Copyright © Ericsson AB 2018. All rights reserved.

This program and the accompanying materials are made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at

Copyright © Ericsson AB 2018. All rights reserved.

This program and the accompanying materials
are made available under the terms of the Eclipse Public License v1.0
which accompanies this distribution, and is available at

All link starts a standalone presentation (“custom shows”) which returns to this page.
I. TITAN TTCN-3 TEST EXECUTOR: INTRODUCTION

WHAT IS TITAN?
HISTORY
BLOCK DIAGRAM
FEATURES

CONTENTS
WHAT IS TITAN?

TTCN-3 Test Executor
- Compile Abstract Test Suites written TTCN-3, ASN.1 to C++
- Generate code skeleton and provide API for developing protocol adaptation
- Control hardware test tools that have well-defined API
- Execute the compiled Executable Test Suites

Runs on multiple platforms
- Platform independent source written in C/C++
- TITAN can be built on any platform with a decent C++ compiler
  (Currently available for: Solaris, Linux, FreeBSD, Windows+cygwin at http://ttcn.ericsson.se ➔ TITAN DOWNLOADS)

Optimized for performance testing

API: Application Programming Interface
HISTORY OF TITAN

2000: M.Sc. thesis work
   - Goal: performance test tool
   - version 1.0 single mode, Test Port API, no semantic analysis

2001-2002: Research prototype
   - Application driven development
   - version 1.1 parallel & distributed execution
   - version 1.2 ASN.1 support, RAW & BER encoding

2003: Productification
   - Candidate as official TTCN-3 tool within Ericsson
   - version 1.4 full ASN.1 support
   - version 1.5 TTCN-3 semantic analysis (static parts)

2004-2008: Official Ericsson TTCN-3 tool
   - Version 1.5pl6 Graphical User Interface
   - Version 1.6pl0 Full semantic analysis
   - Version 1.7pl0 TTCN-3 edition 3

BER: Basic Encoding Rule
ASN.1 Abstract Syntax Notation One
TITAN COMPILER FEATURES I.

Supports modularity
Supports TTCN-3 types:
- All data types
- All test configuration & default types (port, component, address & default)
- Sub-typing (type constraints)
- Semantic check for all supported TTCN-3 types
- RAW, TEXT encoding (TITAN specific, because no standard exists)

ASN.1 support:
- Full semantic check of ASN.1 types and values
- XML related ASN.1 language elements are not supported
- ASN.1 encodings: BER (direct), PER & (E)XER (using 3rd party SW)
TITAN COMPILER FEATURES II.

Supports TTCN-3 data
- Constants and run-time parameterization (module parameters)
- Templates (simple~, compound~, parameterized~, modified~, in-line~)
- Semantic check for all static data constructs

Support of message based communication
- Sending & receiving messages with template matching etc.

Support of procedure based communication (API testing)

Dynamic behaviours
- Dynamic test configuration and port/connection handling
- Sequential and alternative behaviours
- Default behaviour handling
- All functional elements of dynamic behaviour supported: testcases, functions, structured alternative behaviour (altsteps)
- All operations and control constructs supported (interleave since 1.6.pi3/R6D)
TITAN COMPILER FEATURES III.

- Timer handling
  - start, stop, read, timeout
- Test execution control from the TTCN-3 test suite
- Language extensions
  - Additional predefined functions
  - Value returning done (to support SCS/TTCN-2 behaviour)
  - Extended default parameterization (ports, timers etc.)
  - RAW and TEXT encoding for TTCN-3 types
  - Extensions are pushed in ETSI as CRs
- Ports with Test Port API

CR: Change Request
II. SETTING UP TITAN

FEATURES
PREREQUISITES
SETTING ENVIRONMENT
VARIABLES
LICENSING

CONTENTS
PREREQUISITES

The following tools need to be installed for using TITAN:
The same GCC C/C++ version shall be installed which the given TITAN
distribution was built with (with using as and ld from GNU binutils)

- setenv GCC_DIR to your GCC installation
- add $GCC_DIR/bin to your PATH
- add $GCC_DIR/lib to your LD_LIBRARY_PATH

makedepend utility shall be available
make (or gmake) shall be available

- using GNU make (gmake) is recommended
NEdit (>5.4) or XEmacs with TTCN-3 and ASN.1 language mode is recommended
SETTING UP THE ERICSSON TTCN-3 ENVIRONMENT

The following environment variables need to be set for using TITAN:

$TTCN3_DIR shall point to the base directory of the installation
Add $TTCN3_DIR/bin to PATH and $TTCN3_DIR/lib to LD_LIBRARY_PATH
Set $TTCN3_LICENSE_FILE to the full path and name of your license file
The next environment variables are used only for using TITAN GUI:
Set $TTCN3_BROWSER to your browser to be used by TITAN GUI (default browser: *netscape*)
Set $TTCN3_LOGBROWSER_EDITOR to your favorite editor (*nedit* or *xemacs* suggested)
CONTENTS OF THE ERICSSON TTCN-3 ENVIRONMENT

$TTCN3_DIR/bin – TTCN-3/ASN.1 compiler binaries, log tools, etc.
$TTCN3_DIR/etc
  - gui/ contains icon images for GUI;
  - nedit/ contains the NEdit support package (e.g. syntax highlighting);
  - xemacs/ contains the xemacs TTCN-3 and ASN.1 language mode support;
  - skeleton/ holds the TTCN-3 code skeletons;
  - license/ a demo license key
$TTCN3_DIR/include – include files required for translated modules and Test Ports
$TTCN3_DIR/lib – TTCN-3 and ASN.1 base libraries
$TTCN3_DIR/man – manual pages for some binaries
$TTCN3_DIR/doc – PDF and/or PS version of TITAN documentation
$TTCN3_DIR/demo – simple “Hello World!” application demo
LICENSING

You need a license to use TITAN!
Order your license at:
   http://ttcn.ericsson.se → license ordering
Copy your license file to your home:
   cp license_<number>.dat ${HOME}/
Set your TTCN3_LICENSE_FILE environment variable:
   setenv TTCN3_LICENSE_FILE
       ${HOME}/license_<number>.dat
Once your system-related environment variables are set, you can check
the validity of the license:
   from the shell command line: compiler -v
ECLIPSE

“Eclipse is a multi-language software development environment comprising an IDE and a plug-in system to extend it. It is written primarily in Java and can be used to develop applications in Java and, by means of the various plug-ins, in other languages as well, including C, C++, COBOL, Python, Perl, PHP, and others. The IDE is often called Eclipse ADT for Ada, Eclipse CDT for C, Eclipse JDT for Java and Eclipse PDT for PHP. “ WIKIPEDIA

Available at

The Ericsson customized version
Eclipse4Ericsson (E4E)
ECLIPSE AS A PLATFORM

Platform
- Provides a platform to support various programming languages
- Supports TTCN3/C++/JAVA/Tcl/Tk/XML/...etc

Common way of working
- Common shortcuts
- Common windows
- Common concept
- Common way of compiling
- Common way of executing

Eclipse
ECLIPSE TERMS: EDITOR

Editors
- appear in workbench editor area

Functionality
- Open, edit, save, close lifecycle
- Open editors are stacked

Extension point for contributing new types of editors

Example
- JDT provides Java source file editor
- TTCN-3 perspective provides TTCN-3 editor
ECLIPSE TERMS: VIEWS

Views provide information on some object

Views augment editors
- Example: Outline view summarizes content

Views augment other views
- Example: Properties view describes selection

Eclipse Platform includes many standard views
- Resource Navigator, Outline, Properties, Tasks, Search...etc

View API and framework
- Views can be implemented with JFace viewers
ECLIPSE TERMS: PERSPECTIVES

Perspectives are arrangements of views and editors
- Different perspectives suited for different user tasks
- Users can quickly switch between perspectives

Task orientation limits visible views, actions
- Scales to large numbers of installed tools

Perspectives control
- View visibility
- View and editor layout
- Action visibility

Extension point for new perspectives

Eclipse Platform includes standard perspectives
- Resource, Debug, ...
WAY OF WORKING WITH ECLIPSE

TCC provides 2 plugins → 2 perspectives for eclipse
- Titan Editing
- Titan Executing
WAY OF WORKING WITH ECLIPSE

The TTCN Editing perspective

Projects Editing area outline

Console: Information about the processes e.g. compilation
WOW: CREATE NEW...

File menu
- New
  - Titan Project/TTCN-3 Module/Cfg File
OR SIMPLY RIGHT CLICK...

To create new element:
- Either File menu → New → Select the type of the file to be created → select the target folder & assign a name to it.
- Or Select the target folder right click to it → new → select the type of the file to be created → assign a name to it.
WOW: ADD FILES TO THE PROJECT

By local copy

- File Menu → import →

  - Select the type to be imported (to import ordinary files select the General File System) → Next → Browse for the target directory (that contains the file, the file will be in grey) → press ok → Thick the files to be imported (You can add more than open file.)
WOW: ADD FILES TO THE PROJECT

Or...

- Right click to the folder you would like to import
  - import → Select the type to be imported (to import ordinary files select the General File System) → Next → Browse for the target directory (that contains the file, the file will be in grey) → press ok
  → thick the files to be imported
WOW: ADD FILES TO THE PROJECT

As link (By references similar to symlinks)

- File menu
  - New → File or Folder
  - thick the advanced box → thick the Link to file in the file system → browse for the target file select the file.
**WOW: BUILD THE PROJECT**

Build automatically
- Project menu Build Automatically
- After each and every save the code is compiled automatically
- The build option at the context menu turns to grey
WOW: BUILD THE PROJECT

Build manually
- Project menu ➔ Build Project
  - Builds the project
- Project menu ➔ Build All
  - Builds all the opened projects within the workspace
- Check the result of the process at the TITAN console

```
not in /opt/drive/c/orcyrin/mnt/RHE/include/etmacp.h
not in /opt/drive/c/orcyrin/mnt/RHE/include/etmacp.h
makedepend warning: f.o (reading /opt/drive/c/orcyrin/mnt/RHE/include/etmacp.h, line 4): cannot find include file "etmacp.h"
not in /opt/drive/c/orcyrin/mnt/RHE/include/etmacp.h
makedepend warning: f.o (reading /opt/drive/c/orcyrin/mnt/RHE/include/memory.h, line 5): cannot find include file "etmacp.h"
not in /opt/drive/c/orcyrin/mnt/RHE/include/memory.h
Operation finished successfully.
sh -c make all -j 2
gcc -o -W-warnings -I/opt/drive/c/orcyrin/mnt/RHE/include -Wall -o f.o f.o 
-L /opt/drive/c/orcyrin/mnt/RHE/lib -ltomh-parallel 
-L /opt/drive/c/orcyrin/mnt/RHE/lib -ltomh 
-L /opt/drive/c/orcyrin/mnt/RHE/lib -lkernel 
-L /opt/drive/c/orcyrin/mnt/RHE/lib -ltomh
Operation finished successfully.
```
WOW: RUN THE PROJECT

Right click to the project
- Run As..
  - TITAN Parallel launcher
- Select the testcase/control part to be executed

Check the result at the Titan test result view (TITAN EXECUTOR perspective)
WOW: OPEN THE LOG FILE

Locate the log files (files with .log extension)
- Double click to the testcase...
LOGVIEWER PLUGIN

Locate the log files (files with .log extension)
- Each and every test component by default has its own log file
- The information is distributed among the testcases, therefore these log files should be merged first
- Select the log files → right click → Titan Menu Merg

Name the merged log file

© Ericsson 2002-2010.
LOGVIEWER PLUGIN

Open the merged log file

- By a double click to the name, the table view will be opened
  - It is a textual representation of the log file in a tabular format

- By a double click to the testcase name (e.g. “download” in the screenshot) the MSC view will be opened
LOGVIEWER PLUGIN

Open the merged log file

- By a double click the name, the table view will be opened
  - It is a textual representation of the log file in a tabular format

- By a double click to the testcase name (e.g. “download” in the screenshot) the MSC view will be opened
LOGVIEWER PLUGIN

Switch between views: right click to an event
LOGVIEWER PLUGIN

Follow in source: if the cfg file is configured to use source info, the log events are traceable in the source file.
A convenient development environment could contain the MSC + Source + Value views.
WOW: SET THE PREFERENCES

Window menu → Preferences

GENERAL
Editor Font size:
   Appearance Colors and Fonts
   Basic → Text Font
   click edit to set the font size

TITAN Preferences
Content assistant settings

On-the-fly analyzer settings

To tweak the ‘{’, ‘”’ ‘}’ auto insertion
Typing
VII THE TITAN CONFIGURATION FILE

CONFIGURATION FILE STRUCTURE
SECTIONS

CONTENTS
The usual suffix of configuration files is `.cfg`. 
MODULE_PARAMETERS
This section shall contain the values of all parameters that are defined in your TTCN–3 modules.

LOGGING
In this section you can specify the name of the log file and the classes of events you want to log into the file or display on console (standard error).

TESTPORT_PARAMETERS
In this section you can specify parameters that are passed to Test Ports.

EXTERNAL_COMMANDS
This section can define external commands (shell scripts) to be executed by the ETS whenever a control part or test case is started or terminated.

EXECUTE
In this section you have to specify what parts of your test suite you want to execute.

GROUPS (used in parallel mode)
In this section you can specify groups of hosts which can be used to restrict the creation of certain PTCs to a given set of hosts.

COMPONENTS (used in parallel mode)
This section consists of rules restricting the location of created PTCs.

MAIN_CONTROLLER (used in parallel mode)
The variables herein control the behavior of MC.
In single mode the configuration file is useless without this section.
In parallel mode the section is optional.
The executable test program produces a log file during its run. The log file contains the important events of the test execution with time stamps. In this section you can specify the name of the log file and the classes of events you want to log into the file or display on console (standard error).

**LogFile**: To make the names of the log file of the components unique, the string should contain special metacharacters, which are substituted dynamically during test execution. (See Table 8.1 of TITAN User Guide).

**FileMask** resp. **ConsoleMask** may contain symbolic constants enumerated in table 8.2. of TITAN User Guide

**AppendFile** controls whether the run-time environment shall keep the contents of existing log files when starting execution.

**TimeStampFormat**: has three possibilities: Time stands for the format hh:mm:ss.microsec. DateTime results in yyyy/Mon/dd hh:mm:ss.microsec. Seconds produces relative timestamps in format s.microsec. (Zero time is at the starting of the test component or test execution). The default value is Time.

**LogEventType** indicates whether to include the symbolic event type (without the TTCN prefix) in each logged event immediately after the timestamp. The event types are listed in Table 8.2. of TITAN User Guide.
The table lists the symbolic constants referring to bit masks used for filtering the events to be written to the log file or the console, respectively.
This section contains the values of all parameters that are defined in your TTCN–3 modules.
This section defines external commands (shell scripts) to be executed by the ETS whenever a control part or test case is started or terminated.

In case of parallel mode, the external command is executed on the host where the MTC runs.

```plaintext
[EXTERNAL_COMMANDS]
BeginControlPart := "/usr/local/bin/DisplayNOTE"
EndControlPart := "/usr/local/bin/ClearNOTE"
BeginTestCase := "/usr/local/bin/startTCPDUMP"
EndTestCase := "/usr/local/bin/stopTCPDUMP"
```
The [MAIN CONTROLLER] section controls the behavior of MC and includes three variables. Variable TCPPort determines the TCP port on which the MC application will listen for incoming HC connections. Use a port number greater than 1024 if you don’t have root privileges! The recommended port number is 9034.

The optional NumHCS variable provides support for automated (batch) execution of distributed tests. When the specified number of HCs are connected, the MC automatically creates MTC and executes all items of the [EXECUTE] section.

The KillTimer variable tells the MC to wait some seconds for a busy PTC to terminate when it was stopped. The purpose of this function is to prevent the test system from deadlocks.
Groups of hosts specified in the section [GROUPS] can be used in the [COMPONENTS] section to restrict the creation of certain PTCs to a given set of hosts.
VIII. TITAN INTERNALS

BASIC OVERVIEW OF COMPILATION PROCESS
SYNTACTIC/SEMANTIC ANALYSIS
CODE GENERATION
BUILDING/EXECUTING ETS
THE PARALLEL DISTRIBUTED ARCHITECTURE

CONTENTS
TITAN COMPILATION PROCESS

The compilation of TTCN-3 and ASN.1 modules comprises:
Parsing the input TTCN-3 and ASN.1 files (stops at first error)
Semantic analysis of input TTCN-3 and ASN.1 modules:
  • report all errors and warnings appearing in input;
  • continue only if no errors are found
Code generation for input modules
Invoking the C++ compiler to compile generated files
Linking the modules with Test Ports, external functions and the proper version of TTCN-3 and ASN.1 base libraries
SYNTACTIC AND SEMANTIC ANALYSIS

Syntactic analysis:
- Reports “typographic” errors (e.g. mistyped keywords, missing braces)
- Both erroneous and expected tokens are displayed
- No error recovery: Parse errors brake compilation process (work in progress)

Semantic analysis:
- Detect semantic errors (e.g. type mismatch, formal-actual parameter mismatch, incorrect references) in input
- Full semantic analysis for TTCN-3 and ASN.1 modules since 1.6.pl0
- Supports ambiguous language constructs (e.g. start)
- Error recovery: detects all errors in one run w/o avalanche effect
- Basis for code optimization (in future)
CODE GENERATION

- Primarily optimized for execution speed
  - Compilation time had lower priority
- Modest memory usage during execution
- Large C++ classes for data types
  - Relatively long compilation for complex protocols
- Compact code for other definitions
  - Fast compilation for data values & templates
  - One-by-one mapping between TTCN-3 and C++
    behavior statements and expressions
- Exploiting many C++ language features
  - Data representation: Object Oriented
  - TTCN-3 test behaviours: C-like functions
BUILDING EXECUTABLE TESTS

Parts of the executable:
- C++ code generated from TTCN-3 and ASN.1 modules
- Test Ports and external functions (user code)
- Base Library (part of TITAN)

Fully automated build process using standard tools such as make, makedepend, C++ compiler (GCC)

Incremental builds are possible
- Type definitions change seldom
- Optimal TTCN-3 module structure is important to keep compilation time low

Makefile generation from “project”
Clean API for external functions and libraries
IMPORTANT NOTES

Platform independent source but platform-dependent binaries
TITAN version, C++ compiler version, module object version, test port
version MUST be synchronized!
When change to another TITAN version, you should make clean, and
rebuild all files.
Important to have correct build hierarchy – use Makefile and always update
module dependency list if import structure changes!
Always read the limitations section of the user guide!
TEST EXECUTION IN TITAN

Two operation modes
- Single mode: only MTC exists, mainly for debugging
- Parallel mode: fully featured, with PTCs

Run-time configuration file required
- TTCN-3 module parameters
- Test Port parameters
- Logging options (filters, format, …)
- List of test cases to execute (batch execution)

MTC: Main Test Component
PTC: Parallel Test Component
DEBUGGING AND TEST RESULT ANALYSIS

Debugging capabilities:
- No source level TTCN-3 debugger
- Precise, self-explanatory error messages from semantic analyzer and RTE
- Console logging and configuration monitoring
- C++ debuggers (GDB) are usable for Test Ports in single mode

Logging:
- Configurable, detailed logging to console and file
- Log entries contain timestamp, reference to source file and line number, etc.
- Separate log file (with configurable name) for each test component
- Almost “everything” can be logged explicitly from TTCN-3 using log
- Logging of template matching failures in alternatives (TTCN_MATCHING)
- Post-processing (merging, filtering, pretty-printing) of log files
LOG FILE PROCESSING UTILITIES

```
ttcn3_logformat [-i n] [-o outfile] [-s] {input.log}
  - Pretty-prints contents of file.log or stdin
  - Result is saved into outfile or written stdout
  - n defines indentation depth
  - -s places each test case log into a separate file

  ttcn3_logmerge [-o outfile] {input.log}
  - Merges the content of all argument files into a single log file sorted by increasing timestamp. Results go into outfile or stdout.
  - Only files using the same timestamp format (Time, DateTime) are merged (Seconds not implemented!)

  ttcn3_logfilter [-o outfile] {eventtype+-} {input.log}
  - Filters events specified on argument list from input.log or stdin.
```
PARALLEL DISTRIBUTED TEST EXECUTION – FEATURES

Execution environment
- Distributed parallel test execution
- Can work on heterogeneous environment (over a TCP/IP network)
- Scalability, load balancing between group of hosts
- Dynamic component creation and run-time test reconfiguration (controlled from the TTCN-3 code)
- Transparent internal communication (between connected test components)
- Configurable automatic test execution (mixed list of module control parts & test cases)

High execution performance
- Applicable for load testing (traffic generation) as well
- Message sending takes 0.1…0.2 ms (highly depends on machine performance and test port code efficiency)
COMPONENTS OF PARALLEL DISTRIBUTED TEST ARCHITECTURE

Main Controller (MC)
- Central test campaign management, configuration and log monitoring
- Embedded GUI or standalone CLI version
- Maintains direct control connection with all components

Host Controller (HC)
- An instance of the executable program
- Exactly one HC on each computer taking part in distributed execution
- Creates a new test component by duplicating itself

Main Test Component (MTC)
- Special test component that created first at test case execution
- Only one MTC in the Test System but it can change its component type
- Can execute the control part of a TTCN-3 module

Parallel Test Component (PTC)
- Created by HC upon request from MC
- Executes TTCN-3 function
DISTRIBUTED OPERATION OF PARALLEL TEST ARCHITECTURE

Test System

- host 1 shell 1
  - CLI
  - GUI
  - Main Controller (MC)
  - Config. file

host2 (host 1 shell2)
- Host Controller (HC)
- MTC
- PTC
- PTC
- PTC

host3
- Host Controller (HC)
- PTC
- PTC
- PTC

SUT

mapped ports

connected ports
THE CONTROL PROTOCOL BEHIND THE PARALLEL ARCHITECTURE

Purpose of communication
- PTC creation / termination
- Establishing / destroying port connections and mappings
- Transport of internal TTCN-3 messages between PTCs
- Test campaign management

Messages are sent over a reliable transport channel

Platform independent abstract messages

Verified with SPIN model checker

Around 50 different PDUs

Possible bottleneck: MC
- Used for test setup only, inactive during load generation
EXAMPLE: PTC CREATION

MTC
- Reaching a create operation
  - CREATE_REQ (comp_type)
  - Waiting
  - Continuing execution

MC
- Choosing a host
  - Allocating id
  - CREATE (comp_type, id)
  - CREATE_ACK (id)

HC
- CONTROL CONNECTION ESTABLISHMENT
  - CREATED (id)

PTC
- WAITING FOR START

© Ericsson 2002-2010.
ECLIPSE EXECUTOR PLUGIN

The TTCN executor perspective

To control the execution start/stop, etc

Summary about the execution

Editing area (the same window)
ECLIPSE EXECUTOR PLUGIN

The executor plugin provides a GUI interface to set all the required parameters

- Run menu..
  - Run Configuration...
- Create new Parallel Launcher (or edit an existing one)
  - Set the MC options
  - Set the HC options
- Launch the code
WAY OF WORKING WITH ECLIPSE PLATFORM

- Basic Main controller options
  - Browse for the project to be executed
  - The three records below will be filled up automatically
  - Browse for the cfg file if there is any
- Host Controller tab
  - Press the init button it will initialize all the required field

Press Run
IX. BUILDING EXECUTABLE TEST SUITES FROM COMMAND LINE (W/O GUI)

COMPILING MANUALLY
BUILDING EXECUTABLE TEST SUITES
AUTOMATIC BUILD USING MAKE

CONTENTS
BUILDING THE ETS MANUALLY

Translate your TTCN-3 core language module into C++:

```bash
$TTCN3_DIR/bin/compiler modulename.ttcn
```

Results: modulename.hh, modulename.cc (provided the module has the same identifier as the file’s name!)

Compile the generated C++ code:

```bash
g++ -c -I$TTCN3_DIR/include modulename.cc -o modulename.o
```

Link the object with the TTCN-3 base library and OpenSSL code library:

```bash
g++ -o modulename modulename.o -L$TTCN3_DIR/lib -lttcn3 -lcrypto
```
BUILDING THE EXECUTABLE TEST SUITE

Generate makefile:
```
ttcn3_makefilegen (module)* (testport)* (other)*
-e PROJECTNAME
```

Edit the generated makefile:
- Set the TTCN3_DIR to the base directory of installation
- Assure that TTCN3_LIB contains the desired TTCN-3 base library
  (`ttcn3` for single mode, `ttcn3-parallel` for parallel mode)
- Ensure that cxx points to the appropriate C++ compiler version
- Optionally add new switches to CPPFLAGS, CXXFLAGS and LDFLAGS
- You SHOULD NOT modify the rest of the generated makefile –
  regenerate it when new modules are added!

Resolve import dependencies – `make dep`
Build the ETS – `make`
EXTRA FEATURES IN MAKEFILE

Perform syntactic and semantic analysis only `make check`
Clean all generated files using `make clean`
Make an archive of all sources under directory `ARCHIVE_DIR` using `make archive`
More advanced features of GNU make are utilized when generating Makefile with `-g` flag
X. EXECUTING THE EXECUTABLE TEST SUITE FROM COMMAND LINE (W/O GUI)

EXECUTION IN SINGLE MODE
PARALLEL MODE TEST EXECUTION

CONTENTS
EXECUTING THE ETS IN SINGLE MODE

The ETS generated during the build process can be executed without the need for any other application when the ETS was built in single mode. The generated output file will be the ETS itself.

ETS takes one mandatory parameter for its execution: the configuration file.

Optional parameters of the ETS:

- v prints tool version, license information, compilation time and checksum of the participating modules

- l lists the names of all control parts and test cases in the ETS
EXECUTING THE ETS IN PARALLEL MODE

The execution in parallel mode requires the Main Controller (MC)
Two kinds of MC: GUI version (see GUI pres.) and CLI version
The CLI version of MC can be started with:
```bash
mctr_cli [module-cfg-file]
```
where module-cfg-file is the run-time configuration file
In addition to the ETS itself, the target file generated during build process
contains some static code, the Host Controller (HC)
The HC can be started from a shell; it requires the hostname (or IP
address) and TCP port number of the MC in order to connect to it:
```bash
modulename <MC-hostname> <MC-port>
```
The parallel architecture is based on client-server mode operation between
MC (TCP server) and some HCs (TCP clients)
MAIN CONTROLLER COMMANDS IN CLI MODE

MC commands available in CLI:
- help [command]: display help overview or specific help on command
- cmtc: create Main Test Component (1st step of test execution)
- smtc [something-to-execute]: start something-to-execute or contents of the [EXECUTE] section of the run-time configuration file on MTC
- emtc: terminate Main Test Component
- quit, exit: terminate MC and quit to shell
- stop: stop test execution at next appropriate state
- pause [ on | off ]: toggle interrupt after test case execution
- continue: resume interrupted test execution
- info: display information on current status of test configuration
- !command: execute command in underlying shell
EXAMPLE EXECUTION WITH SINGLE HC

Start the MC in shell 1:
`mctr_cli Test.cfg`

<MC gives prompt and waits for HC connections>

<HC connection is accepted>

<No more HC is expected>

Create MTC: `cmtc`

Execute control part of module
Test: `smtc Test.control`

<Wait for execution to finish>

Shut down the MTC: `emtc`

Quit MC: `quit`

TCPPort := 9999 is set in [MAIN_CONTROLLER] section

Start a HC in shell 2:
`Test mc-host 9999`

<Terminal is blocked>
XI. TITAN TEST PORTS

TEST PORT BASICS
TEST PORT API
PROGRAMMING A TEST PORT

CONTENTS
WHAT IS A TEST PORT (TP)

Ericsson proprietary communication interface between TITAN and SUT (i.e. abstract TTCN-3 test configuration and real world)
Translate abstract TTCN-3 messages into real-world messages
It contains:
- Formal Data Type Definitions
- Encoder/Decoder Functions
- (Utility Functions)
- Transport

Test System Architecture

TTCN-3 abstract environment

```
| type record
| p.send(t.MyType)
```

SUT

010100110110....

Test Port

Formal Data Type Definitions
Encoder/Decoder Functions
Utility Functions
Transport
TEST PORT CONCEPT

Ericsson proprietary communication interface between TITAN and SUT (i.e. abstract TTCN-3 test configuration and real world)
Translate abstract TTCN-3 messages into real-world messages:
  - Directly or using some 3rd party HW or SW (e.g. EIN protocol stack, UNIX OS)
  - Protocol specific encoding and decoding
Consequently:
  - Different protocols need different Test Ports
  - Alternative applications of the same protocol may need different Test Ports
Test Ports written by TCC are PRODUCTS related to TITAN, therefore:
  - come with full documentation (User Guide, Function Specification, PRI)
  - include maintenance (Trouble Reporting in MHWEB)

SUT: System Under Test
TCC: Test Competence Center
TEST PORT PROPERTIES AS SEEN FROM TTCN-3

Test Ports are transparent in TTCN-3
Test Ports are associated to communication ports of components
- except ports used solely for internal communication
  i.e. declared using with { extension "internal" } attribute
Link between TTCN-3 port type definition and TITAN Test Port is their identifier:
- TTCN-3 port type: type port MyPort message { ... };
- is for instance implemented in C++ as: class MyPort;
Single Test Port instance is used for each TTCN-3 port
Test Ports are created and started together with the TTCN-3 ports automatically at component instantiation
TEST PORT CONTENTS

Test Ports can be obtained from TCC’s ClearCase VOB
- usually mounted under /vobs/ttcn/TCC_Releases/TestPorts
- refer to TCC’s FAQ pages on intranet or contact your ClearCase administrator otherwise!

Test Ports contain
- TTCN-3 source file with port type definitions of the Test Port
- Optional additional TTCN-3 source file containing additional definitions
- A pair of C++ header and source files containing the Test Port implementation
- Extensive documentation
- Optional unofficial demo or test modules in TTCN-3
USING TEST PORTS

Always read Test Port’s User Guide first
Required Test Ports must be added to TITAN Project
- including all source files (e.g.: *.ttcn, *.cc, *.hh) and optional dependencies
  (e.g.: Abstract Socket)
TTCN-3 module of selected Test Port containing the port type definition
MUST be imported into your module
TTCN-3 and/or ASN.1 module(s) of selected Test Port containing the
protocol type definitions MUST be imported into your module
Test Ports can accept run-time parameters
- depending on particular Test Port product
- documented in Test Port’s User Guide
- can be set in [TESTPORT_PARAMETERS] section of configuration file
In this section you can specify parameters that are passed to Test Ports.

[TESTPORT_PARAMETERS]

Specify parameters to be passed to Test Ports when started

Syntax:

\(<component\_name>.\<port\_name>.\<parameter\_name>\) :=
  \<parameter\_value>\n
\(component\_name\) – component reference (integer value), mtc, system
  or * (all components)
  – system refers to Test System Interface, passed at mapping of system
  ports

\(port\_name\) – port identifier, including optional array reference; or * (all
  ports of the given component)

\(parameter\_name\) – depends on the test port used

\(parameter\_value\) – only charstring values permitted
PROGRAMMING A TEST PORT

Test Port skeleton can be generated from port type definition
- Port type definitions of a Test Port reside in a separate TTCN-3 module that has to be imported in all TTCN-3 modules, which use this Test Port
- When generating test port skeleton a pair of .hh and .cc files are created

Protocol-dependent parts written by the user:
- C++ functions generated for messages declared out are invoked when the particular message is sent in TTCN-3; these functions are responsible for message encoding
- The generated event handler function gets triggered by state change of designated file descriptors or at regular time interval
- The event handler implementation is responsible for decoding and forwarding incoming messages to the TTCN-3 Run-Time Environment

Test Port functions MUST be non-blocking
```ttcn
// TestPortTypeDef.ttcn
// TTCN-3 module containing Test Port's port type definition
module TestPortTypeDef {
    type port PortTypeDef message {
        inout charstring;
    }
}

// C++ class implementing the Test Port resides in:
// PortTypeDef.hh, PortTypeDef.cc

// Using the Test Port defined in module TestPortTypeDef
module ExampleUsesTestPortTypeDef {
    import from TestPortTypeDef all;
}
```
WHAT IS A PROTOCOL MODULE (PM)

Protocol modules implement the message structure of the related protocol in a formalized way, using the standard specification language TTCN-3.

It contains:
- Formal Data Type Definitions
- [Encoder/Decoder Functions]
- [Utility Functions]

Test System Architecture

TTCN-3 abstract environment

- type record
  - p.send(t_MyType)

Protocol Module

PDU (FABABA12)

TestPort

11100 0101001101101....

SUT
EXAMPLE

module Demo {
  import from GTPC_Types all;
  import from UDPPortType all;

  type component GTP_C ( port UDPPortType UDP_PCO ; )
  testmain to() run on...
  { ...
    var PDU_GTPC v_gtpc_pdu:={
      pn_bit="0'B,
      s_bit="1'B,
      s_bit="1'B,
      spare="0'B,
      pt="1'B,
      version=100'B,
      messageType="0A'0,
      length=0,
      tid=01020304'0
      opt_part=omit,
      gtp_pdu:={subRequest={(private_extension_gtpc:omit)}}
    }

    var ASP_UDP_message v_udp_asp;
    v_udp_asp:={
      data: enc_GTPC_PDU(v_gtpc_pdu),
      portf: 0,
      id:omit
    }
    UDP_PCO.send(v_udp_asp);
  } // C++ class implementing the Protocol Module's encoding function
  // TCP_EncDec.cc

  type record PDU_GTPC {
    BIT1 pn_bit,
    BIT1 s_bit,
    BIT1 s_bit,
    BIT1 spare,
    BIT1 pt,
    BIT3 version,
    OCT4 messageType,
    LINS NO_LAST
    OCT4 tsi,
    OCT4 Reader_optional_part opt_part optional,
    GTPC_PCDs gtpc_pdu
  }

  type record ASP_UDP_message{
    UDP_UDP data,
    AddressType data_addr optional
    PortType remote_port optional,
    integer id optional
  }