User Guide for TITAN TTCN-3 Test Executor

Jenő Balaskó

Version 1/198 17-CRL 113 200/6, Rev. F, 2019-05-17

Table of Contents

Abstract

This document describes detailed information on using the TITAN TTCN-3 Toolset, creating, compiling and executing test suites.

Contents

Chapter 1. About the Document

1.1. Purpose

The purpose of this document is to provide detailed information on using the TITAN toolset, that is, creating test suites from TTCN-3, ASN.1 modules, and test port files, by modifying a `Makefile` and using `make` to build executables.

1.2. Target Groups

This document is intended for users of the TITAN TTCN–3 Test Toolset. In addition to this document, readers requiring additional information on creating and building test suites or writing test ports are referred to the TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor.

NOTE	Test port writing requires a sound knowledge of C++ programming.

1.3. Typographical Conventions

This document uses the following typographical conventions:

- **Bold** is used to represent graphical user interface (GUI) components such as buttons, menus, menu items, dialog box options, fields and keywords, as well as menu commands. Bold is also used with '+' to represent key combinations. For example, **Ctrl+Click**
- The character '/' is used to denote a menu and sub-menu sequence. For example, **File / Open**.
- `Monospaced` font is used represent system elements such as command and parameter names, program names, path names, URLs, directory names and code examples.
- `Bold monospaced` font is used for commands that must be entered at the Command Line Interface (CLI).

Chapter 2. Overview of TITAN

This Test Executor is an implementation of the TTCN–3 Core Language standard with support of ASN.1. There are limitations to supported TTCN–3 language constructs in the Test Executor. In addition, there are some non-standard extensions to the TTCN–3 language implemented by TITAN. Information on these limitations and extensions and also some clarifications of how the standard has been implemented in TITAN, refer to the TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor.

2.1. Components

The main components are the following:

- The **Compiler**, which translates TTCN–3 and ASN.1 modules [1: Compilation of ASN.1 modules is necessary only if the test suite imports type definitions from ASN.1 modules.] into C++ program code.
- The **Base Library**, written in C++ language, which contains important supplementary functions for the generated code.
- The **Test Port(s)**, which facilitate the communication between the TTCN–3 Test System and the System Under Test (SUT).

The generated C++ modules as well as the Test Ports should be compiled to binary object code and linked together with the Base Library using a traditional C++ compiler.

All parts, except the protocol specific Test Ports, are included in the binary package of the Test Executor. The Test Executor is a protocol and platform independent tool that can be easily adapted to any kind of protocols by writing the appropriate Test Port. The generated C++ code is exactly the same on all platforms, provided that the pre-compiled Base Library that matches the operating system and C++ compiler is used. The Test Port may use operating system calls or external library functions for sending or receiving messages towards System Under Test so it may become platform dependent.

Writing a Test Port is not an easy task for the first time, but the Compiler alleviates it by generating an empty skeleton for each function to be written. This skeleton is also useful for checking the correctness of an existing test suite because the Executable Test Program can be linked with this empty Test Port. In this case the resulting program actually does nothing, but the successful linking means that no modules or functions are missing from the test suite.

This document describes building and running test suites using the command line.

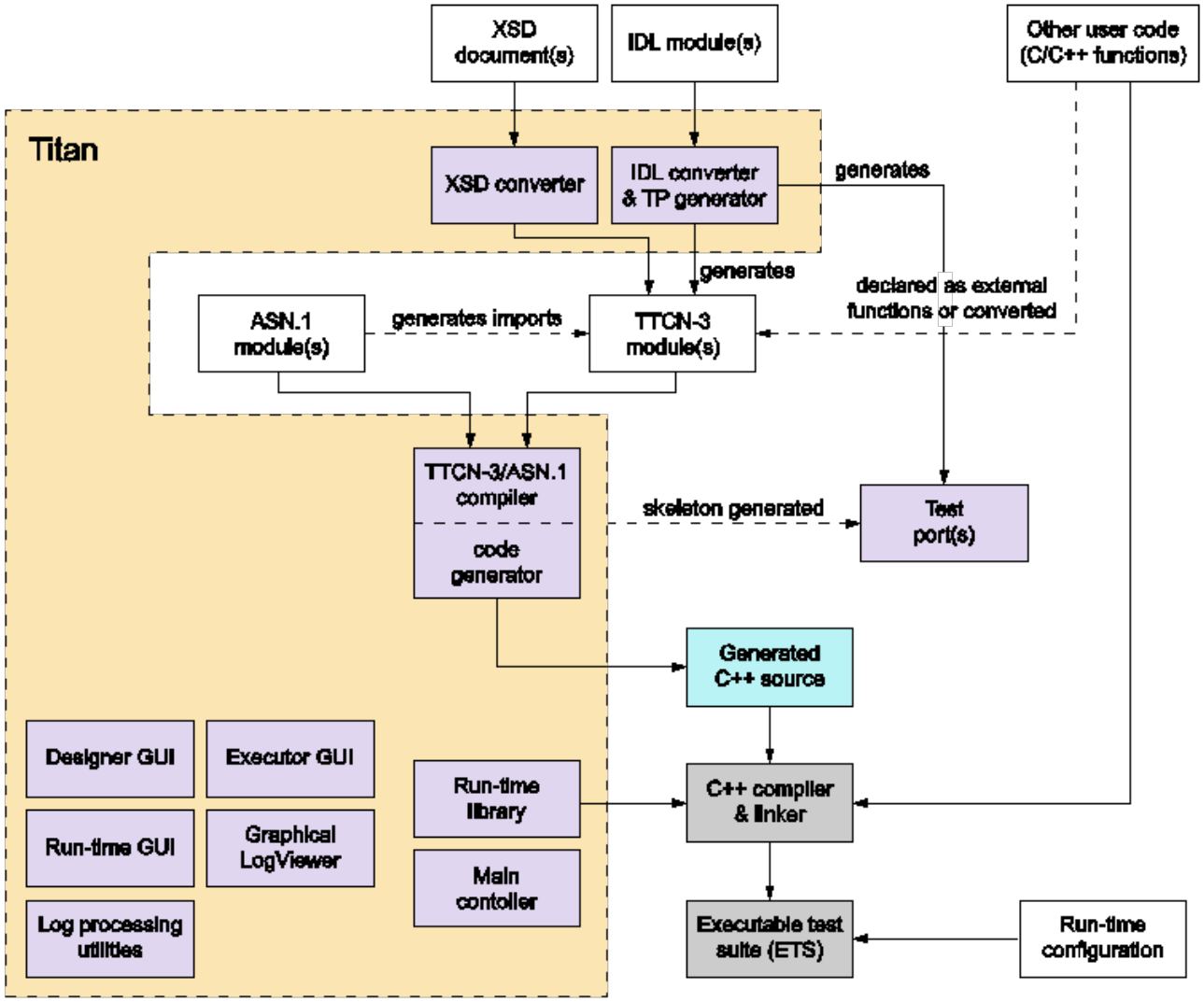

Figure 1. Titan structure

2.2. General Workflow

- Generating and editing a `Makefile`
- Building the executable
- Executing test suites
- Analyzing the execution log files.

2.3. Building Test Suites

Creating a TTCN–3 test suite involves building an executable from the initial modules (TTCN–3, ASN.1 or both) and test port files. The process basically comprises creating and modifying a `Makefile` and using the `make` command to build the executable.

For detailed information, refer to Creating Executable Test Suites from the Command-line.

2.4. Executing Test Suites

After the test suite has been created a suitable configuration file has been built, the executable is

ready to run.

The test executor can operate in single or parallel mode. The single mode—also called non-parallel mode—is thought for TTCN–3 test suites built around a single test component. It is forbidden to create parallel test components in single mode: the test suite is not supposed to contain any `create` operation otherwise the test execution will fail. The parallel mode, on the other hand, offers full-featured test execution including distributed and parallel execution. The goal of introducing the single operating mode was to eliminate redundancies and thereby increase the speed of execution. It is possible to execute non-parallel test suites in parallel mode, but doing so results in unnecessary overhead. The C++ code generated by the compiler is suitable for both execution modes, there are no command line switches to select mode. The only difference is that different Base Libraries must be linked in single and parallel modes.

For detailed information on executing test suites in single or parallel mode, refer to Executing Test Suites.

Chapter 3. Creating Executable Test Suites from the Command-line

This section describes the elementary commands that comprise the build process. The primary audience of this section is the group of users who want to integrate TTCN–3 to a new or an existing build system.

3.1. Using `make`

This section gives an example about how to create a new `Makefile` or modify an existing one manually to make it capable of handling TTCN–3 test suites. For example, if using many external libraries and program modules with TTCN–3, it can be beneficial to write an own `Makefile`.

The generated skeleton is always a good starting point for a custom `Makefile`.

The following lines are mandatory in the `Makefile`:

```
TTCN3_MODULES = MyModule.ttcn

ASN1_MODULES =

GENERATED_SOURCES = MyModule.cc

GENERATED_HEADERS = MyModule.hh

$(GENERATED_SOURCES) $(GENERATED_HEADERS): $(TTCN3_MODULES) $(ASN1_MODULES)

$(TTCN3_DIR)/bin/compiler $(TTCN3_MODULES) $(ASN1_MODULES)
```

`TTCN3_MODULES` and `ASN1_MODULES` contain the names of the TTCN–3 and ASN.1 files, respectively.

The variables `GENERATED_SOURCES` and `GENERATED_HEADERS` store the name of the source and header files that the compiler will generate. This rule calls the compiler with an argument list that contains the name of all TTCN–3 and ASN.1 files. Beginning from version 1.2.pl0 the compiler does *not* duplicate the underscore ("_") characters in output file names, so you may safely use such module and file names that contain this character.

To compile the generated C++ code using `make`, the following rule in the `Makefile` is also needed:

```
.cc.o:

	g++ -c -I$(TTCN3_DIR)/include -Wall $<
```

In this case simply issue the command **`make MyModule.o`** and the two translation steps will be performed automatically in a row.

3.1.1. Rules for Modular Test Suites

The compiler supports modular TTCN–3 test suites as well. Each module is translated to a separate C++ header and source file. These source files can be compiled by the C++ compiler one-by-one separately.

The importing mechanisms work in the following way. For example, two TTCN-3 modules are present in files `A.ttcn` and `B.ttcn`, respectively. Definitions of module A may be used from module B, so the `import from A all`; statement must be added to module B. The modules A and B **must** be translated by the compiler in one step to `A.cc`, `A.hh`, `B.cc` and `B.hh`. During the compilation from TTCN–3 to C++ of module B, the import statement will be translated to `#include "A.hh"`. This statement will be put to the beginning of `B.hh`, so you can refer to any definitions of A in module B. But note that when compiling `B.cc`, `A.hh` must exist and it must be up to date.

Thus, additional rules are needed in the `Makefile`. It is recommended adding them automatically using the utility `makedepend` [2: The makedepend utility is available on all supported platforms. It usually can be found in the X11 development package.]. Run the following command:

```
makedepend -I$TTCN3_DIR/include A.cc B.cc
```

This will add the rules to the end of the `Makefile` and they will be updated upon re-running `makedepend`. For further details please consult the manual page of `makedepend`.

Multiple imports of the same module are handled correctly. For example, if importing all definitions of module C from both modules A and B in the previous example, all three C++ source files will compile correctly.

3.2. Automatically Generated `Makefile`

This section describes the automatically generated `Makefile`, its structure, the supported commands and the possibilities for customization.

3.2.1. `Makefile` Generation

The `Makefile` for a project can be generated using the generator tool `ttcn3_Makefile_gen` [3: Up to version 1.6pl4 Makefile generation was part of the compiler (using the -M option).]. A project usually consists of some TTCN–3 and ASN.1 modules and at least one test port and results in an executable test suite.

`Makefile` generation is performed with the following command syntax:

```
$TTCN3_DIR/bin/ttcn3_Makefilegen [options] <Main module> {Module}* {Test_Port}*
{Other_File}*
```

- `[options]` can be one or more of the options that are listed in the TITAN Programmer's Technical Reference for TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor.
- `<Main module>` is the main TTCN–3 Core Language module. The argument can be either a file

name (with or without path) or a module name. The name of the desired executable will be derived from the name of this module unless the `-e` option is used.

- `{Module}*` are additional TTCN–3 or ASN.1 modules, which are directly or indirectly referenced (imported) from the main module and thus required for building the executable test suite. Each argument should be a file name (with or without path) or a module name.
- `{Test Port}*` specifies names of all test ports or other required C++ program modules. The names can be given with or without suffix.
- `{Other File}*` specifies the names of other files (configuration files, shell scripts, and so on) that are used in this project.

For detailed content of the generated `Makefile`, refer to Makefile Structure.

`Makefile` Generation Algorithm

Before generating the `Makefile` the `Makefile` generator tries to figure out the file name, module type and module name for each argument automatically. It uses some heuristics which yield correct results in most cases, but not always. Typically, the algorithm works perfectly with shell wildcards. For example, if all source files reside in the same directory the following command will generate the right `Makefile`:

```
$TTCN3_DIR/bin/ttcn3_Makefilegen *.ttcn *.asn *.c*
```

The `Makefile` generator looks for an existing file for each argument. It tries the given argument without any suffix, then the following list of suffixes are tried in this order: `.ttcn`, `.ttcn3`, `.3mp`, `.ttcnpp`, `.ttcnin`, `.asn`, `.asn1`, `.cc`, `.c`, `.hh`, `.h`, `.cfg`, `.prj`. Once a file is found, the `Makefile` generator tries to guess its type as described below. If no suitable file is found for a given argument the `Makefile` generator prints an error message and exits.

In the case of TTCN–3 preprocessing (using the `-p` command line argument) the TTCN–3 files with special suffix `.ttcnpp` will be added to the list of TTCN–3 modules which need to be preprocessed before compilation. Files with the `.ttcnin` suffix will be added to the list of TTCN–3 include files (without the `-p` switch these will be added to the other files section of the `Makefile`).

Then the `Makefile` generator tries to classify the file in the following categories based on the contents and/or the suffix:

- TTCN–3 modules (based on contents)
- ASN.1 modules (based on contents)
- TTCN–3 include files (based on suffix, only with `-p`)
- C/C++ source files (based on suffix)
- C/C++ header files (based on suffix)
- other files (the rest)

The `Makefile` generator has two built-in "light" parsers that can decide whether a file is a TTCN–3 or ASN.1 module, respectively. Those parsers read only the first few lines of the input and do not check the syntactical correctness of the modules. They are capable of retrieving the module name as well.

If the Makefile generator ensured that the file is neither a TTCN–3 nor an ASN.1 module then it checks whether the file has .cc, .c, .hh or .h suffix. The content of the file is not examined anymore.

The remaining files (configuration files and so on) will be added to the other files' section of the Makefile. These files do not take part in the build process, but they are added to the archive files created using the Makefile.

After the classification, the Makefile generator filters out the redundant generated C++ files. If a given C/C++ file was found to be generated from one of the given TTCN–3 or ASN.1 modules, a warning is printed and the file will be dropped from the list of C/C++ files. That is, the file will not be added to the list of user source files since it is already a member of the generated sources. This feature is useful if one wants to regenerate the Makefile using the shell wildcard *.cc while the generated files from the previous compilation are still present.

In the next step the algorithm tries to complete the list of C/C++ files by checking the pairs of header and source files. If a C/C++ source file was identified and a header file with the same name exists (only the suffix differs) too, the Makefile generator will add the header file automatically. This step is performed in the reverse direction too: the Makefile generator can find an existing source file based on the header file given to it. Of course a C++ source file can exist without a header file or vice versa.

The Makefile generator continuously checks the uniqueness of files and module names. If the same file was given more than once in the command line the repeated argument is simply ignored and a warning message is displayed. It is not allowed to use two or more different TTCN–3 or ASN.1 files containing modules with the same name because the generated C++ files would clash. For similar reasons the user C/C++ files cannot have identical names even if they are located in different directories.

Finally the Makefile is generated based on the resulting data. If the Makefile generator finds an existing Makefile in its working directory, it will not be overwritten unless the option -f is used.

It is always assumed that the working directory of the generated Makefile will be the same as the current working directory of the Makefile generator even if the Makefile is placed into another directory using the -o switch.

When a path name passed to the Makefile generator contains a directory part the Makefile generator analyzes and canonizes the directory name by resolving relative directory references (such as . or ..) and symbolic links pointing to directories [4: Symbolic links pointing to files will not be resolved.]. If the path name does not contain any directory part or it turns out that the file is located in the current working directory the generated Makefile will refer to the file using a simple file name without any directory. Files located in other directories will be referenced in a uniform way using either absolute or relative path names depending on whether the command line switch -a was specified or not. Thus it is not relevant whether the file was given as relative or absolute path name in the command line.

The Makefile is generated based on the following assumptions:

- Each object and if applicable, shared object file is located in the same directory as the C/C++ source file it is derived from. This allows the use of efficient wildcard rules.

- The TTCN–3 /ASN.1 compiler will place all generated C++ files in the current working directory.

Use of `GNU make`

If option `-g` is used, the resulting `Makefile` will be less redundant as it will use some suffix substitution rules. These rules are supported only by `GNU make`, other versions of the make utility will find such `Makefiles` erroneous.

The more of the file naming conventions below are fulfilled, the more suffix substitution rules can be applied in the generated `Makefile`. If the rules are only partially fulfilled, the `Makefile` will be also correct, but it will be more difficult to maintain. It is recommended to follow these rules especially when starting a new project.

- Unless option `-c` is used, all TTCN–3, ASN.1 and C++ modules should reside in the current working directory. If these files are stored in a different scheme (for example in a hierarchical directory tree) symbolic links can be used to collect all input files into one build directory.
- The suffix should be `.ttcn` for TTCN–3 modules, `.asn` for ASN.1 modules and `.cc` for C/C++ files.
- The file name (without suffix) should be identical to the module name. If the name of the ASN.1 module contains a hyphen, the corresponding file name should contain an underscore character instead. For example, the TTCN–3 module `My_Module` should be stored in `My_Module.ttcn` and the file containing ASN.1 module My-ASN1-Module should be named as `My_ASN1_Module.asn`.
- Each C/C++ module should have a header file with identical name, but with the suffix `.hh`.

Use of Central Storage

Option `-c` can be used to create a `Makefile` that can use pre-compiled files from one or more central directories to save disk space and compilation time. Such `Makefiles` have different layout and more complex build rules.

The central directories should contain those common modules that do not change frequently (type definitions, test ports, external functions, test configurations, and so on). The central directories should be updated and maintained by the project administrators while the individual testers are developing their test cases in their working directory based on the common files. Moreover, it is allowed to create a hierarchy of central directories, that is, to use a directory that takes files from other central directories as a central directory of another project. In such cases the files of all central directories should be passed to the compiler for `Makefile` generation.

In addition to the above mentioned ones the following assumptions are used in these `Makefiles`:

- The compiler will generate C++ files only for those TTCN–3 and ASN.1 modules that are located in the current working directory. The generated C++ files of the remaining TTCN–3 and ASN.1 modules should be located in the same directory as the respective module. If a module is located in a directory other than the current working directory and it does not have pre-compiled files a symbolic link must be created in the current working directory, which should point to the file containing the module.
- Object and if applicable, shared object files will be created only from those C/C++ source files that are located in the current working directory. Object and if applicable, shared object files of

the remaining source files should be located in the same directory as the respective source file.

- The TTCN–3 and ASN.1 modules of central directories should not import definitions from the modules of the current working directory. Importing in the reverse direction is allowed, of course.
- C/C++ files of central directories should not include header files of the current working directory. Local C/C++ files can include headers from other directories.
- The generated C++ files and object and if applicable, shared object files of all central directories must be up-to-date before invoking `make`. Otherwise the build process will fail immediately with an error message [5: If an object and if applicable, a shared object file of a central directory is not up-to-date, but make is invoked it tries to build that file instead of printing an error message. The build will usually fail due to missing access rights. This is a known limitation of this `Makefile` system that cannot be easily solved in a generic way.]. In case of multi-level hierarchy of central directories the re-compilation should be performed in bottom-up order in the central directories.
- All directories must use the same environment, that is, same hardware platform, operating system, version of TTCN–3 Executor and C++ compiler, command line switches, and so on, for building. Otherwise compilation or run-time errors may occur.

Note that when a pre-compiled TTCN–3 or ASN.1 module is taken from a central directory the following three files will be used from the central directory during the build process. Thus it is essential to keep all these files always consistent and up-to-date.

- The module itself when performing the semantic analysis on the local modules importing it.
- The generated C++ header file when compiling the generated C++ files of the importing modules.
- The object and if applicable, the shared object file when linking the executable.

TTCN–3 Preprocessing

Preprocessing of TTCN–3 source code is supported with the use of the option `-p`. The TTCN–3 source files to be preprocessed must have the suffix `.ttcnpp`; these files will be preprocessed with the C preprocessor before being compiled. The compiler will detect all TTCN–3 files, including the ones containing directives for the preprocessor, but only the ones with the suffix `.ttcnpp` will be preprocessed. If any other suffix is used the user has to edit the `Makefile` manually to add the file to the list of files which will be preprocessed. The output of the preprocessing will be an intermediate file with the extension `.ttcn`. Do not use the extension `.ttcn` for any TTCN–3 file that will be preprocessed; also avoid using the same name for different `.ttcn` and `.ttcnpp` files. Files included in `.ttcnpp` files with C preprocessor directive `#include` should have suffix `.ttcnin`.

3.2.2. `Makefile` Structure

This section presents the internal structure of the generated `Makefile`.

For example, the following command will generate a `Makefile` for TTCN–3 test suite "Hello, world!", which can be found in binary distribution:

```
$TTCN3_DIR/bin/ttcn3_`Makefile`gen -gs MyExample.ttcn PCOType.cc MyExample.cfg
```

The `Makefile` generator creates the `Makefile` with the following content:

```
# This Makefile was generated by the Makefile Generator
# of the TTCN-3 Test Executor version 1.6.pl5
# for Adam Delic (edmdeli@ehubuux110)
# on Tue Oct 10 13:53:04 2006

# Copyright Ericsson Telecom AB 2000-2014

# The following make commands are available:
# - make, make all Builds the executable test suite.
# - make archive Archives all source files.
# - make check Checks the semantics of TTCN-3 and ASN.1
# modules.
# - make port Generates port skeletons.
# - make clean Removes all generated files.
# - make compile Translates TTCN-3 and ASN.1 modules to
# {cpp}.
# - make dep Creates/updates dependency list.
# - make objects Builds the object files without linking
# the executable.
# - make tags Creates/updates tags file using ctags.
# WARNING! This Makefile can be used with GNU make only.
# Other versions of make may report syntax errors in it.
#
# Do NOT touch this line...
#
.PHONY: all archive check clean dep objects
#
# Set these variables...
#
# The path of your TTCN-3 Test Executor installation:
# Uncomment this line to override the environment variable.
# TTCN3_DIR =
# Your platform: (SOLARIS, SOLARIS8, LINUX, FREEBSD or WIN32)
PLATFORM = SOLARIS8
# Your {cpp} compiler:
CXX = g++
# Flags for the {cpp} preprocessor (and makedepend as well):
CPPFLAGS = -D$(PLATFORM) -I$(TTCN3_DIR)/include
# Flags for the {cpp} compiler:
CXXFLAGS = -Wall
# Flags for the linker:
LDFLAGS =
# Flags for the TTCN-3 and ASN.1 compiler:
COMPILER_FLAGS = -L
# Execution mode: (either ttcn3 or ttcn3-parallel)
TTCN3_LIB = ttcn3
# The path of your OpenSSL installation:
# If you do not have your own one, leave it unchanged.
```

```
OPENSSL_DIR = $(TTCN3_DIR)
# Directory to store the archived source files:
ARCHIVE_DIR = backup
#
# You may change these variables. Add your files if necessary...
#
# TTCN-3 modules of this project:
TTCN3_MODULES = MyExample.ttcn
# ASN.1 modules of this project:
ASN1_MODULES =
# {cpp} source & header files generated from the TTCN-3 & ASN.1
# modules of this project:
GENERATED_SOURCES = $(TTCN3_MODULES:.ttcn=.cc) $(ASN1_MODULES:.asn=.cc)
GENERATED_HEADERS = $(GENERATED_SOURCES:.cc=.hh)
# C/{cpp} Source & header files of Test Ports, external functions
# and other modules:
USER_SOURCES = PCOType.cc
USER_HEADERS = $(USER_SOURCES:.cc=.hh)
# Object files of this project that are needed for the executable
# test suite:
OBJECTS = $(GENERATED_SOURCES:.cc=.o) $(USER_SOURCES:.cc=.o)
# Other files of the project (Makefile, configuration files, and so on)
# that will be added to the archived source files:
OTHER_FILES = Makefile MyExample.cfg
# The name of the executable test suite:
TARGET = MyExample
#
# Do not modify these unless you know what you are doing...
# Platform specific additional libraries:
#
SOLARIS_LIBS = -lsocket -lnsl
SOLARIS8_LIBS = -lsocket -lnsl
LINUX_LIBS =
FREEBSD_LIBS =
WIN32_LIBS =
#
# Rules for building the executable...
#
all: $(TARGET) ;
objects: $(OBJECTS) ;
$(TARGET): $(OBJECTS)
$(CXX) $(LDFLAGS) -o $@ $^ \
-L$(TTCN3_DIR)/lib -l$(TTCN3_LIB) \
-L$(OPENSSL_DIR)/lib -lcrypto $($(PLATFORM)_LIBS)
.cc.o .c.o:
$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) -o $@ $<
$(GENERATED_SOURCES) $(GENERATED_HEADERS): compile
@if [ ! -f $@ ]; then $(RM) compile; $(MAKE) compile; fi
check: $(TTCN3_MODULES) $(ASN1_MODULES)
$(TTCN3_DIR)/bin/compiler -s $(COMPILER_FLAGS) $^
port: $(TTCN3_MODULES) $(ASN1_MODULES)
```

```
$(TTCN3_DIR)/bin/compiler -t $(COMPILER_FLAGS) $^
compile: $(TTCN3_MODULES) $(ASN1_MODULES)
$(TTCN3_DIR)/bin/compiler $(COMPILER_FLAGS) $^ - $?
touch $@
tags: $(TTCN3_MODULES) $(ASN1_MODULES) \
$(USER_HEADERS) $(USER_SOURCES)
$(TTCN3_DIR)/bin/ctags_ttcn3 --line-directives=yes $^
clean:
-$(RM) $(TARGET) $(OBJECTS) $(GENERATED_HEADERS) \
$(GENERATED_SOURCES) compile \
tags *.log
dep: $(GENERATED_SOURCES) $(USER_SOURCES)
makedepend $(CPPFLAGS) $^
archive:
mkdir -p $(ARCHIVE_DIR)
tar -cvhf - $(TTCN3_MODULES) $(ASN1_MODULES) \
$(USER_HEADERS) $(USER_SOURCES) $(OTHER_FILES) \
| gzip >$(ARCHIVE_DIR)/'basename $(TARGET) .exe'-\
'date '+%y%m%d-%H%M''.tgz
#
# Add your rules here if necessary...
#
```

3.2.3. Editing the Generated `Makefile`

Assume that the TTCN–3 and ASN.1 modules together with the test ports have been written and a `Makefile` skeleton has been generated. The `Makefile` generator recognizes the operating environment and sets up some compiler/linker flags accordingly. The path to the TTCN–3 test executor installation must be set in `TTCN3_DIR` before starting to use `make`. If OpenSSL is installed and proprietary shared libraries will be used, the variable `OPENSSL_DIR` may be changed to point to the directory of the proprietary OpenSSL installation. In the above "Hello, world!" example the user also needs to change the execution mode (variable `TTCN3_LIB`) to non-parallel.

Always perform the following checklist before the first build of the executable test suite from the generated `Makefile`:

- Verify that the variable `TTCN3_DIR` is set to point to the root directory of the TTCN–3 test executor installation. If this variable is automatically set in the login script, this line can be removed from the `Makefile`.
- Ensure that the variable PLATFORM is set to match the test execution platform [6: The test suite must be translated on the same platform on which it will be executed.].
- Verify that the variable `TTCN3_LIB` contains the name of the appropriate Base Library for the chosen operating mode, that is,` ttcn3` for single and `ttcn3-parallel` for parallel execution mode!
- The variable `CXX` should point to the name or full path of the C++ compiler.
- The variables `CPPFLAGS`, `CXXFLAGS` and `LDFLAGS` should contain the extra command line switches to be passed to the C\\++ preprocessor, compiler and linker, respectively [7: For the detailed list and explanation of possible command line switches, refer to the manual page of the used C++

compiler]. For example, profiling or optimization is set here.

- Using the variable `COMPILER_FLAGS` you can pass additional command line options to the TTCN–3 /ASN.1 compiler.
- Ensure that the version of the TTCN–3 /ASN.1 compiler used is identical to the version of Base Library it is linked with. In case of version mismatch the generated C++ source files will not compile and an `#error` notification will be received. This means that changing to another version of TTCN–3 Test Executor, a full re-build of all modules using `make clean` must be performed.
- Make sure to always build test ports from their source distribution. A version mismatch between the object and if applicable, shared object files may cause improper linkage or unpredictable behavior. It is thus contra-indicated to link precompiled test port objects and if applicable, shared objects into your executable (for example taken from a central repository). If the `Makefile` was generated with the option `-p` check also:
- The variable `CPP` should point to the name or full path of the used C preprocessor.
- Command line options for the C preprocessor can be given using the `CPPFLAGS TTCN3` variable.

WARNING	do not confuse it with the `CPPFLAGS` variable, which is used on the generated C++ code.

- Specify additional files which are included (`#include` directive) into `ttcnpp` files with the variable `TTCN3_INCLUDES`. These files will be checked (modification time) at every build to determine if any dependent files need to be recompiled. Any file with extension `.ttcnin` will be added to TTCN3_INCLUDES by the `Makefile` generator.

3.2.4. Available Commands

The generated `Makefile` supports the following:

- **`make all`, `make`**

  Creates or updates the executable test suite. Performs only those steps of compilation that are really necessary, that is, the output of which is outdated.

- **`make archive`**

  Creates a backup copy of all source files and other files in a tar-gzip archive stored in directory set by the variable `ARCHIVE_DIR` [8: The value archive should not be assigned to the variable ARCHIVE_DIR otherwise the make archive command will work incorrectly. Choose other directory name, like backup.]. The command can be applied periodically: to avoid overwriting older versions, a time stamp containing the current date and time is included in the name of the archive file. The output of this command can be attached to trouble reports submitted for the TTCN–3 compiler or other parts of the TTCN–3 toolset.

- **`make check`**

  Checks the syntax and semantics of the TTCN–3 and ASN.1 modules. This command does not create or update any generated files.

- `make clean`

  Removes all generated files (generated C++ files, object and TITAN generated shared object files and the executable) and log files. This command is useful when changing to another version of the test executor or simply for saving disk space.

- `make compile`

  Translates the TTCN–3 and ASN.1 modules to C++. It is useful when the user wants to carry out the compilation of the generated C++ code later. As a result, an empty file named `compile` is created in the working directory. The attributes of this file contain the date and time of the last compilation, which helps the compiler in selective code generation. It is not recommended to change this file manually. The compiler will be invoked only if one or more of the TTCN–3 or ASN.1 modules were modified after that timestamp, otherwise the generated C++ files are up to date.

- `make diag`

  Lists general information about the environment and the build. This information can be useful to fix build problem by the developers or the support team. The output contains:
  - the compiler related information (titan version, build date, C\+ version, license information, see command `*"compiler –v"*`), + - main controller related information (titan version, {cpp} compiler version, build date, license information, see command `*"mctr_cli –v"*`), + - {cpp} compiler information (see command `*"g+ -v"`),
  - library creator info (see command `**"ar –v"***),
  - values of environment variables `$TTCN3_DIR`, `$ OPENSSL_DIR`, `$XML_DIR`, `$PLATFORM`.

- `make dep`

  Obsolete. Creates or updates the dependency list between the C++ header and source files by invoking the utility `makedepend`. This command must be invoked before the first compilation or when the list of modules or test ports has changed. It is also necessary to run `make dep` if an import statement has been added or removed in a module. The command implies `make compile` and after that it modifies the `Makefile` itself. Used only with older `gcc` versions.

- `make objects`

  Creates or updates the object files created from the C++ source files. This command has the same effect as `make all` except that the executable test suite is not linked in the final step.

- `make port`

  Creates Test Port skeleton header and source files for all port types in the input TTCN-3 modules. Existing Test Port files will not be overwritten.

- `make shared_objects` Creates the shared object files from object files, compiled with `-fPIC`. This target is present only when dynamic linking is enabled. For detailed information, refer to the TITAN Programmer's Reference.
- `make run`

  Creates or updates the executable test suite and then runs it. This is only recommended for simple test suites in single mode. Running requires a configuration file; its name by default is

`config.cfg`. This file has to be written by the user.

3.2.5. Building the Executable

Issue the command `make dep` when finished creating and editing the `Makefile`. This command will translate all TTCN–3 and ASN.1 modules to C++ and will find the dependencies between them automatically. The `Makefile` will be modified; many lines will be appended to it.

Finally, issue the `make` command, which will build the executable test suite. If any of the source files (TTCN–3 or ASN.1 modules or test port source files) has been changed, issue the `make` command to get an up-to-date binary.

If TTCN–3 or ASN.1 modules or test ports are need to be added or removed to or from the project, regenerate the `Makefile` skeleton or change the variables `TTCN3_MODULES`, `ASN1_MODULES`, `GENERATED_HEADERS`, `GENERATED_SOURCES`, `OBJECTS` or `SHARED_OBJECTS` accordingly. If a new test port or other C/C++ module should be added, add it to the lines `USER_HEADERS`, `USER_SOURCES` and `OBJECTS` or `SHARED_OBJECTS`.

WARNING	It is recommended to use the `makedepend` utility together with make. This ensures that all dependencies are handled correctly. Therefore, `make dep` command must be issued before the first use of `make` and whenever the module hierarchy (imports) changes! If no `make dep` command is issued then in some cases two `make` commands shall be issued for the successful compilation.

Use the command `make clean` to remove all generated files.

3.2.6. Modifying the Generated `Makefile`

NOTE	this is a deprecated feature; whenever possible, a .tpd (Titan project descriptor) file should be used instead.

When there is a `Makefile` in a project, it should be updated each time a further file is added or removed from the project.

However, some manual modifications were made to the originally created `Makefile` skeleton, regeneration of the `Makefile` will cause the manually performed changes to be lost. To avoid this situation, write a shell script containing the `Makefile` updates, and configure this shell script to be automatically run after each instance of `Makefile` regeneration.

This way, there is no need to perform the same manual updates upon every `Makefile` generation and file addition process.

The shell script example below can be used to automate the modification of the `Makefile` with the updates every time it is regenerated.

Example Shell Script for `Makefile` Modification

```
#!/bin/sh
editcmd='s/CPPFLAGS = -D$(PLATFORM) -I$(TTCN3_DIR)\
/include/CPPFLAGS = -D$(PLATFORM)
-I$(TTCN3_DIR)\/include -I$(ERLANG_DIR)\
/include -I$(OPENSSL_DIR)\/include/g
s/TTCN3_LIB = ttcn3-parallel/TTCN3_LIB = ttcn3/g
s/OPENSSL_DIR = $(TTCN3_DIR)/OPENSSL_DIR = \/mnt\/TTCN\/Tools\
/openssl-0.9.7d/g
s/^ makedepend/ \/mnt\/TTCN\/Tools\/makedepend-R6.6\
/bin\/makedepend/g
/ARCHIVE_DIR = ./ {
a\
a\
# Directory for ERLANG:
a\
ERLANG_DIR = /OTP/LXA_11930_R9C_6/lib/erl_interface-3.4.2
}
s/-lcrypto $($(PLATFORM)_LIBS)/-lcrypto \\/g
/-lcrypto \\/ {
a\
-L$(ERLANG_DIR)/lib -lerl_interface -lei $($(PLATFORM)_LIBS)
}
'
if [ `uname` = SunOS ]
then
case `uname -r` in
5.6) editcmd="$editcmd
s/CXX = g++/CXX = \/usr\/local\/gnu\/bin\/g++/g"
;;
5.7) editcmd="$editcmd
s/CXX = g++/CXX = \/mnt\/TTCN\/Tools\/gcc-3.0.4-sol7\/bin\/g++/g"
;;
5.8) editcmd="$editcmd
s/CXX = g++/CXX = \/usr\/local\/gnu\/gnu28\/gcc3.0.4_shared_sol8\
/bin\/g++/g"
;;
*) echo 'Unsupported Solaris version.'; exit 1
esac
else echo 'This script runs on Solaris only.'; exit 1
fi
sed -e "$editcmd" <$1 >$2
```

3.3. Manual Building

This section contains information useful for the experienced users who are using a build framework other than `make` for TTCN–3 -based testing.

3.3.1. Compiling the Generated C++ Code

If the TTCN–3 test suite was successfully translated to C++, it's a good idea to check if the generated code contains any errors. The simplest way is to compile it using a C++ compiler. Since the generated code refers to the base library, run the following command:

```
g++ -c -I$TTCN3_DIR/include -Wall MyModule.cc
```

In the following, using of an GNU C++ compiler is assumed. If the TTCN–3 /ASN.1 compiler did not report any errors in the input test suite, the generated C++ code must be correct (that is, compile without errors). After certain TTCN–3 warnings (such as unreachable statements) the generated code may trigger similar warnings in the C++ compiler.

The generated code has been tested on various versions of GNU C++ and Sun Workshop C++ compilers. However, the code should work with any standard-compliant C++ compiler since it does not depend on hardware or compiler specific features. If the generated code fails to compile on a supported platform and C++ compiler the situation is considered as a compiler bug and a Trouble Report can be issued [9: The Trouble Report must include the compiler error message(s), all input files and command line switches of the TTCN–3 /ASN.1 compiler, the platform and the exact version of TITAN TTCN–3 Test Executor and the C++ compiler. It is highly appreciated if the user could minimize the input by dropping out irrelevant modules and definitions.].

The switch `-c` tells the GNU C++ compiler to compile only and not to build an executable because, for example, the `main` function is missing from the generated code. The switch `-I` adds the `$TTCN3_DIR/include` directory to the compiler's standard include path. The optional argument, `-Wall`, forces the compiler to report all warnings found in its input. This argument can be used in GCC only.

The result after a successful compilation is an object file named `MyModule.o` and if applicable, a shared object file named `MyModule.so`. If compilation fails, a lot of error messages may be generated. For example, a misspelled type name in an included test port can totally confuse the C++ compiler. That's why it is recommended to analyze the reason of the first error message only.

3.3.2. Linking the Executable

In order to get the executable test suite, the following files must be linked:

- The object and if applicable, shared object files generated from all used TTCN–3 modules.
- The object and if applicable, shared object files generated from all used ASN.1 modules.
- The object and if applicable, shared object files generated from all used test ports and any libraries that are used in the test ports.
- The parallel `ttcn3-parallel` or the non-parallel `ttcn3` version of the TTCN3 Base Library depending on the chosen operating mode. They reside in `$TTCN3_DIR/lib`.
- The shared library of OpenSSL, that is `$TTCN3_DIR/lib/libcrypto.so`.

Assuming only one TTCN–3 module (called `MyModule`) and one test port (called `MyTestPort`), the linking command will be the following for parallel operation mode:

```
g++ -o MyModule MyModule.o MyTestPort.o -L$TTCN3_DIR/lib-lttcn3-parallel -lcrypto
```

The linking command for single operation mode:

```
g++ -o MyModule MyModule.o MyTestPort.o -L$TTCN3_DIR/lib -lttcn3 -lcrypto
```

The name of the executable file will be `MyModule` in both cases.

3.3.3. Dynamic Linking

In order to save disk and memory space, the TTCN–3 Base Library may be dynamically linked to the executable. In this case use the following command in single mode:

```
g++ -o MyModule MyModule.o MyTestPort.o -L$TTCN3_DIR/lib -lttcn3-dynamic -lcrypto
```

In parallel mode use **`-lttcn3-parallel-dynamic`** instead of **`-lttcn3-dynamic`**.

When running the executable, add the directory `$TTCN3_DIR/lib` to the system library path (which is specified in `/etc/ld.so.conf` on most of UNIX systems) or simply add it to the environment variable `LD_LIBRARY_PATH`.

From version 1.8pl2, `ttcn3_Makefilegen` supports the generation of (per module) shared objects. If this option is enabled with the `-l` command line switch, the project's working directory (together with the central storage directories, if applicable) should be added to `LD_LIBRARY_PATH` in addition to `$TTCN3_DIR/lib`. Otherwise, the resulting executable may not run. If moving the executable from one machine to another, all the generated shared object (.so) files should be copied as well. For more information about the `–l` command line switch, please consult the TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor.

Chapter 4. Executing Test Suites

This chapter describes the modalities of test suite execution.

4.1. The Run-time Configuration File

The behavior of the executable test program is described in the run-time configuration file.

Each section of the configuration file begins with a section name within square brackets. Different sections use different syntax, thus the section name determines the possible syntax of the members.

Refer to the TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor for details of the runtime configuration file including descriptions of each of its sections and examples.

4.2. Running Non-parallel Test Suites

If an application is built for single operation mode the resulting executable contains the ETS itself.

It takes a single optional parameter (the name and path of its configuration file) and two optional command line switches related to debugging:

- `-h`

  Automatically halts execution at the beginning, when the first test case's or control part's execution begins, and displays the debugger's user interface (debugging must be activated).

- `-b file` Automatically executes the specified batch file (containing debugger commands) at the beginning of the program's execution.

  The ETS also accepts the command line options `-l` and `-v` with the following semantics:

- `-l`

  Lists the names of all control parts and individually executable test cases of the ETS to standard output. The list is suitable as the `[EXECUTE]` section of a configuration file. Refer to TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor for more details.

- `-p`

  Lists the names of all module parameters of the ETS to standard output. Each module parameter in the list is prefixed with its module's name.

- `-v`

  Prints the tool version, license information and the name, compilation time, checksum and (if available) the version info of the participating modules.

If the ETS contains exactly one module with a control part, then a configuration file need not be specified. In this case, running the ETS with no parameters will execute the control part. If more than one control part is present (or none at all) then the configuration file is mandatory.

The ETS blocks until all test cases are executed as specified in the section `[EXECUTE]` of its configuration file. Console log messages are displayed on the terminal, while the execution log is written into `LogFile`.

ETSes built for single operation mode are unable to act as HCs thus these cannot be executed in the parallel environment. The test suite should be re-linked with the parallel version of Base Library instead if this was the intention (see Editing the Generated Makefile for information on editing the `Makefile`).

4.3. Configuration

The TITAN runtime environment uses configuration files to control execution of the test suites. An ordinary text editor can be used to create and modify configuration files. The configuration file (with the default extension `.cfg`) is a simple text file consisting of the following sections:

- Module parameters

  This section contains the value of each parameter that is defined in the TTCN-3 or ASN.1 modules of the project.

- Logging

  This section indicates logging conditions: the name of the log file, category and component based logging filters or the like.

- Testport parameters

  This section specifies the parameters that are passed to the test ports during the execution of the test suite.

- Define

  This section contains definitions of macros that can be used in other configuration file sections (except Include) for entry of recurring values.

- Include

  Paths to additional configuration files may be listed in this section. The host controller takes into account the values listed in those configuration files, too.

- External commands

  This section contains shell scripts that are called whenever a control part or a test case is started or terminated.

- Execute

  This section indicates which parts of the test suite will be executed. This section is mandatory in single execution mode. Only test cases without parameters, or test cases where every parameter has a default value, can be started from this section.
  Testcases with parameters can be started from the control part.

The following sections are used only in parallel mode:

- Groups

  This section specifies a groups of hosts used in the Components section.

- Components

  This section contains the rules that restrict the location of PTCs.

- Main controller

  This section controls the behavior of the main controller when executing a test suite.

TITAN processes the configuration file sequentially. If a section occurs several times in the configuration file, all sections will be processed without an error message.

Refer to the corresponding chapter of the TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor for details of the runtime configuration file including descriptions of each of its sections and examples.

4.4. Running Parallel Test Suites

The test execution in parallel mode comprises the following steps:

1. Start Main Controller. (See The TTCN-3 Main Controller)
2. Start Host Controllers, that is, the executable test suite, on all participating computers. (See The TTCN–3 Host Controller)
3. Create MTC.
4. Start the control part or a selection of test cases of a TTCN–3 module on MTC.
5. View the verdicts of executed test cases on MC.
6. Terminate MTC after the end of execution.
7. Terminate HCs and MC.
8. Analyze the logs of each test component.

4.4.1. Parallel TTCN–3 Execution Architecture

The components of test environment form two main groups: the Test System and the SUT. As TTCN–3 is used for black box testing, that is, the test suite does not assume anything about the internal structure of the SUT, this section describes the internal structure of Test System only. The Test System consists of one or more test components, whose behaviors are entirely described in a TTCN–3 test suite. The test system has other components for special purposes, listed below.

Each component of the test system runs independently, they are different processes of the operating system. Every component executes one single thread of control. The components can be located on different machines and, of course, there can be more than one component running on the same computer. In the latter case scheduling among them is provided by the scheduler of the

operating system. Regardless of their roles, all test components execute binary code generated from the same C++ source code. Their code consists of three parts: the code generated from the test suite by the TTCN–3 compiler, the Test Ports and the TTCN–3 Base Library.

The components communicate with each other using TCP connections with proprietary protocols and platform independently encoded abstract messages. The components form three groups according to their functionality.

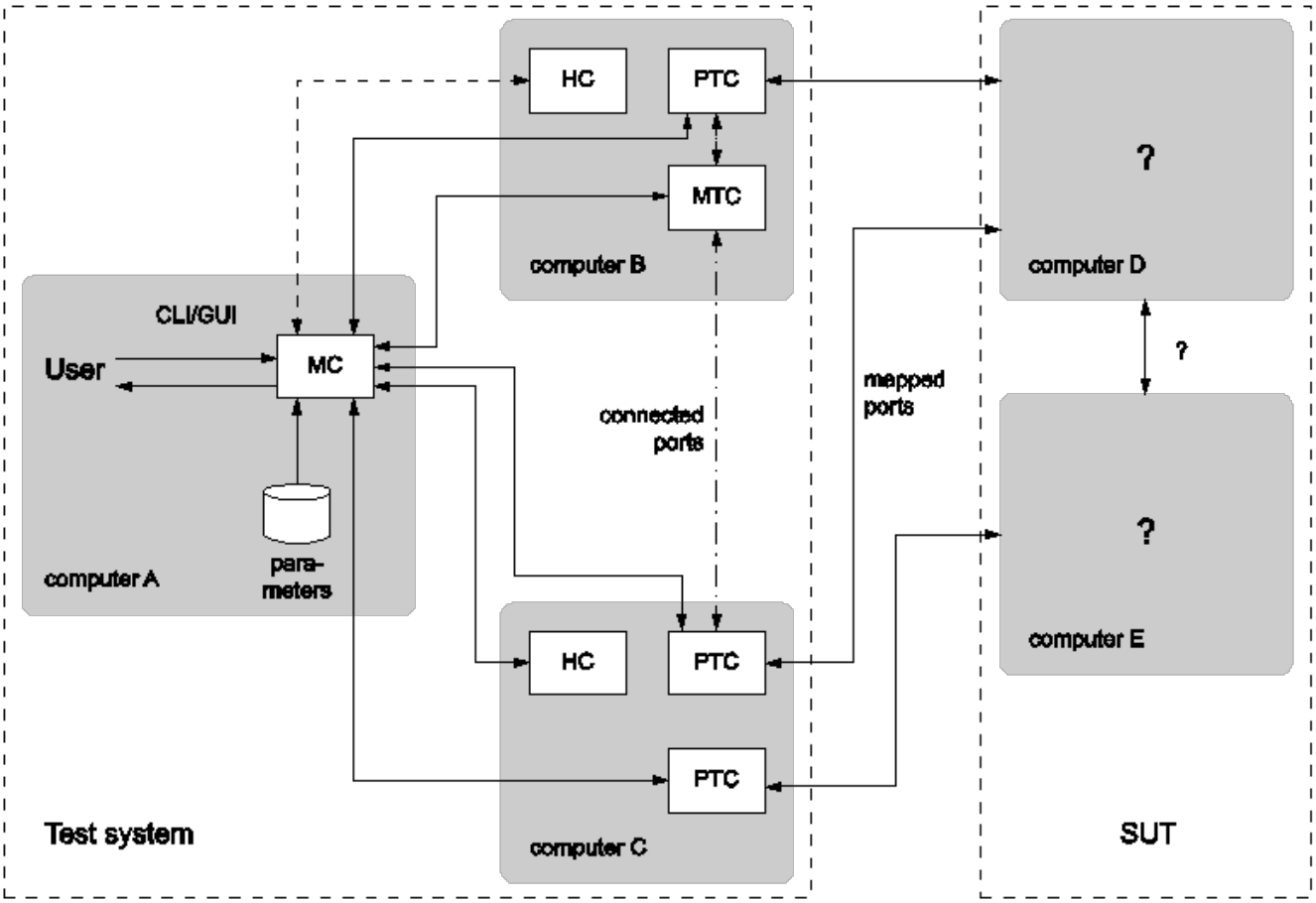

Figure 2. Components of parallel test execution

- **Main Controller (MC)**

  The Main Controller is a stand-alone application delivered with the distribution (`$TTCN3_DIR/bin/mctr`). It is started manually by the user and runs in one instance during the entire test execution. MC provides the user with CLI to the test executor system. It arranges the creation and termination of Main Test Component on user request and the execution of module control part. It shows the user the verdicts of executed test cases. MC has many hidden tasks that can only be performed in a centralized way, for example component reference assignment, verdict collection, and so on. MC maintains a control connection with all other components.

- **Host Controller (HC)**

  Host Controllers are instances (processes) of the executable test program, that is, the translated test suite linked with Test Ports and Base Library. Exactly one HC should be run on each computer that participates in (distributed) TTCN–3 test execution. HCs are started by the user manually on all participating computers. They maintain a connection to MC and if MC wants a

new test component to be created on that host, HC duplicates itself and its child process will act as the new test component.

- **Test Component (TC)**

  Can be either the Main Test Component or a Parallel Test Component.

- **Main Test Component (MTC)**

  The Main Test Component is an instance of the executable test program that is firstly created on a user request. There is exactly one MTC in the Test System. It can execute the control part of a TTCN–3 module if requested by the user. If a test case is executed MTC changes its component type to the type specified in the `runs on` clause of the testcase. Note that MTC is the only one test component that can change its component type. MTC maintains a control connection to MC.

- **Parallel Test Component (PTC)**

  Parallel Test Components are also instances of the same executable test program. TCs execute TTCN–3 functions written by the user in the same way as in non-parallel mode. They are automatically created by HC when requested from the MTC or other PTCs. PTCs also maintain a connection to MC.

4.4.2. The TTCN–3 Main Controller

The binary executable of Main Controller is `$TTCN3_DIR/bin/mctr_cli`. It takes the optional configuration file (The Run-time Configuration File) as its single argument. The variables in the section `[MAIN CONTROLLER]` of the configuration file determine important MC properties, for detailed information refer to the TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor.

The Main Controller has two operation modes: interactive and batch mode. In interactive mode the user can control and monitor the test execution from a CLI. Batch mode is useful for automated and unattended execution of parallel and distributed tests. The actual operation mode depends on the configuration file and is determined at program startup. If the option `NumHCs` is set in the `[MAIN CONTROLLER]` section, the MC starts in batch mode, otherwise interactive mode is selected.

Interactive Mode

After starting MC in interactive mode a welcome screen and command prompt appear.

```
******************************************
* TTCN-3 Test Executor - Main Controller 2 *
* Version: 1.3.pl0 *
******************************************
MC2>
```

The MC command line interface uses the `editline` library which is compatible with the GNU `readline` editing functionality. In addition to its powerful line editing functions it provides command completion, line history and help function.

Command completion is activated using the tabulator key. It presents the list of applicable commands according to the typed prefix. The typing of the command is concluded when a single alternative remains (for example pressing key `c` followed by the tabulator puts the `cmtc` command onto the command line).

The last couple of entered command lines are stored in the history buffer. The implementation is based on GNU `history` library. The buffer elements can be browsed with the cursor keys or an incremental search backward can be performed following a `<CTRL>-r` keystroke and a lot more. History buffer contents are automatically saved and loaded when the `mctr cli` is started or stopped into a file named `.ttcn3 history` located in the home directory. Note that console log messages as well as notifications of HC connection establishments are printed on the MC's screen and may disrupt its contents.

The following commands are accepted by the MC:

- `help [command]` displays the list of available commands or a short use information about the command submitted as parameter.
- `cmtc [hostname]` creates the MTC on the given host. If the optional hostname is omitted, the MTC will be created on the host whose HC has connected first. Once an MTC is created, this command cannot be used before terminating the MTC via emtc.
- `smtc [module name[.control|.testcase name|.*]]` is used to start test execution. smtc has a single optional parameter defining the name of the module or test case to start. The MTC must exist and it must be in idle state when using this command. smtc is a non-blocking command, there is a prompt and it is possible to issue other commands while the test case execution is proceeding. When the module name argument is used (with or without the .control suffix) then smtc starts executing the control part of that module. [10: TTCN–3 assumes to have a single control part within an ETS. Our Test Executor, however, removed this limitation and permits multiple module control parts within the ETS. The smtc command can be used to select between the available control parts, which one needs to be executed. Moreover, it can be specified to execute a number of different control parts, too.] When it is intended to select a single test case for execution, smtc is told using the format `module name.testcase name`. Only those test cases can be executed individually that have no formal parameters, or every formal parameter has a default value. It is also possible to execute all individually startable test cases defined inside a module by specifying the module `name.*` as smtc parameter. In case the optional parameter is omitted, the contents of the `[EXECUTE]` section of the configuration file are run after each other if that section was specified.
- `emtc` terminates MTC. When using this command MTC must be in idle state, that is, it cannot be killed.
- `info` prints statistics and status information of the currently connected HCs and test components.
- `reconf` instructs MC to re-read and re-distribute its configuration file to the connected HCs. This feature is useful when restarting a test campaign involving multiple HCs, because the tester configuration can be altered eliminating the drawback of restarting and reconnecting all elements of the test set-up manually.
- `stop` terminates test execution. The verdict of the actual test case will not be considered in the statistics of the test suite.

- `pause [on|off]` sets whether to interrupt test execution after each test case. For setting the state of the pause function on or off values can be used. If the state of the pause function is on and the actual test case is finished, the execution is stopped until the continue command is issued. If pause is in off state and the actual test case is finished, the execution is continued with the next test case. Using pause without these options it simply prints the state of the pause function.
- `continue` resumes interrupted test execution.
- `log [on|off]` enables/disables console logging. It can be set using on or off. If log is in off state no log messages will be printed to MC's console. Using log without these options it simply prints the state of logging.
- `!` prefix is used to execute command line contents in a subshell.
- `exit` terminates all HCs and MC itself. This command can be used when test execution is not in progress. If MTC still exists it will be terminated gracefully, like with emtc.
- `quit` is an alias to exit to provide backward compatibility.

Batch Mode

If MC is started in batch mode no command prompt is given. In order to monitor the actual state of execution the console messages are printed to the standard output.

In batch mode, the MC performs the following actions sequentially:

- MC waits until the specified number of HCs, that is given in configuration option `NumHCs`, are connected.
- MTC is created on the host of firstly connected HC. Equivalent command: `cmtc`
- The items of the `[EXECUTE]` section are launched sequentially. Equivalent command: `smtc`
- After all items are finished the MTC is terminated. Equivalent command: `emtc`
- The session and all HCs are shut down and MC exits. Equivalent command: `exit`

If the `[EXECUTE]` section of the configuration file is empty or it is missing the MC stops in batch mode immediately with an error message.

If a fatal error is encountered during initialization, for example due to an error in the configuration file, no MTC is created and the session stops immediately. If an error happens within a test case the normal error recovery routines are activated and the execution continues with the next test case.

Performance Hints

NOTE	if performance tests are executed with a large number of test components, MC can be a performance bottleneck in the test executor system. If performance problems occur around the test executor, the first thing that should be checked is the operating environment of MC. Running MC on a dedicated computer with a powerful CPU can help in the most cases.

MC maintains a control TCP connection with all other components (HCs, MTC and PTCs). Each of these connections use an open file descriptor, which is a limited resource in the operating system. If many test components should be run simultaneously, this limitation can be a bottleneck. However,

the number of open files per process can be increased up to a so called hard limit (for example 1024 on Solaris and unlimited [11: The total number of open files can also be a bottleneck on Linux kernel, which can be changed through the /proc file system.] on Linux). The limit can be increased by a built-in shell command [12: Called limit on tcsh and ulimit on bash. For more details please consult the manual page of the used shell.], of course, before starting MC. On the other hand, the license key also limits the number of simultaneously active PTCs, which is considered in MC when processing TTCN–3 create operations.

Displaying ASCII Art on Startup

The command line main controller displays an ASCII art file that is located in the `$TTCN3_DIR/etc/asciiart` directory. There can be any number of ASCII art text files in that directory, a random file will be chosen from those. The file name can contain special filtering instructions, if such instructions are detected in the file name then the file is grouped into the special files group, all other files are in the normal group. If there is at least one file in the special group that was not filtered out by the condition(s) given in the file name then the file to be displayed will be chosen randomly from the list of special files. If there are no such special files or all of these were filtered out by their filtering instructions then a normal file will be displayed. The filtering instructions in the file name are separated by dots, one instruction consists of a name and a value which are separated by a dash. If the value is of numerical type then it can be a single number or an interval, an interval consists of 2 numbers separated by an underscore. Currently the following filtering condition name and value pairs can be used:

Filter condition name	Value, type of value	Example
user	User name, string	user-edmdeli
weekday	Number/interval, 1-7	weekday-6_7
day	Number/interval, 1-31	day-1
month	Number/interval, 1-12	month-12
year	Number/interval	year-2013
hour	Number/interval, 0-23	hour-18_23
minute	Number/interval, 0-59	minute-30
second	Number/interval, 0-61	second-0_30

Example file names:
`xmasparty.month-12.day-24_26.txt`
`weekendwork.weekday-6_7.txt`

Displaying ASCII art can be prevented by deleting all files from the directory. Adding some filtering conditions can be done by renaming the file according to the above described naming rules.

4.4.3. The TTCN–3 Host Controller

The ETS built for parallel operation mode will act as Host Controller. After starting up it establishes a TCP connection to MC (which must be started prior to HC) and waits for requests. The executable takes two mandatory arguments, the host name or IP address and the TCP port number that MC listens on [13: If MC and HC runs on the same computer and you run Host Controllers on other

computers as well, never use localhost or 127.0.0.1 as host name argument to HC. The IP address that the HC's connection comes from may be transferred by MC to TCs running on other hosts. Giving out the local IP address may result in incorrect behavior.].

The optional command line switch `-s` can be used to specify the source address of control connections towards MC. Either an IP address or a DNS name can be given after the switch. Only such IP address is accepted that is assigned to one of the local network interfaces. This option can be useful on multi-homed hosts, that is, computers with more than one network interfaces, in order to route all traffic of control connections to a separate network path to avoid disturbances in the communication with SUT. If the option is omitted the local IP address is chosen automatically based on MC's IP address and the kernel routing table. The test components, child processes of HC, will use the same local IP address for their connections as the HC independent if it was set manually or automatically.

The command line synopsis for HC is the following:

```
<executable_program_name> [-s <local_address>] <MC_host> <MC_port>
```

NOTE	In earlier versions, the HCs accepted an optional third command line argument specifying the configuration file name. From version 1.3 (MC version 2), the MC distributes configuration data to all participating HCs. Consequently, the configuration file became a command line argument of the MC.

The ETS linked in parallel mode accepts the command line switches `-l` and `-v` like in single mode (see Running Non-parallel Test Suites). If the test execution is performed in a distributed environment and file synchronization between computers is not automatic (for example you use FTP instead of a shared NFS directory), it is useful to check the module checksums and versions with flag `-v` on each computer before starting the HCs.

From version 1.3.pl0 the MC checks the version of each connected HC automatically in order to ensure the consistency of the distributed test system. If the ETSes used in the same test campaign contain different TTCN–3 modules or different versions of the same TTCN–3 modules the HC connections, except the firstly connected one, will be refused by the MC.

4.4.4. Logging in Parallel Mode

During test execution all test components create separate log files. Each log file has the same format as presented in non-parallel mode. Logging into the same, NFS shared directory makes the log analysis easier.

The name of log files can be explicitly set in the configuration file using a metacharacter substitution mechanism. If the file names are not set, the backward compatible default naming convention is used. It is important to ensure that every component has its own unique log file name. Refer to the TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor for more details.

In parallel mode the log messages sent to the console are transmitted through the network and printed on the user interface of MC in normal cases. Thus it is an unwise thing to log all messages to

the console without filtering when the test suite is used for load generation. If the control connection from a TC or HC to MC is broken due to any error, the console log messages are written to the standard error of the ETS locally.

4.4.5. Automation of Testing in Parallel Mode

The starting procedure of TTCN–3 tests in parallel mode can be a tiring task if it has to be repeated the tests several times. We have developed a small script that can do this work for you. It is based on the `expect` command, which is an extension of the TCL scripting language. The script is called `ttcn3_start` and is located in `$TTCN3_DIR/bin`. In order to use it a working `expect` interpreter must be in the `$PATH`.

The script itself is very simple, it takes one mandatory and one or more optional arguments. The first mandatory argument is the name of the ETS that is launched. The second argument can be the name of the configuration file that will be passed to MC during execution. If this argument is omitted or the second argument does not resemble to a file name, the script will look for file `<ETS name>.cfg` in its current working directory. If such file exists, it will be used as configuration file. Otherwise MC will be launched without configuration file.

Additionally, the IP address of the interface used for communication between the MC and the ETS can be specified. The syntax is `–ip` followed by the IP address in dotted decimal format, for example 192.168.0.1. If not specified explicitly, the address defaults to the IP address of the local machine.

The rest of arguments are the list of test cases to be executed in format `<modulename>.<testcase name>`. They are passed to MC command `smtc` sequentially, see The TTCN–3 Main Controller for details. If these arguments are missing and a configuration file is present the items of section `[EXECUTE]` will be executed, that is, `smtc` will be called without arguments. If neither configuration file nor test cases are specified the control part of the main TTCN–3 module, that is, the module that has the same name as the ETS, is executed.

The script works the following way: first it launches the MC. If the environment variable `TTCN3_DIR` is set the MC is started from directory `$TTCN3_DIR/bin` (to find the right one multiple versions are present), otherwise the command `mctr_cli` is invoked using your search path. If the configuration file is present it is passed to MC as a command line argument. After that `ttcn3_start` launches the ETS, that is, the HC, locally with the appropriate arguments. That is, the script guesses the host name and extracts the TCP port number from the output of MC automatically. Then the script issues the `cmtc` and the appropriate `smtc` commands in the MC command prompt and waits until test execution is finished. Finally it terminates the programs by issuing `emtc` and `quit`. It also takes care of MC's answers and issues the commands in the right state.

The messages coming from the standard output or standard error of MC, HC and the test components are continuously displayed in the output of `ttcn3_start`.

Note that this script does not support distributed test execution when more than one HC has to be started.

Examples for the invocation of `ttcn3_start`:

```
$ ttcn3_start Main_Control
$ ttcn3_start Main_Control multi.cfg
$ ttcn3_start Main_Control –ip 10.10.10.10 multi.cfg
$ ttcn3_start Main_Control SNMP_Testcases.tc_110 SNMP_Testcases.tc_113
 SNMP_Testcases.tc_114
$ ttcn3_start Main_Control multi.cfg SNMP_Testcases.tc_110 _Testcases.tc_113
SNMP_Testcases.tc_114
```

The script returns different exit codes which can be used by user written software which invokes it. In case of success the return code is 0, in error cases the return codes are the following:

Return code	Error description
1	The expect tool was not found.
2	Parameters are missing.
3	Cannot find the given executable.
4	The script cannot be used when MC is run in batch mode.
5	The MC has terminated unexpectedly.
6	The given executable is not a TTCN-3 executable in parallel mode.
7	The executable could not connect to the MC.
8	The MTC cannot be created.
9	The MTC cannot be created on an unknown host.
10	The MTC terminated unexpectedly.

4.5. Strange Behavior of the Executable

If modular test suites are executed, sometimes the executable test program can do strange things, for example, the execution terminates without any reason or the send functions of the Test Port is not called, and so on. This is because out-of-date C++ header files are used for translating the C++ modules, that is, there is a wrong `Makefile`.

This may happen when the Test Port files are renamed, so the compiler regenerates them. Thus the C++ source files generated by the compiler see an empty Test Port header file, but the fully functional Test Port object file is linked to the executable. In this case, the linking will be successful, but during the execution strange things can happen. The reason behind this phenomenon is that the modules consider the raw binary structure of the same C++ class different, for example they fetch the virtual function pointer from a wrong place.

Avoid these situations and re-compile all C++ files before reporting such bugs, and the use of `makedepend` utility is strongly recommended.

Chapter 5. Log Processing

The logs generated by the test executor, although they are ASCII text files, are perfect for machine processing, but not for analyzing by humans. To make these log files more readable log formatting tools are provided. All of these programs require the same license feature, `LOGFORMAT`. The programs are designed so that they can be used either individually or bundled together with UNIX pipelines.

- `Logmerge` is useful for test suites that are run in parallel mode. It can merge the logs of different PTC into one single file based on the timestamps.
- `Logfilter` can be used for post filtering large log files based on the kind of logged events. It can be specified to keep or remove the event type(s).
- `Logformat` breaks the sent and received data structures into lines and indents the fields according to their hierarchy. Moreover, if the test suite was executed in single mode, the log formatter splits the logs of each test case into separate files.
- `Repgen` can present not only the formatted log files but the description and TTCN–3 source code of test cases as well as the output of other network monitor programs, like `tcpdump`, in HTML format. The test results can be easily viewed by any JavaScript capable Web browser.

5.1. The `logmerge` Utility

The `logmerge` utility, which can be found in `$TTCN3_DIR/bin`, merges all files given in the command argument into a single output file. The output of logmerge is sorted based on the timestamps found in the log files.

The command line syntax is:

```
ttcn3_logmerge [ -o outfile ] [ file.log ] ⋯
```

or

```
ttcn3_logmerge -v
```

Available command line switches are:

- **`-o outfile`**

  Merges all input log files into `outfile`. If the `outfile` exists its contents will be overwritten. This switch is optional, if it is omitted, merged logs will be printed to standard output.

- **`-v`**

  Prints `version` and license key information and exits. Each line of the input files should contain an event in the following format:
  `<timestamp> <rest of the event>`

Merging log files with different types of timestamps, for example with timestamp format `Time` and

`DateTime`, results in warning message(s), and only files with same format are merged. Merging log files with timestamp format `Seconds` is not. If a file contains one or more timestamp(s) that is in wrong order, the resulting order will be incorrect too. In this case a warning message will be printed to the standard error.

The output of the utility is the following:

```
<timestamp> <component id> <rest of the event>
```

where `<component id>` is taken from the name of the respective input file. If the name of the input file is not in the format `<ets name>.<host>-<component id>.log`, the whole input file name will be used as `<component id>`. Events spreading over multiple lines are also handled properly.

5.2. The `logfilter` Utility

The `logfilter` utility, which can be found in `$TTCN3_DIR/bin`, filters the input log file given in the command line argument based on the event types in the file, and filter parameters given in the program argument. The output is then written to an output file if specified, or to the standard output. The program is useful only if the variable `LogEventTypes` is set to `yes` in section `[LOGGING]` of the configuration file.

The command line syntax is:

```
ttcn3_logfilter [ -o outfile ] {eventtype(+|-)}[input.log]
```

or

```
ttcn3_logfilter -v | -h | -l
```

Available command line switches are:

- `-h`

  Prints `help` on using the utility.

- `-l`

  Prints the `list` of supported event types.

- **`-o outfile`**

  Puts its output into `outfile`. If the `outfile` exists, its contents will be overwritten. This switch is optional, if it is omitted, the output will be printed to standard output.

- `-v`

  Prints `version` and license key information and exits.

The utility can handle one file at a time, giving more input files results an error. If no input file is given, it reads the log from standard input. `Logfilter` can be efficiently used as the middle stage of a pipeline, combined with `logmerge` and `logformat`.

Event types to be included or excluded should be given without the `TTCN` prefix, that is, as they appear in the log file. Undefined event type(s), that are not listed in the TITAN Programmer's Technical Reference for TITAN TTCN-3 Test Executor, specified as filter parameters will cause warning message(s), but will not cause the utility to quit. If there are parameters specified both to include and to exclude one or more event types, the program will quit with an error message, because in this case it is not well defined how to handle other event types. All possible error and warning messages will be printed to standard error.

5.3. The `logformat` Utility

The `logformat` utility, which can be found in `$TTCN3_DIR/bin`, reads the unformatted log file generated by test executor from its standard input. It can split up the log into several files based on the lines that are automatically logged at the beginning or end of each test case. Furthermore, `logformat` formats the sent and received messages in the log file. The structured values are indented and each field is put into a new line according to the braces and commas.

The command line syntax is:

```
ttcn3 logformat [ -i n ] [ -o outfile ] [ -s ] [ -n ][ file.log ] ⋯
```

or

```
ttcn3 logformat -v
```

The switches denoted by square brackets are optional. The following command line options are available (listed in alphabetical order):

- **`-i n`**

  Sets the depth of each indentation level to `n` characters. The default value is `4`. If the sent or received PDU is too complex and has too deeply nested fields, this number can be decreased to get more readable output.

- **`-o outfile`**

  Places the output into file `outfile`. If the `-s` flag is also set, only those parts of the log files will be written into this file that were logged outside the test cases, that is, in control part or on PTCs. If this option is omitted, the formatted log will be printed to standard output.

- **`-s`**

  If this option is set, the entries that were recorded during the execution of a particular test case will be saved in a separate file in `logformat`'s working directory. The name of this file will be identical to the name of the test case. If the same test case is executed several times after each

other, the results of repeated test runs will be collected after each other. If the output file contained some data before `logformat` was started, for example the results of previous test run, the output file will be emptied and the old logs will be destroyed.
`logformat` recognizes any types of timestamps that can be set in the `[LOGGING]` section of the configuration file.

WARNING	This option is useless when formatting the log files of PTCs, because these logs do not contain the name of the testcase the PTC belongs to.

- `-n`

  If this option is set, newline and tab control characters are not modified, they are printed as \n and \t.

- `-v`

  Prints `version` and license key information and exits.

`logformat` formats all files that are given as arguments and concatenates them after each other. If no files are given, it reads the log from standard input.

5.4. The HTML Report Generator

The HTML report generator called `repgen` can be found in `$TTCN3_DIR/bin`. The program requires one command line argument that contains the name of its configuration file. The behavior of `repgen` can be configured only through this file. If the switch `-h` is given instead of the name of the configuration file, `repgen` prints a sample configuration file to its standard output.

The configuration file of `repgen` is a simple text file which contains a sequence of directives. Its usual suffix is `.ts`. Each directive starts with a special keyword beginning with a hash mark (#) character. The first part of configuration file should contain global settings, the description of test cases can be added after that.

The following table summarizes all global settings:

Keyword	Meaning
`#Title`	The name of the ETS. This string will be used as title in the resulting HTML pages.
`#Tab length`	The report generator replaces all tabulator characters with spaces in the TTCN–3 test cases and log files. This parameter sets how many spaces a single tab character should be replaced with. The default value is `8`.
`#Column width`	The report generator breaks the long lines of the ATS and the external monitor logs. The resulting lines in HTML output will not be longer than this limit. The default value is `80` characters.

Keyword	Meaning
`#TTCN-3 code`	The name of the directory that contains the TTCN–3 source files of the test suite. All files will be searched in this directory whose name ends with `.ttcn`. `repgen` collects the source code of test cases that are listed in the remainder of this configuration file. The referenced functions, templates and other definitions are not collected. An absolute or a relative path may be entered, the starting point is always the `repgen`'s working directory, for this and the following three parameters. The same directory may be used for many purposes because the file names do not clash.
`#TTCN-3 log`	The name of the directory that contains the log file(s) of the test executor. The report generator splits and formats the log files using the log formatter `logformat`. All files will be formatted in this directory whose name ends with .log. If the log of one test case can be found more than once in the log file(s), for example, because of repeated test execution, the resulting HTML page will contain the log of one execution. The others will be lost.
`#Other log`	The name of the directory that contains the log files of the external monitor programs, for example `tcpdump`. Each file should contain the messages (network packets) recorded during the execution of one test case. The log files in this directory must be named as `<testcase name>.dump` where `<testcase name>` stands for the name of the corresponding test case. All files must be in ASCII format. `logformat` will simply copy them into the destination directory and will not change their content.
`#Destination`	The name of the destination directory where the files of the resulting HTML report should be stored by `repgen`. The starting page will be `<title>-report.html` in this directory and the other files will be stored under sub-directory `<title>-report`, where `<title>` stands for the string set as the value of parameter `#Title`. Note that each space and dash in this name will be replaced by an underscore character.

After the global settings, the name and description of all test cases after each other (in arbitrary order) can be listed.

NOTE `repgen` processes the source code and logs only for those test cases that are listed in the configuration file. The TTCN–3 code and logs of other test cases will be silently ignored. A test case can be specified using the following keywords:

Keyword	Meaning
`#Testcase`	The name of the test case. It must be the same as in the TTCN–3 code.
`#Purpose`	A short summary of the test case describing in one sentence what it does. It must not be longer than one line. These short descriptions will be listed on the HTML page that lists the results of all test cases in one table.

Keyword	Meaning
`#Description`	This section should contain the detailed description of the test case. It may continue through several lines, until the next `#Testcase` directive. Figures or message sequence charts can be drawn using ASCII characters, but images cannot be embedded.

For browsing the HTML reports the only thing needed is to open the starting page, the file `<title>-report.html` in the destination directory, with a JavaScript capable web browser. The reports should work with any versions of Netscape and Microsoft Internet Explorer on any platforms. The reports can be viewed locally or remotely using any web server.

The starting page consists of two list boxes and four buttons (in addition to the title and the Ericsson logo). The test case can be selected in the left list box. After selecting a test case the available descriptions and logs will be shown on the right list box. The following items can be selected:

- **Detailed description**
- **TTCN–3 code**
- **TTCN–3 executor's log**
- **Other type of log**

If one or more items for the test case are missing from input files, the missing option will not be shown. Select or unselect the available descriptions and logs one-by-one independently by clicking on them.

After selecting a test case and its items the "Show" button at bottom should be pressed to view the selected logs and descriptions. A new browser window will be opened for each test case and the selected items will be shown in vertically split frames. The text in each frame can be scrolled independently. Of course, the `logformat` tool is unable to figure out the relation between the TTCN–3 source code and the produced log events.

In the root window, it is possible to walk through the available test cases step-by-step using the buttons **Previous** and **Next**. The button **Summary** will bring up another window that lists all test cases, their short descriptions and verdicts in a single table to get a quick overview about the test results.

Example: *In the following is an example configuration file of* `logformat`*. We included the descriptions of the first three test cases only.*

```
#Title ROHC
#Tab length 8
#Column width 80
#TTCN-3 code /home/ethpkr/ROHC
#TTCN-3 log /home/ethpkr/ROHC/log
#Other log ./
#Destination ./
#Testcase CTC01
#Purpose Mode transition from Unidirectional to Optimistic mode.
#Description
Comp->IRs, Comp->IR_DYNs, Comp->UO-0s, Decomp->Feedback(mode
transition u->o), Comp->UO-0s
#Testcase CTC02
#Purpose Feedback processing in Unidirectional mode.
#Description
Testing the compressor's feedback processing capabilities in U
mode.
#Testcase CTC03
#Purpose Operation in Optimistic mode (NACK).
#Description
Testing the compressor's operation in Bidirectional Optimistic
mode. Preamble: taking the compressor to SO state and O mode.
After that a NACK is sent as an answer to a received compressed
packet. The answer for that should be an IR with dynamic chain
or UOR-2 or an IR-DYN packet.
[...]
```

NOTE `repgen` was designed to present the results of non-parallel test cases. In case of parallel test execution, the logs of PTCs cannot be browsed, only the MTC log.

WARNING During its run `repgen` will start the other log formatter program `logformat`. That is why `repgen` works correctly only if the directory `$TTCN3_DIR/bin` is included to the path.

Chapter 6. References

- [1] Installation guide for TITAN TTCN-3 Test Executor
- [2] Programmers Technical Reference for TITAN TTCN-3 Test Executor
- [3] Release Notes for TITAN TTCN-3 Test Executor
- [4] TTCN–3 Style Guide 1/0113-FCPCA 101 35
- [5] TTCN–3 Naming Convention ETH/R-04:000010