that only the GciSend function and a polling/decoding function need to be implemented. Some familiarity with the GCI interface is required in order to successfully add a custom PCO implementation to the TTCN Exerciser – in particular, the use of the GciValue type should be known. Also, knowledge of the protocol and APIs for the implementation of the PCO is required.

Files

The TTCN Exerciser is interfacing to the code generator output in a manner that is conformant to the GCI interface recommendation. The `adaptor.c` file which is supplied with the TTCN Exerciser interface, is pretty much a plain adaptor that calls GCI equivalent functions in the

TTCN Exerciser. It is possible to edit the `adaptor.c` file while still retaining functionality of the TTCN Exerciser.

Custom PCO Registration

PCOs may be registered in the main function, before the IsmMain function is called. The PCO registration is done by calling the IsmRegisterPCOImplementation function. This functions arguments are described in detail in the `ism.h` file. Conceptually, each registered PCO is polled at a maximum interval supplied when the PCO is registered. The PCO has an associated polling function that should be used to check if a message is completely received on the PCO, and in that case decode it and use the GciReceive function to notify the TTCN runtime behavior of the newly arrived message. For sending, the PCO, registration function also requires a Send function argument. This function should send a message to the implementation under test.

It should be noted that these functions should all be possible to call from multiple threads, since the TTCN Exerciser may use multithreading to implement concurrent TTCN.

Timers

For both the realtime and the discrete time simulation, the implementation of the TTCN Exerciser has 64 bit timers. Time is kept as the difference from the timers expiration time and the last time the current time was updated. In realtime mode, the time is updated in the snapshot function. With discrete time simulation, the time is updated at the timeout command. With the 64 bit implementation and the constraint that the resolution is 1 ns, the consequences are that:

- No period of less than 1 ns can be measured.
- The maximum period that can be measured is more than 2 billion seconds, or some 65 years.

Hopefully, this resolution is sufficient for most applications. There are some aspects of discrete simulation that are well worth a special note:

- Discrete time simulation assumes that no processing time is required for executing TTCN and that there is no delay due to messaging and scheduling overhead in concurrent TTCN. This is not en-

tirely true. Apart from this, it is in many cases sufficient for testing a test script in a non-realtime environment.

- Discrete time simulation also expires timers at the exact right time (from the testers aspect, not in a realtime aspect). In many cases, a test process cannot be scheduled at exactly the time when a timer expires, and consequently there is normally a lag in a real-time tester. This lag should normally be accounted for by the test designer in determining timer durations.

Runtime Timer Errors and Warnings

With the timer implementation, there are also some conditions that may be reported in the conformance log:

- READTIMER of a timer that is not currently running. This condition will make the readtimer call return 0 and also produce a message in the conformance log.
- START of a timer with a ps resolution will result in the timer to get an approximate expiration time. This expiration time is the duration of the timer / 1000 ns. In effect this is too early. If the unit of a timer cannot be determined, it is assumed to be seconds. This condition should normally not appear, but will result in a message in the log and possibly also test case termination.

Test Suite Parameters

The test suite parameters are read at startup of the TTCN Exerciser. There are two locations where they are searched for:

- If the test suite parameters pics/pixit reference field contains the literal substring `.ttp`, the whole fields content will be treated as a file name, naming a file in which the parameter should be searched for. The file format is defined below.
- In a file named `pixit.ttp`.

The file format is defined as:

```
File ::= {Line};
Line ::= Comment | ParameterNameAndValue;
Comment ::= <Non alpha Character> <Any text> | <empty line>;
ParameterNameAndValue ::= ParameterName ISMValueEncoding;
```

The ISMValueEncoding is defined in a separate section. An example would be:

```
#
# This is my parameter file, named pixit.ttp, located in the
# working directory of the ETS with the TTCN Exerciser.
#

TspFoo          3
TspUsePDUPDUA ( INTEGER 1, BOOLEAN TRUE )

# End of this example file.
```

Test Case Validation

The TTCN Exerciser has a rudimentary automatic test validation feature in that it supports running tests with random inputs and timeouts. This feature may be used to attempt to detect missing alternatives and some runtime error conditions without having to manually step through all these alternatives.

Detected conditions include:

- Test incompleteness
- Tests failing to terminate
- Concurrent TTCN dynamic errors:
  - Failure to terminate a PTC
  - Creation of already started PTC
- Tests with paths missing a verdict
- A number of other conditions such as timer errors etc.

The random walk does not provide a guaranteed detection of these conditions, but it will run thousands of test tests in the time it normally takes to run one test.

The random walk has no feature for message generation, and hence requires messages to be imported or defined in the tool. All the messages are stored in a list of eligible messages. This list is referred to as the message list.

The random walk selects random messages from this list, and also timeouts if in a given state, any timers are running. This is repeated until a

verdict is reached, an deadlock is detected, or until a defined maximum depth is reached.

Reports are stored in a report list and can be saved or reviewed.

Also, test statistics are stored while running the random walk. These test statistics can be used to determine how many tests were run in total, and which verdicts were reached.

TTCN Exerciser Commands

The TTCN Exerciser operates through a command line interface (which can be encapsulated by for instance the TTCN-SDL Co-simulator user interface).

Example 207: General syntax of the command input

```
CommandInput::= {CommandLine} EOF
CommandLine::= Comment|Command
Comment::= <Non Alpha Character> <Free Text>
Command::= <Recognized command or '+'> <Parameters> <NewLine>
```

It is permitted to have any amount of spaces or tabs before the command start. In general, the assembler code modes of some editors works quite well for composing command files.

Example 208: A command file

```
;;;
;;; File Name: example
;;; This is an example command file, uses ; for command delim.
;;;


            ;; Initialisation (see command descriptions below)
            cl                      ; Set command line log mode
            loglevel 2      ; Set log level to 2
            nomsc                   ; Disable MSC generation
            nopoll                  ; Disable command polling
            discrete        ; Use discrete time sim

            ;; Now run a test case
            step TestCase1  ; Will create context
            receive Lower ConnectInd { Address 3 }
            run                     ; Will actually run till idle
            timeout                 ; Do a timeout when idle
```

```
;;; End of this file...
```

These are the available commands when a tester built with the TTCN Exerciser is run from the command line or with a batch script. The commands may be abbreviated, as long as the abbreviation yields one unique command name. For instance, the command `cancel` may be abbreviated with `ca` but not with `c`, since that would also match the `cmdlog` command.

The command listing below has the following information:

- Command name

  The formal name of the command, as defined in the ETS/SIMUI protocol document, available on request from Telelogic.

- Aliases

  Alternative command names that may be handy if running from a command line interface. These are often common names for similar operations in debuggers. A hyphen indicates that the command has no aliases apart from any possible abbreviations. Aliases may also be abbreviated, for instance, the alias `rts` for the `realtime` command can be abbreviated `rt`.

- Synopsis

  The command with “symbolic” parameters. If a parameter is surrounded by square brackets “`[ ]`”, it is optional. If a parameter is surrounded by curly brackets “`{ }`”, it indicates that a list may be accepted. If the parameter is surrounded by angle brackets “`< >`”, it means that the name should be substituted by an identifier of an object of an appropriate type.

- Description

  An informal textual description of the command and some of its possible side-effects.

- Example

  An example of the command used in some context. The examples frequently use other commands than the described one in order to explain the use of the command.

General Commands

Help

- Command name: `help`
- Aliases: `-`
- Synopsis: `help`
- Description:

  List the available commands with a brief note on each commands purpose. The general format is:

  `help [SR *]` Display a brief help on commands

  The printed records are:

  `<command-name> [<states> <debug>] <description>`

`<command-name>`	The unique name of the command.
`<states>`	The set of states where the command is applicable: • `S` – Stopped, no test case is running. • `R` – Running, as test case is in progress.
`<debug>`	Denotes if the tester needs to be generated with the line information for the command to be running correctly: • `D` – Requires line-number information. • `*` – Available with or without line-number information.
`<description>`	A brief summary of the command functionality.

The help command will automatically transition the tester to a Paused state if it is in running state.

Example 209 ––

```
-         Command?
```

```
help
```

Quit

- Command name: `quit`
- Aliases: `-`
- Synopsis: `quit`
- Description:

  Terminate the current test case and also terminate the tester program.

Example 210 ----------

```
-       Command?
quit
```

Include

- Command name: `include`
- Aliases: `-`
- Synopsis: `include <filepath>`
  `+filepath`
- Description:

  Enqueue all the commands from the file named by file path. The file path can be relative to the current working directory of the tester, or it may be absolute. Also, the shortcut `+` may be used for simpler access to common scripts. It is recommended but not required that scripts have a file extension of `.ics` (ISM Command Script).

  It is sometimes desirable to use the command `nopoll` in conjunction with script reading, in particular when running test cases from scripts. The `nopoll` command gives a more synchronous behavior.

Example 211 ----------

```
-       Command?
include setup.ics
-       Command?
+setup.ics
```

Test Management Commands

List

- Command name: `list`
- Aliases: `-`
- Synopsis: `list`
- Description:

  List all available test cases. Test cases whose selection expression evaluates to false, get the tag `[N/S]` appended to their name to show that the case is not selected.

Example 212 ----------

```
        Command?
list
MyTest1
MyTest2
MyTest3[N/S]
```

Glist

- Command name: `glist`
- Aliases: `-`
- Synopsis: `glist`
- Description:

  List all available test case groups.

Example 213 ----------

```
-       Command?
glist
MyGroup1
MyGroup2
```

Run

- Command name: `run`
- Aliases: `go`, `start`
- Synopsis: `run {<TestCaseOrGroupNames>}`
- Description:

  Start the execution of the named test cases or groups. The test cases will run until a breakpoint is reached, the test case terminates, or until another command is entered.

  If a test is already in progress, this command will continue the test execution until one of the above conditions are met.

Example 214 ----------

```
-       Command?
run MyTest1
-
-       Line: MyTest1 1
-       Line: MyTest1 2
-       Line: MyTest1 3
-       Line: MyTest1 4
-       At breakpoint MyTest1 5
-       Line: MyTest1 5
-       Command?
run
```

Step

- Command name: `step`
- Aliases: `line`

- Synopsis: `step {<TestCaseOrGroupNames>}`
- Description:

  Start the execution of the named test cases or groups. The test cases will run until a breakpoint is reached, the test case terminates, or until another command is entered.

  As opposed to the run command, the test execution will also stop if a line is matched, or if a snapshot (idle) state is reached. This command requires that the tester has been built with line debug information for correct operation.

  If a test is already in progress, this command will continue the test execution until one of the above conditions are met.

  This command will toggle the timer mode to Discrete, since it is not possible to use real-time and line stepping in conjunction.

Example 215 ––

```
-       Command?
step MyTest1
-
-       Line: MyTest1 1
-       Command?
step
-       Line: MyTest1 2
-       Command?
step
-       Line: MyTest1 3
-       Command?
```

––

Stop

- Command name: `stop`
- Aliases: `pause`, `break`
- Synopsis: `stop`
- Description:

  Pause the execution. This will toggle the time mode to discrete and halt the test case without discarding any context information. This

command requires line debug information to be included when the ETS is built to function correctly.

Example 216 ----------

```
-       Command?
run MyTest1
-       Line:   MyTest1 1
-       Line:   MyTest1 2
stop
-       Command?
step
-       Line:   MyTest1 3
-       Command?
```

Cancel

- Command name: `cancel`
- Aliases: `abort`
- Synopsis: `cancel`
- Description:

  Cancel the current test run and reset the tester. This leaves the IUT in an undefined state. It is not recommended to use this command unless a test has lost contact with the IUT or is incomplete.

Example 217 ----------

```
- run test1
```

Loglevel

- Command name: `loglevel`
- Aliases: `ll`
- Synopsis: `loglevel [0-3]`
- Description:

This command defines how much conformance logging should be performed. The levels are:

0	Only verdicts
1	Level 0 + PTCs, PCOs and TIMERS (default level)
2	Level 1 + matched lines and test trees
3	Level 2 + non-matched lines and defaults

If no level is supplied, the default level will be used.

Example 218 ––

```
-       Command?
loglevel 3
-       Set conformance log level = 3
-       Command?
```

––

Savestats

- Command name: `savestats`
- Aliases: `ss`
- Synopsis: `savestats [<file path>]`
- Description:

  Save a simple tabulated file with a list of what tests have been executed and what their verdicts were. This is useful for tracking test results over time, typically by loading the result file in some type of information management program. If the file path is omitted, the default file path `results.txt` will be used.

Example 219 ––

```
-       Command?
savestats testresults.txt
-       Command?
# Just list the contents of the file
system cat testresults.txt
Test                            Verdict
```

```
MyTest1                          PASS
MyTest2                          FAIL
MyTest1                          INCONCLUSIVE
```

Liststats

- Command name: `liststats`
- Aliases: `ls`
- Synopsis: `liststats`
- Description:

  List the test results of all test cases that have been run since the TTCN Exerciser was started, or since the last clearstats command.

Example 220 ----------

```
-       Command?
liststats
- Listing test statistics:
- MyTest1                        PASS
- MyTest2                        FAIL
- MyTest1                        INCONCLUSIVE
-       Command?
```

Clearstats

- Command name: `clearstats`
- Aliases: `cs`
- Synopsis: `clearstats`
- Description:

  Clear the list of test results

Example 221 ----------

```
-       Command?
clearstats
-       Clearing test statistics.
-       Command?
```

Test Debugging Commands

Most of these test debugging commands require the tester to be built with line debugging information for correct operation.

Breakpoints

- Command Name: `breakpoints`
- Aliases: `bps`
- Synopsis: `breakpoints`
- Description:

  List all currently set breakpoints.

Example 222 ––

```
-       Command?
breakpoints
MyTest1                 5
-       Command?
```

Breakpoint

- Command name: `breakpoint`
- Aliases: `bp`
- Synopsis: `breakpoint [table] [line]`
- Description:

  Set a breakpoint at line in table. If either is omitted, the last matched line or table will be used respectively.

Example 223 ––

```
-       Command?
breakpoints
```

```
-       Command?
step MyTest1
-       Line: MyTest1 1
-       Command?
breakpoint
-       Breakpoint set at MyTest1 1
-       Command?
breakpoint 4
-       Breakpoint set at MyTest1 4
-       Command?
breakpoint MyTest2 2
-       Breakpoint set at MyTest2 2
breakpoints
-       MyTest1 1
-       MyTest1 4
-       MyTest2 2
-       Command?
```

Delete

- Command name: `delete`
- Aliases: -
- Synopsis: `delete [table] [line]`
- Description:

  Delete the breakpoint at table and line. If either or both is omitted, the last matched line will be used.

Example 224

```
-       Command?
breakpoint MyTest1 3
-       Breakpoint set at MyTest1 3
-       Command?
breakpoint MyTest1 5
-       Breakpoint set at MyTest1 5
run
-       Line:   MyTest1 1
-       Line:   MyTest1 2
-       Breakpoint reached: MyTest1 3
-       Line:   MyTest1 3
-       Command?
delete
-       Breakpoint deleted: MyTest1 3
-       Command?
delete MyTest1 5
```

```
-       Breakpoint deleted: MyTest1 5
-       Command?
breakpoints
-       Command?
```

Disable

- Command name: `disable`
- Aliases: -
- Synopsis: `disable`
- Description:

  Temporarily disable all breakpoints.

Example 225 ----------

```
-       Command?
disable
-       Breakpoints disabled
-       Command?
```

Enable

- Command name: `enable`
- Aliases: -
- Synopsis: `enable`
- Description:

  Re-enable breakpoints after a disable command.

Example 226 ----------

```
-       Command?
disable
-       Breakpoints disabled
-       Command?
enable
-       Breakpoints enabled
```

```
-       Command?
```

Timers

- Command name: `timers`
- Aliases: `-`
- Synopsis: `timers`
- Description:

  List all currently running timers. The output format contains the following information columns:

`PTC`	The PTC to which the timer belongs
`Timer Name`	The TTCN name of the timer
`Id`	The GCI timer identifier
`Remaining (s)`	The remaining time to timeout in seconds

Example 227 ----------

```
-       Command?
timers
-       PTC     Timer Name      Id      Remaining (s)
-       MTC     Tms                     210         0.400000000
-       MTC     TWatchDog       211       59.600000000
-       PTC1    Tms                     214         0.400000000
-
-       Command?
```

Pcos

- Command name: `pcos`
- Aliases: `-`
- Synopsis: `pcos`
- Description:

List all PCOs and the contents of their associated queues.

Example 228 ––

```
-       Command?
step MyTest1
-       Line: MyTest1 1
-       Command?
receive LowerPCO ConnectReq { 6 "Peer1" }
-       Command?
receive LowerPCO ConnectReq { 7 "Peer2" }
-       Command?
pcos
-       PCO UpperPCO
-            <empty input queue>
-       PCO LowerPCO
-            1:  ConnectReq { 6 "Peer1" }
-            2:  ConnectReq { 7 "Peer2" }
-
-       Command?
```

––

Ptcs

- Command name: `ptcs`
- Aliases: `-`
- Synopsis: `ptcs`
- Description:

  List all the currently active test components. The output columns include the following information:

`Name`	The name of the test component
`Id`	The GCI id of the test component
`Thr`	The host operating system thread id of the component
`Table`	The table name of the last matched TTCN line
`Line`	The last matched line number

  Also, the currently active test component will be indicated with an arrow (`->`) in the output of this command.

Example 229 ----------

```
-       Command?
ptcs
-       Name            Id      Thr             Table Line
-       MTC             124     0               MyTest1 5
-    -> PTC1            125     13              MyStep1 3
-
-       Command?
```

Gett

- Command name: `gett`
- Aliases: `print`
- Synopsis: `gett [<ptc>] <variablename>`
- Description:

  Print the value of the named variable in the named test components-context. If the component name is omitted, the current component will be used.

  Example 230 ----------

  ```
  -       Command?
  gett MTC TsvFoo
  -       TsvFoo: 4711
  -       Command?
  gett PTC1 TsvFoo
  -       TsvFoo: 1234
  -       Command?
  ```

Test Simulation Commands

Discrete

- Command name: `discrete`
- Aliases: `dts`
- Synopsis: `discrete`

- Description:

  Toggle to discrete time simulation mode. The command has no effect if already in discrete time simulation mode.

  The log prefix will be switched to `'-'`.

Example 231 ----------

```
*       Line:   Table1 3
*       Line:   Table1 4
*       Line:   Table1 5
discrete
*       Discrete time simulation is now used.
-       Command?
```

Realtime

- Command name: `realtime`
- Aliases: `rts`
- Synopsis: `realtime`
- Description:

  Toggle to realtime simulation mode. The command has no effect if already in realtime simulation mode.

  The log prefix will be switched to `'*'`. A running test will immediately resume execution.

Example 232 ----------

```
-       Line:   Table1 1
-       Command?
step
-       Line:   Table1 2
realtime
-       Realtime simulation is now used
*       Line:   Table1 3
*       MTC              SEND    ASP1
*       MTC              START   Tms(200)
*       Line:   Table1 4
*       MTC              TIMEOUT Tms
*       Line:   Table1 5
```

```
*       FINAL VERDICT: PASS
*       Command?
```

Timeout

- Command name: `timeout`
- Aliases: `to`
- Synopsis: `timeout`
- Description:

  Switch to discrete time simulation. Force the shortest pending timer to expire.

Example 233 ----------

```
-       Command?
realtime
*       Command?
run Test1
*       Line:   Test1 1
*       MTC             START TWatchDog(60)
timeout
-       Discrete time simulation is now used
-       Command?
run
-       Line:   Default1 1
-       MTC             TIMEOUT TWatchDog
-       FINAL VERDICT:  FAIL
-       Command?
```

Receive

- Command name: `receive`
- Aliases: `input`
- Synopsis: `receive <PCOId> <ISMValueEncoding>`
- Description:

  Enqueue the message defined by ISMValueEncoding in the PCO input queue connected to the named PCO.

For an alternative form, see "Messageinput" on page 1339.

Example 234 ----------

```
-       Command?
pcos
-       PCO UpperPCO
-           <empty input queue>
-       PCO LowerPCO
-           <empty input queue>
-
-       Command?
receive UpperPCO AcceptCall { 1, TRUE, { 34, 13, 99, '5'B } }
-       Command?
pcos
-       PCO UpperPCO
-           1:  AcceptCall { 1, TRUE, { 34, 13, 99, '5'B } }
-       PCO LowerPCO
-           <empty input queue>
-       Command?
```

MSC Generation Commands

Nomsc

- Command name: `nomsc`
- Aliases: `-`
- Synopsis: `nomsc`
- Description:

  Disable MSC generation.

  The command is only available when the tester is stopped.

  Example 235 ----------

  ```
  -       Command?
  nomsc
  -       MSC generation disabled
  -       Command?
  ```

Decomposed

- Command name: `decomposed`
- Aliases: -
- Synopsis: `decomposed`
- Description:

  Enable MSC generation with decomposed MSCs. For more information, see chapter 33, *The TTCN to C Compiler (in Windows)* and chapter 28, *The TTCN to C Compiler (on UNIX)*.

  The command is only available when the tester is stopped.

Example 236 ----------

```
-       Command?
decomposed
-       Decomposed MSC generation enabled
-       Command?
```

Composed

- Command name: `composed`
- Aliases: -
- Synopsis: `composed`
- Description:

  Enable MSC generation with composed MSCs. For more information, see chapter 33, *The TTCN to C Compiler (in Windows)* and chapter 28, *The TTCN to C Compiler (on UNIX)*.

  The command is only available when the tester is stopped.

Example 237 ----------

```
-       Command?
composed
-       Composed MSC generation enabled
-       Command?
```

Mscsystem

- Command name: `mscsystem`
- Aliases: -
- Synopsis: `mscsystem [<system-name>]`
- Description:

  Define the name of the IUT instance in the composed MSC generation. This is for making it easier to generate MSCs which should later be reused in a system verification with an MSC verification tool such as the SDL Validator, as well as for test re-generation with a tool such as Autolink.

  If the system-name parameter is omitted, a default system name will be used (by default `IUT`).

Example 238

```
-       Command?
mscsystem env_0
-       IUT Name for composed MSC generation defined.
-       Command?
```

Mscprefix

- Command name: `mscprefix`
- Aliases: -
- Synopsis: `mscprefix <path/prefix>`
- Description:

  Set a prefix for file names used when generating MSCs. This prefix is prepended to the file name and can be both a path or a file prefix.

  The default is `log_`.

Example 239 ----------

```
-       Command?
mscprefix /tmp/traces/trace_
-       Path/file prefix defined for MSC generation.
-       Command?
#
#       All test traces will be saved in the named directory
#
```

Test Validation Commands

Clearreports

- Command name: `clearreports`
- Aliases: `cr`
- Synopsis: `clearreports`
- Description:

  Clear the list of reports of dynamic errors encountered so far in test runs.

  Example 240 ----------

  ```
  -       Command?
  clearreports
  -       Reports cleared.
  -       Command?
  ```

Listreports

- Command name: `listreports`
- Aliases: `lr`
- Synopsis: `listreports`
- Description:

List all the reports of dynamic errors encountered so far in test runs. The reports are in the following general format:

```
Report <index>: [<table>] <line>: <description>
```

The `<table>` field is optional. If it is not present, it implies that the report was not detected in the context of a running test component. If the `<line>` field is 0, there is a context available, but the context is not executing.

Also, if no line debug information is available, the table and line will be omitted and 0 respectively.

Example 241 ––

```
-       Command?
listreports
-       Listing reports:
-       Report  1:      TestCase1 7: Missing ?DONE in MTC
-       Report  2:      TestCase2 3: Incomplete test case
-       Command?
```

––

Savereports

- Command name: `savereports`
- Aliases: `sr`
- Synopsis: `savereports [<filename>]`
- Description:

  Save the reports to named file, or to the file `reports.txt` if no file name is supplied. The file format is the same as for the `listreports` command output.

Example 242 ––

```
-       Command?
listreports
-       Listing reports:
-       report  1:      TestCase1 7: Missing ?DONE in MTC
-       report  2:      TestCase2 3: Incomplete test case
-       Command?
```

Messagefile

- Command name: `messagefile`
- Aliases: `mf`
- Synopsis: `messagefile <filename>`
- Description:

  Append the message definitions in the messagefile to the list of input messages that can be used with the `messageinput` command, and also for selection of messages with the `randomwalk` command.

  The syntax of the file named by `filename` is:

```
MessageFile ::= MessageLine | CommentLine;
MessageLine ::= PcoId ISMValueEncoding NL;
CommentLine ::= NonAlphaChar Any NL;
```

  Example message file:

```
# This is an example message file named file1.msg

PCO1 ASP1 { 1 2 FALSE '1234'O }
PCO2 ASP3 { 4 2 TRUE '3123'H }

# end of example file
```

  Also note that messages can be added with the command `messageadd`. The messagefile is mainly intended for message lists generated by other means than with this tool.

Example 243

```
-       Command?
messagefile file1.msg
-       Adding message definitions:
-       PCO1 ASP1 { 1 2 FALSE '1234'O }
-       PCO2 ASP3 { 4 2 TRUE '3123'H }
-       Command?
```

Messageadd

- Command name: `messageadd`
- Aliases: `ma`
- Synopsis: `messageadd <pconame> ISMValueEncoding`
- Description:

  Add the named message on the named PCO to the inputs that can be sent with the `messageinput` command and from which inputs are selected with the random walk.

Example 244 ----------

```
-       Command?
messagelist
-       Listing message definitions
-       1   : PCO1 ASP1 { 1 2 FALSE '1234'O }
-       2   : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       Command?
messageadd PCO1 ASP2 ( FALSE TRUE )
-       Command?
messagelist
-       Listing message definitions
-       1   : PCO1 ASP1 { 1 2 FALSE '1234'O }
-       2   : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       3   : PCO1 ASP2 ( FALSE TRUE )
-       Command?
```

Messagelist

- Command name: `messagelist`
- Aliases: `ml`
- Synopsis: `messagelist`
- Description:

  List all messages defined in the message list.

Example 245 ----------

```
messagelist
-       Listing message definitions
```

```
-       1  : PCO1 ASP1 { 1 2 FALSE '1234'O }
-       2  : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       3  : PCO1 ASP2 ( FALSE TRUE )
-       Command?
```

Messageclear

- Command name: `messageclear`
- Aliases: `mc`
- Synopsis: `messageclear [<index>]`
- Description:

  Clear the message at index `<index>` in the message list. If no index is provided, all messages are cleared. The `messagelist` command prints indices for each message definition.

  Also note that the `messageclear` command may change the ordering and indices of messages at higher indices.

Example 246 ----------

```
-       Command?
messagelist
-       Listing message definitions
-       1  : PCO1 ASP1 { 1 2 FALSE '1234'O }
-       2  : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       3  : PCO1 ASP2 ( FALSE TRUE )
-       Command?
messageclear 1
-       Cleared message definition.
-       Command?
messagelist
-       Listing message definitions
-       1  : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       2  : PCO1 ASP2 ( FALSE TRUE )
-       Command?
messageclear
-       Clearing all message definitions
-       Command?
messagelist
messagelist
-       Listing message definitions
-       Command?
```

Messageinput

- Command name: `messageinput`
- Aliases: `mi`
- Synopsis: `messageinput [<index>]`
- Description:

  Provide the message at index `<index>` in the message list as input to the Tester. This is a short form of the `receive` command, appropriate for frequently used messages, and it may also be used to test contents of message files.

  The message is decoded and added at the end of the named PCO queue.

Example 247 ----------

```
-       Command?
messagelist
-       Listing message definitions
-       1   : PCO2 ASP3 { 4 2 TRUE '3123'H }
-       2   : PCO1 ASP2 ( FALSE TRUE )
-       Command?
messageinput 2
-       Command?
step
-       MTC     RECEIVE         PCO1 ? ASP2
-       MTC     STARTTIMERTimer1
```

Randomwalk

- Command name: `randomwalk`
- Aliases: `rw`
- Synopsis: `randomwalk [<repetitions>] {<testcaselist>}`
- Description:

  Will run the test cases of `<testcaselist>` `<repetitions>` times, providing random inputs and timeouts as they go. Reports are collected while running the test cases. Test logging is disabled, but

can be re-enabled if a breakpoint is reached while doing the random walk.

Example 248 ----------

```
-       Command?

# This is a command for running the test case MyTestCase 100
# times while providing random inputs from the message list
# and also timeouts.

randomwalk 100 MyTestCase

-       Starting random walk...

[some logging of verdicts and selected path]

-       Random walk completed
-       Command?
listreports
-       Listing reports:
-       report  1: MyTestCase 4: READTIMER: Timer not running
-       Command?
```

Maxdepth

- Command name: `maxdepth`
- Aliases: -
- Synopsis: `maxdepth <depth>`
- Description:

  Define how many messages and timeouts a test case may receive, until a test case run not producing a verdict should be abandoned. If the maxdepth is exceeded, a report will be produced.

  Example 249 ----------

  ```
  -       Command?
  maxdepth 10
  -       Defined max depth for random walk
  -       Command?
  ```

Kernel Management Commands

Cl

- Command name: `cl`
- Aliases: -
- Synopsis: `cl`
- Description:

  Turn off the TTCN-SDL Co-simulator user interface tagging. This makes the outputs of the tester more human-readable. By default, this tagging is enabled, so the first command you enter when running from the command line is typically a `cl` command.

Example 250 ————————————————————————————

```
<VERDICT>
-       FINAL VERDICT:PASS
</VERDICT>
<MESSAGE UI::READY>
-       Command?
</MESSAGE UI::READY>
cl
-       Output tagging disabled
-       Command?
list
-       TestCase1
-       TestCase2
-       Command?
```

——

Ui

- Command name: `ui`
- Aliases: -
- Synopsis: `ui`
- Description:

  Enable the TTCN-SDL Co-simulator user interface tagging. This makes the output less readable for a human, but easier to parse by

computer programs (such as the TTCN-SDL Co-simulator user interface).

Example 251 ----------

```
cl
-       Output tagging disabled
-       Command?
list
-       TestCase1
-       TestCase2
-       Command?
ui
<LOG>
-       Using simulator ui output tagging
</LOG>
<MESSAGE UI::READY>
-       Command?
</MESSAGE UI::READY>
```

Poll

- Command name: `poll`
- Aliases: `-`
- Synopsis: `poll`
- Description:

  Enable command polling and dispatch on each matched TTCN line. This slows down the execution somewhat, but makes it possible to interrupt a tester in other states than an idle state. The command requires that the tester was built with line debugging enabled.

Example 252 ----------

```
-       Command?
poll
-       Command?
```

Nopoll

- Command name: `nopoll`
- Aliases: -
- Synopsis: `nopoll`
- Description:

  Disable command polling in other states than stopped or snapshot (idle). This increases the execution speed, and also facilitates script writing by ensuring that a tester is in one of these two known states before it processes its next command.

Example 253 ––

```
-       Command?
nopoll
-       Command?
```

––

Cmdlog

- Command name: `cmdlog`
- Aliases: -
- Synopsis: `cmdlog [<filepath>]`
- Description:

  Start or stop a command log. If the `<filepath>` argument is supplied, the named file will be opened for writing and all commands from the command line or included scripts will be written to the file. If no file name is supplied, the file will be closed and command logging disabled.

Example 254 ––

```
-       Command?
cmdlog  asp1
-       Command?
receive pco1 asp1 { '10'B FALSE '11'B }
-       Command?
run
```

```
-       Command?
cmdlog
-       Command?
+asp1
-       Command?
+asp1
-       Command?
pcos
-       PCO pco1
-           1: asp1 { '10'B FALSE '11'B }
-           2: asp1 { '10'B FALSE '11'B }
-           3: asp1 { '10'B FALSE '11'B }
```

Status

- Command name: `status`
- Aliases: -
- Synopsis: status
- Description:

  Display current TTCN Exerciser status and mode information overview.

Example 255 ----------

```
-       Command?
status
-       Tester status: STOPPED
-       Time Mode:     Discrete Time Simulation
-       Logging level: 2
-       Breakpoints:   Disabled
-       Command Log:   Disabled
-       Logging Mode:  Command Line
-
-       Command?
```

System

- Command name: `system`
- Aliases: -
- Synopsis: `system <commandline>`

- Description:

  Execute the <commandline> using the command line interpreter of the platform. Uses the C stdlib function system(<commandline>). Use this with care for reviewing, deleting and creating files.

Example 256 ----------

```
-       Command?
system mkdir ./msctraces
-       Command?
mscprefix ./msctraces/trace_
-       Command?
composed
-       Command?
run tc1 tc2 tc3 tc4

        [snip]

-       Command?
system ls ./msctraces
trace_tc1.mprtrace_tc2.mprtrace_tc3.mprtrace_tc4.mpr
-       Command?
```

ISM Value Encoding

The ISM Value Encoding defines how values are encoded when sent, or decoded when received. It also defines the format for values when printed or read as test suite parameters. The format is not 100% compatible with the SDL Simulator format, or the MSC generation format, though in most cases it will be able to read those values as well.

The general syntax for the value notation is as follows. Rules named with all capital letters are tokens.

```
ISMValueEncoding ::= OptionalTypeAndValue ;
OptionalTypeAndValue ::= [TYPE] Value ;
TYPE ::= <Name of type defined in TTCN test suite> ;
Value ::= Composite | Atomic | OMIT;
Atomic ::= CSTRING | OSTRING | HSTRING | BSTRING | INTEGER | BOOLEAN ;
Composite ::= LPAR {OptionalTypeAndValue OPTCOMMA } RPAR;
CSTRING  ::= ’"’ .* ’"’ ;
OSTRING  ::= ’’’ ([09afAF][09afAF])* ’’O’ ;
HSTRING  ::= ’’’ [09afAF]* ’’H’ ;
BSTRING  ::= ’’’ [01]* ’’B’ ;
INTEGER  ::= ’-’ [09]* | [09]* ;
BOOLEAN  ::= ’TRUE’ | ’FALSE’ | ’true’ | ’false’ ;
```

```
LPAR      ::= '{' | '(' ;
RPAR      ::= '}' | ')' ;
OPTCOMMA  ::= ',' | ;
OMIT      ::= '-';
```

The CSTRING type is by default IA5String, use TYPE to override with a different character string type. Note also that in TTCN, the character string types are compatible and it may not be necessary to make the distinction.

Examples of this encoding include:

Encoding	Description
`1`	The INTEGER value 1
`FALSE`	The BOOLEAN value FALSE
`MyINTEGER 23`	The INTEGER derivate MyINTEGER value 23
`BIT4 '1001'B`	The BITSTRING derivate BIT4 value '1001'B
`{ 1 , 2 , 3 }`	A composite type with field values 1, 2 and 3
`ASP1 { 711, TRUE }`	An ASP1 with field 1 = 711 and field 2 = TRUE
`ASP2(BIT4 '1001'B PDU1(12))`	An ASP2 with second field being a PDU1