Title:

D.1.1.1 Specification of Test Infrastructure and Architecture

Version:	2.0
Date :	**30/10/2005**
Pages :	65

Author: FOKUS

To:
TT-MEDAL Consortium

Tests & **T**esting **Me**tho**d**ologies with **A**dvanced **L**anguages

The TT-MEDAL Consortium consists of:

Conformiq Software, CWI, DaimlerChrysler, FOKUS, Improve Quality Services, LogicaCMG, NetHawk, Nokia, Railinfrabeheer, Testing Technologies, VTT Electronics

Printed on:

Status:

[ ] Draft
[ ] To be reviewed
[ ] Proposal
[X] Final / Released

Confidentiality:

[X] Public - Intended for public use
[ ] Restricted - Intended for TT-MEDAL consortium only
[ ] Confidential - Intended for individual partner only

Deliverable ID: **D.1.1.1**

Title:

Specification of Test Infrastructure and Architecture

Summary / Contents:

The specification of the TT-Medal test infrastructure and architecture aims to provide a comprehensive overview and understanding of the test infrastructure elements and tools required to perform industrial TTCN-3 tests. An introductory information including a market viewpoint and an overview on the overall architecture of test infrastructure and platform components is given at the beginning. The major contents of this document is the explanation of test infrastructure elements, i.e. general TTCN-3 test system entities for a distributed test execution. In addition the document presents required features for test logging, the integration of IDL or XML based information on the SUT, and extensions regarding the adaptation to heterogeneous technical SUT interfaces in addition to Java or C++. Furthermore the document includes an introduction of different test architectures used in TT-Medal case studies and a catalogue of a basic terminology for testing that has been derived from various standardization bodies in order to support a common understanding on the TT-Medal infrastructure.

Note: Since there is a strong relationship between Test Infrastructure and Test Methodology it is recommended to consider als the contents of TT-Medal deliverable D1.1.2.

TABLE OF CONTENTS

CHANGE LOG

Vers.	Date	Author	Description
0.1	01.11.04	Axel Rennoch (FOK)	Initial version based on the German milestone Z21 with contributions from Ina Schieferdecker (FOK), Justyna Zander (FOK), Dimitrios Apostolidis (TT), Stephan Pietsch (TT)
0.1.1	11.11.04	Edzard Höfig (FOK)	Update of Terminology section
0.2	22.11.04	Axel Rennoch (FOK)	Addition of initial contributions from Richard Poppe (CMG), Thomas Deiss (NOK), Matti Kärki (VTT), Dirk Tepelmann (TT)
1.0	29.12.04	Edzard Höfig, Axel Rennoch (FOK)	Consideration of reviewers comments; Extension of the Terminlogy section
1.5	04.10.05	Axel Rennoch (ed.)	Update of ch. 4.7 (E. Höfig, A. Hinnerichs) and 4.8 (M. Kärki), addition of new ch. 5 (T. Deiss)
2.0	20.10.05	Axel Rennoch (ed.)	Consideration of reviewers comments;

APPLICABLE DOCUMENT LIST

Ref.	Title, author, source, date, status	Identification
[1]	Methods for Testing and Specification; The Testing and Test Control Notation Part 1 : Core Language, ETSI ES 201 873-1, v2.2.1	
[2]	Methods for Testing and Specification; The Testing and Test Control Notation Part 3 : Graphical Presentation Format, ETSI ES 201 873-3, v2.2.1	
[3]	Methods for Testing and Specification; The Testing and Test Control Notation Part 4 : Operational Semantics, ETSI ES 201 873-4, v2.2.1	
[4]	Methods for Testing and Specification; The Testing and Test Control Notation Part 5: TTCN-3 Runtime Interface (TRI), ETSI ES 201 873-5, v1.1.1	
[5]	Methods for Testing and Specification; The Testing and Test Control Notation Part 6 : TTCN-3 Control Interfaces, Draft ETSI ES 201 873-6, v1.0.0	
[6]	Z1: DaimlerChrysler Requirements	
[7]	Z1: Nokia Requirements	
[8]	ETSI TS 102 219 V1.1.1 (2003-06): IDL to TTCN-3 mapping	
[9]	Test Process Improvement, T. Koomen and M. Pol, Addison-Wesley / acm press, 1999	
[10]	Structured Testing of Information Systems, M. Pol and E. v. Veenendaal, Kluwer, 1998	
[11]	OMG CORBA (V2.4): "The Common Object Request Broker: Architecture and Specification", Section 3, October 2000.	
[12]	http://www.corba.org/vendors/pages/roguewave.html	
[13]	MOST cooperation: http://www.mostnet.de/downloads/Specifications/MOST	
[14]	World Wide Web Consortium: XML Schema, W3C Recommendations, May 2001. http://www.w3.org/XML/Schema	
[15]	World Wide Web Consortium: SOAP documentation. W3C Recommendation. http://www.w3.org/TR/soap	
[16]	D.-M. Jeaca: XMLSchema to TTCN-3 Mapping. Diploma thesis. Fraunhofer FOKUS, September 2004.	
[17]	Syntext dtd2xs v2.0 user's guide, Syntext Inc. 2003, http://www.syntext.com/products/dtd2xs/doc/index.html	
[18]	IEEE Standard for Software Test Documentation, No. 829-1998	
[19]	Int. Software Testing Qualification Board: Glossary of terms used in Software Testing. Version 1.0, December 2004,	
[20]	ISO/IEC 9126 - Information Technology, Software Product Evaluation, Quality, Characteristics and Guidelines for their Use	
[21]	ISO 8402 - Quality management and quality assurance – Vocabulary (revised by ISO 9000 - Quality management systems -- Fundamentals and vocabulary)	
[22]	ITU-T X.290: OSI Conformance testing methodology and framework, 04/95	
[23]	OMG Adopted Specification: UML 2.0 Testing Profile Specification, ptc/03-08-03	
[24]	S. Blom, N. Iuostinova, J v.d. Pol, Testing Railway Interlockings with TTCN-3, TTCN-3 User Conference, Sophia-Antipolis, 2005	
[25]	C/C++ To TTCN-3 Mapping, version 0.3, 08.03.2005 http://portal.etsi.org/docbox/MTS/MTS/05-Meetings/200503-MTS40/	

EXECUTIVE SUMMARY

The first deliverable in project task 1.1 reports on the initial specification of the TT-Medal test infrastructure and architecture. It provides an overview on required test tools and focuses on test execution. The described test infrastructure elements have been selected due to the requirements collected in WP4. The current specification is related to implementations developed in WP 2 and 3. The development of an overall testing methodology and the integrated use of the test tools is subject of a different task 1.1 report [D1.1.2].

1. INTRODUCTION

TT-Medal develops a test platform based on TTCN-3 for various applications in mobile communication systems (e.g. in GSM, GPRS and UMTS), automotive (e.g. in MOST and CAN), and – if possible - other domains like financial software etc.

TTCN-3 is a test specification and implementation language (as it defines the test execution interfaces TRI and TCI; part 5 and 6 of the TTCN-3 standard series), but it leaves open further aspects of a test platform that addresses requirements of industrial testing environments. These requirements have been derived from case studies. The requirements address architectural, methodological and tooling aspects, which all have an impact on the overall architecture. This deliverable represents a first definition of the test infrastructure architecture of TT-Medal. It should serve as a reference basis for the tool development within the project.

The main contents of this report is the presentation of the key elements of the TTCN-3 toolset as it will be developed in the TT-Medal project, i.e. it explains the core elements of a TTCN-3 test architecture according to the ETSI standardization and additional extensions which are needed due to the TT-Medal case studies. Thus, the TT-Medal toolset and the specific technical interfaces are reflected to provide a coherent overview of the required TTCN-3 infrastructure elements.

At the beginning of this report a section on a market questionnaire has been placed to give a background on the conditions an innovative test infrastructure has to address from the customers viewpoint. The list of questions and some details about the participating companies are given in the annex. The results from this questionnaire have been available only recently and – as far as not done - will be considered for the updated version of this report.

An additional effort is given to an annex about important testing and software quality terms collected from various glossaries of standardization institutions.

2. MARKET SURVEY

The TTCN test specification language was originally developed in the telecom domain for protocol and conformance testing. The goal of the TT-Medal project is to investigate in the TTCN-3 test specification language so that it can be used in other domains as well. The telecom and automotive (embedded) domains are supported well by TTCN-3, because of the message and procedure based approach. The financial domain in which LogicaCMG is active uses a very different test approach, because it is almost always GUI (Graphical User Interface) based.

A market survey about test environments is held at different domains, with a focus on the financial domain, to compare test environments and to investigate whether improvements are needed for test methods and test tools. The results shall be used to improve the TTCN-3 test specification language application.

The highlights of the questionnaire results are summarized in the next four sections. Section *test equipment* describes the test equipment as well as some information about the System Under Test (SUT). The section *test development* describes how the tests are developed and/or generated. Section *test execution and analysis* describes how tests are executed, whether test automation is implemented and how results are analysed. Also adapters (encoder/decoder and stubs) and tools are part of this section. The last section, *testing practices*, describes whether testing is in place as a process, if improvements are needed for the process and/or tooling and whether there is time/budget to implement improvements.

2.1 TEST EQUIPMENT

In the telecom domain UNIX and embedded systems are used for product implementation (SUT). Testing of these SUTs is implemented on UNIX as well as on Windows systems. The UNIX testers are mostly used for protocol testing (SMS, MMS, etc.) and occasionally for testing database content (subscriber information, credits, etc.). UNIX testers are not (or rarely) used for GUI based testing (not X-window nor web-based interface). Windows based testers are used for protocol testing (e.g. WAP) as well as GUI- and web-based testing. The Windows tester is occasionally used for testing database content, but also then mostly via a GUI interface.

In general the financial domain uses a Windows front-end to test their SUT. The SUT itself can be implemented under Windows, but is in most cases a UNIX based system. The Windows test tools are running on PC's under Windows versions 98, 2000/NT and XP. Older versions of Windows are not in use anymore.

2.2 TEST DEVELOPMENT

Test case generation is not in place in the financial domain, which means that tests are created manually. Input for test creation is normally a functional specification or system requirements document. E.g. TestFrame method with the TestFrame Toolbar can be generally used to develop test cases. Its Language with ActionWords is used to describe the test conditions, test cases and test steps. Sometimes a proprietary test description language is used, which is directly related to a proprietary test tool. Test validation is only done by reviews or inspections.

The most appreciated features of the test development tools:

- readability and simplicity
- intuitive and easy to learn
- test data separated from test case
- possibility in test specification for reading dynamic values to be used in the rest of the test
- reuse of test specification / scripts / ActionWords
- test specification can be used to execute test manually (automation not required)

Improvements for test development tools could be:

- link to the requirements for traceability
- conditions and loops in test case design

2.3 TEST EXECUTION AND ANALYSIS

Customers are interested in test automation or are using automated testing for regression test already. Test execution for GUI based testing on Windows is mostly managed by the test execution tool (e.g. WinRunner) and occasionally by a proprietary (Unix) tool. Test results from the Test Engine are e.g. in RTF format (word document), and are in most cases verified manually. Proprietary test execution tools can give an overview of the total results.

Debugging of the test case by stepping through the script is an important feature, which is supported by most test automation tools. However, when a DLL-function is called, e.g. WinRunner does not stop the test execution anymore because it cannot set a correct breakpoint. It is very wishful that the test execution tool supports stepping through test cases, even when DLL-functions are called.

WinRunner is experienced as relatively slow for test execution compared to the actions executed on the SUT. This means that execution of the complete test suite is limited by the performance of this test tool instead of the SUT. This should not be the case.

Interfaces to the SUT, which are not supported by the GUI test tool, are generally implemented as proprietary tools with a commandline interface. The GUI test tools are able to start the proprietary tools and to enter the necessary commands.

The features that are appreciated most in currently used GUI-based test automation tools are:

- debugging test cases by step through the script
- extensions possible for protocol adapters

- overall reporting functions

Topics that could be improved:

- performance of the test tool
- selection for one or multiple test cases or complete test suite
- custom data types for calling API
- compiler function (to avoid test case modification afterwards)

2.4 TESTING PRACTICES

Most customers have a process in place for testing, which fits their needs. But there is no guarantee that test engineers are using the processes as intended, so the result of testing depends largely on the quality of these test engineers. An improvement can be to integrate the test process in the test management tooling.

Testing is in general a project based activity, which means that the costs for improvements and changes are also booked on the projects' budget. Therefore project managers are generally not interested in changing tools and methods, because it does not have an added value to their project. This could become a problem for introducing the TTCN-3 specification language to the market.

For those customers, who are investigating to change their test environment, a need is seen for ("out-of-the-box") integrated solutions for testing, instead of using different and proprietary tools. Tool vendors more and more try to deliver (integrated) solutions for the whole testing process. This trend must be kept in mind when developing tools for TTCN-3.

3. OVERALL ARCHITECTURE

For the overall architecture, we propose to differentiate between the test infrastructure and the test platform as shown in Figure 1.

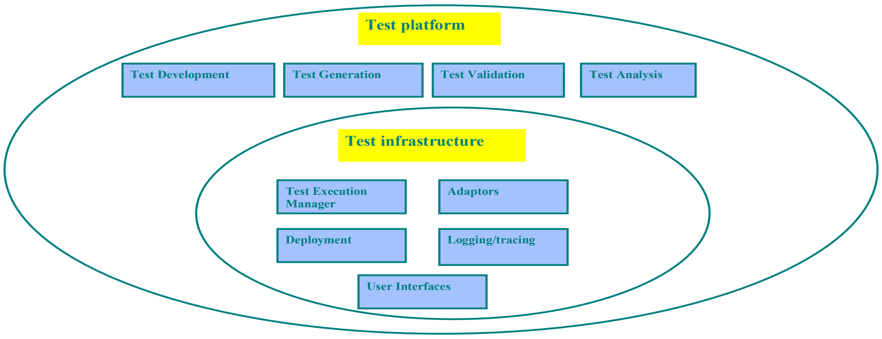


Figure 1. Overall Architecture

The **test infrastructure** comprises of all the components that are essential for the execution of TTCN-3 tests. These components are taken from the components defined in the TTCN-3 test system architecture as defined in the TTCN-3 runtime interfaces (TRI [4]) and the TTCN-3 control interfaces (TCI [5]). These components are basically components for the interaction with the system under test (SUT), for timing and external functions, for the control of the test execution, for the handling of (local and remote) test components, and for the encoding and decoding of data. Components that are beyond the TTCN-3 test system architecture are components for deployment and for logging and tracing. In addition, a user interface should be part of the test infrastructure. Both, graphical user interfaces (GUI) and command-line interfaces (CUI) should be supported.

The **test platform** consists of the test infrastructure and additional components needed to develop and maintain test systems. These additional components address the test development, test generation, test validation and test analysis.

3.1 OVERVIEW ON THE TEST INFRASTRUCTURE COMPONENTS

The test infrastructure provides means to execute TTCN-3 tests and to analyse the test results, i.e. it is used for test setup, execution and result presentation.

Abstract test suites (taken from test development) are compiled into executable tests with a TTCN-3 compiler (in our case TTthree).

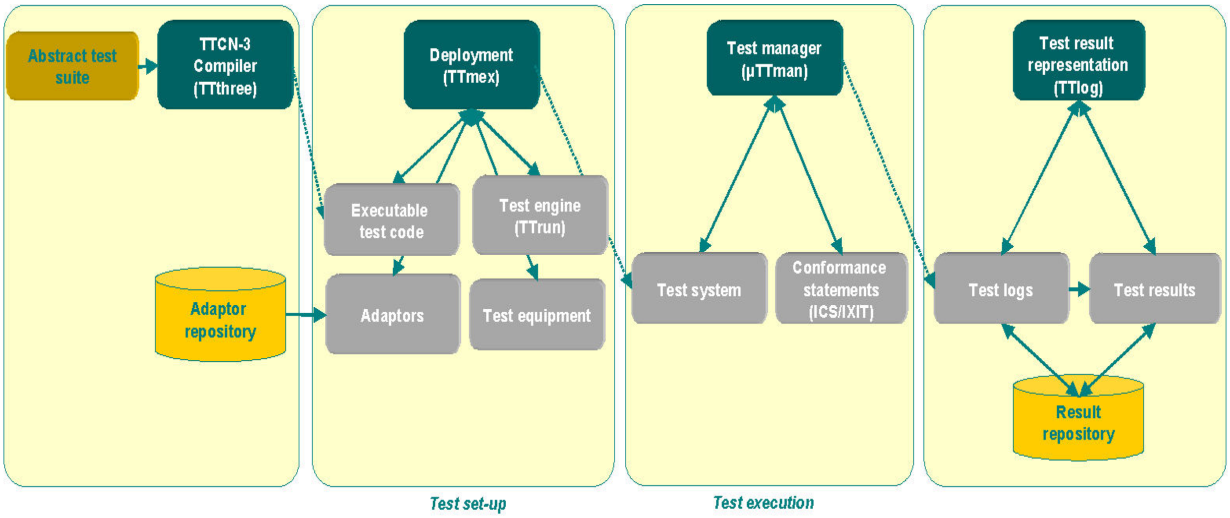


Figure 2. Test Infrastructure Components

During test set-up all components needed for the test execution are (locally or remotely) deployed. These components include the executable test code, the test engine (in our case TTrun), adaptors (often taken from adaptor repositories) and the test equipment itself. Deployment is supported by TTmex. The result of deployment is a fully functional test system.

The test system is the basis for test execution. It is configured and parameterized according to the answers in the Implementation Conformance Statement (ICS) and the Implementation Extra Information for Testing (IXIT). The test manager µTTman is used to control the test execution.

The results of test execution via the test manager are test logs, which are the basis for determining the final test results. The test logs and test results can be visualized with a test result representation (in our case TTlog). They can also be stored in a result repository to be used e.g. for future variations during test development phases.

3.2 OVERVIEW ON THE TEST PLATFORM COMPONENTS

The test platform offers additional components dedicated to the development, validation and analysis of tests. The tests are developed along the kind of tests (such as functional, interoperability, performance, scalability, load, stress, etc. tests) and the test purposes and test objectives for which they are aimed at. Tests can be specified in TTCN-3, in domain specific profiles of TTCN-3 (such as for mobile communication or for automotive), or in the UML 2.0 Testing Profile (which then needs to be mapped to TTCN-3). The results of the test development are abstract test suites in TTCN-3, which are compiled into executable code. That is given to the test infrastructure for execution.

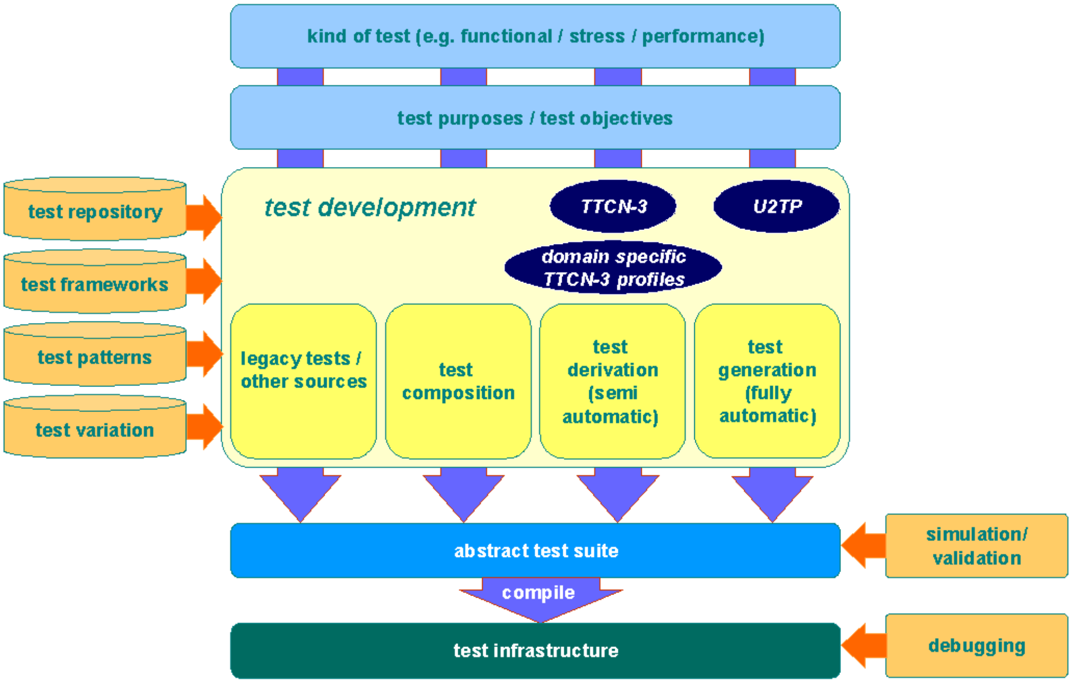

kind of test (e.g. functional / stress / performance)
test purposes / test objectives
test repository
test frameworks
test patterns
test variation
test development
TTCN-3
U2TP
domain specific TTCN-3 profiles
legacy tests / other sources
test composition
test derivation (semi automatic)
test generation (fully automatic)
abstract test suite
simulation/ validation
compile
test infrastructure
debugging

Figure 3. Test Platform Components

Test development itself can make use of legacy tests or tests from other sources such as test suites from standardization bodies or from manual test development. Tests can be composed from other tests; they can be derived semi-automatically or fully automatically generated from system models, system scenarios and/or test purpose models.

Legacy tests are typically part of test repositories. Test composition uses test patterns and test frameworks. Tests can be varied by means of parameterization or refinement.

Abstract test suites can be simulated and validated for internal correctness and for the correctness with respect to a system model or test purpose/test objective model. Executable test suites can be debugged. Simulation, validation and debugging are dedicated to analyse test suites and to check/demonstrate their correctness.

4. INITIAL INFRASTRUCTURE

To have a better overview about a TTCN-3 test system and its componentes, first of all the general structure of such a system and the needed interfaces will be presented. After that, the distributed test system architecture TTmex with all its components is explained.

4.1 GENERAL STRUCTURE OF A TTCN-3 TEST SYSTEM

A TTCN-3 test system, also commonly referred to as “tester“, can be thought of conceptually as a set of interacting entities where each entity corresponds to a particular aspect of functionality in a test system implementation. These entities manage test execution, interpret or execute compiled TTCN-3 code, realize proper communication with the SUT, implement external functions and handle timing.

The part of the test system which implements interpretation or execution of a TTCN-3 test specification is the TTCN-3 Executable (TE) entity. In a TTCN-3 tester, this entity corresponds either to an interpreter or an executable test suite (ETS). The ETS is produced by a compiler from a TTCN-3 abstract test suite (ATS). In case of using the TTthree compiler, it is an executable test suite, i.e. the compiled abstract test suite.

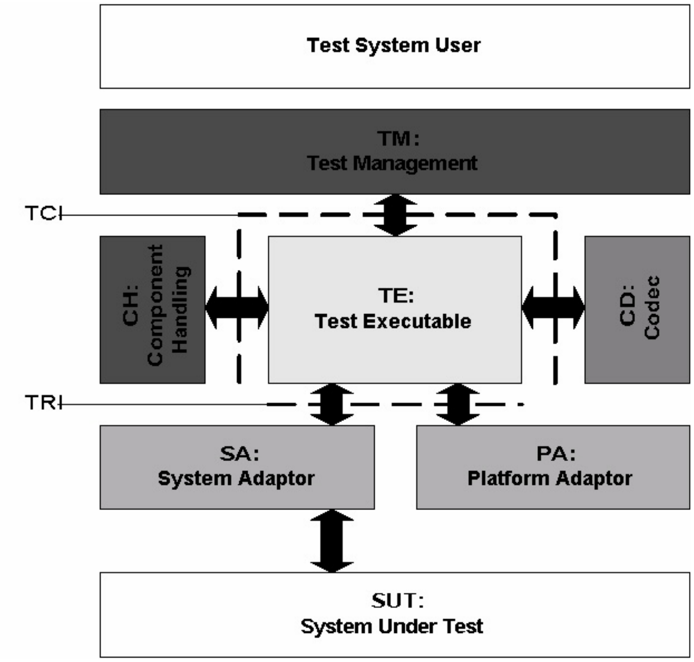


Figure 4. General Structure of a TTCN-3 Test System

The remaining part of the TTCN-3 test system, which deals with the aspects that cannot be concluded from information being present in the original ATS alone, can be decomposed into Test Management (TM), Component Handling (CH), Codec (CD), SUT Adapter (SA) and Platform Adapter (PA) entities. In general, these entities cover a test system user interface, test execution control, test event logging, communication between test components, encoding and decoding of TTCN-3 values, as well as communication with the SUT and timer implementation.

As depicted in figure 4, a TTCN-3 test system has two major interfaces, the TTCN-3 Control Interface (TCI) and the TTCN-3 Runtime Interface (TRI). The TCI [5] provides a standardized adaptation for management, test component handling and encoding/decoding of a test system to a particular test platform, i.e. it specifies the interfaces between Test Management (TM), Component Handling (CH), Codec (CD) and TTCN-3 Executable (TE) entities. The TRI [4] specifies the interfaces between the TE and Platform/SUT Adapter entities, respectively.

4.1.1 The TTCN-3 Executable (TE) Entity

As already mentioned, the TE entity is responsible for the interpretation or execution of the TTCN-3 ATS. Conceptually, the TE entity can be thought of an implementation of three separate tasks:

- control of test case execution
- proper execution of TTCN-3 behavior
- queueing of events

The control aspect implements the control part specified in TTCN-3 ATS. It properly sequences the execution of test cases or functions and collects the associated verdicts. Other tasks are the initialisation of SUT and platform adapters prior to the execution of a test case, the invocation of TRI operations for SUT operations, TTCN-3 external functions and timer implementations.

The behavioral aspect implements the execution or interpretation of test cases and functions as defined in the appropriate TTCN-3 modules. Other tasks are:

- propagation and matching of test events
- logging of test events
- creation and removal of TTCN-3 test components
- invocation of TRI operations in order to map test component ports to the system interface ports, to instruct the SA which message has to be sent to the SUT, to instruct the PA which external function has to be called or which timer has to be started, stopped, queried or read
- receiving of incoming messages or procedure calls from the SUT as well as timeout events
- handling of SUT action operations

Communication with the SUT has to be implemented within the SA, but communication between test components is the task of the TE. This applies to message based as well as procedure based communication.

The TE entity is also responsible for resolving the appropriate codecs in order to encode test data prior to the invocation of TRI communication operations and decode receiving test data from the SUT Adapter according to the encoding rules specified in the TTCN-3 ATS.

The queueing aspect is responsible for maintaining own port queues for receiving of operations and storing of events from the SA or PA. These events are not yet processed and therefore, the TE is informed to store them until snapshots are performed.

Additionally, also a list of timers, that have timed out, has to be kept, as these are also events that have to be evaluated in a similar way with the previous mentioned operations.

4.1.2 SUT Adapter (SA) and Platform Adapter (PA) entities

The SUT Adapter handles message and procedure based communication of the TTCN-3 test system with the SUT to the particular execution platform of the test system. It is responsible to propagate send requests and SUT action operations from the TTCN-3 executable (TE) to the SUT and to notify the TE of any received test events by enqueueing them in the port queues of the TE. It also realizes the procedure based communication with the SUT.

The Platform Adapter (PA) implements TTCN-3 external functions and provides a TTCN-3 test system with a single notion of time. All timers are to be implemented in this entity. Notice that although timers can be instantiated in the TE, they are implemented in the PA (to be more precise, the duration of the timers is implemented there).

4.1.3 The Component Handling (CH) Entity

Basically, the CH handles the communication between test components. It distributes TTCN-3 configuration operations like create, start and stop of test components, the connection between test components (connect and map), and intercomponent communication like send, call and reply between two TTCN-3 executables participating in the test session.
The CH is not implementing any kind of TTCN-3 functionality. Instead it will be informed by the TE that for example a test component shall be created (1 in fig. 5). Based on internal knowledge of the CH, the request for the creation of a component will be either transmitted to the local TE (2a in fig. 5) or to a remote participating one (2b in fig. 5) if the component has to be created on the remote TE. The remote TE will create the TTCN-3 component and will provide a handle back to the requesting (local) TE (3 and 4 in fig. 5). The requesting (local) TE can then operate on the remote created test component via the component handle given by the remote TE.

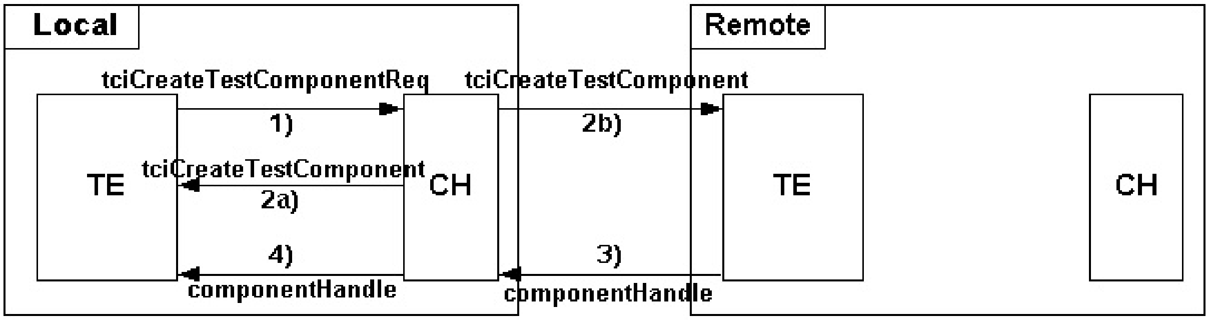


Figure 5. Local and Remote Component Creation

4.1.4 The Test Management (TM) Entity

The TM entity is responsible for the overall management of a test system. Within the TM entity, one can distinguish between test execution control and logging related functionality. The Test Control (TC) entity is responsible for general tester management and includes the implementation of a user interface and general test execution management, e.g. the preparation of the test system for a test execution, start of the test execution (1 in fig. 6), the collection of final verdicts (3 in fig. 6) and propagation of module parameters to the TTCN-3 Executable (2 in fig. 6). The Test Logging (TL) entity maintains the log of test execution.

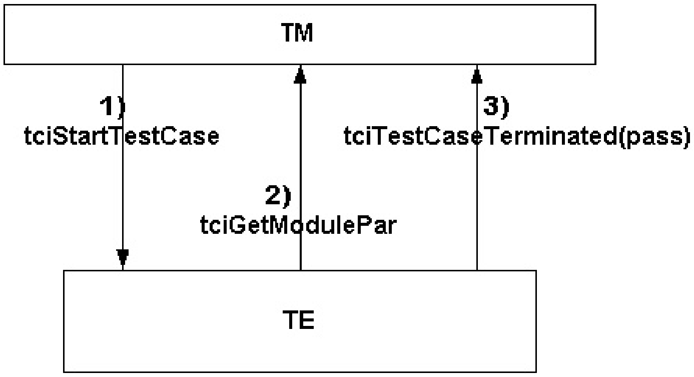


Figure 6. Test Management Entity

4.1.5 The Codec (CD) Entity

The CD entity is responsible for encoding of TTCN-3 values into bit strings in order to be sent to the System under Test. The TE determines which codecs have to be used and passes the TTCN-3 data to the appropriated codec in order to obtain the encoded data (1 and 2 in fig. 7) The received data is decoded back into the TTCN-3 values by the appropriate decoder (3 and 4 in fig. 7).

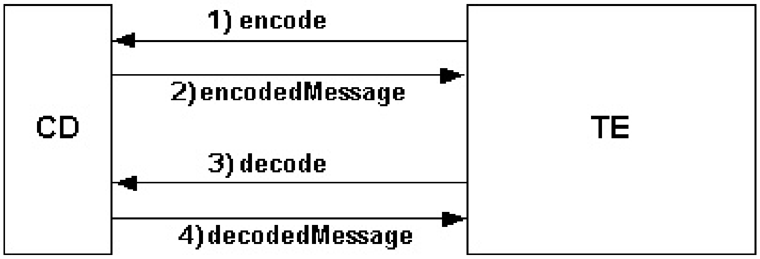


Figure 7. Codec Entity

4.2 GENERAL STRUCTURE OF A DISTRIBUTED TTCN-3 TEST SYSTEM

As the TE can be distributed among several test devices there must be an instance that implements the communication between the distributed entities. This instance is the CH. It provides the means to synchronize the different entities of the test system being potentially distributed onto several nodes (node has the same meaning as test device or host). The general structure of a test system distributed via several nodes is shown in figure 8.

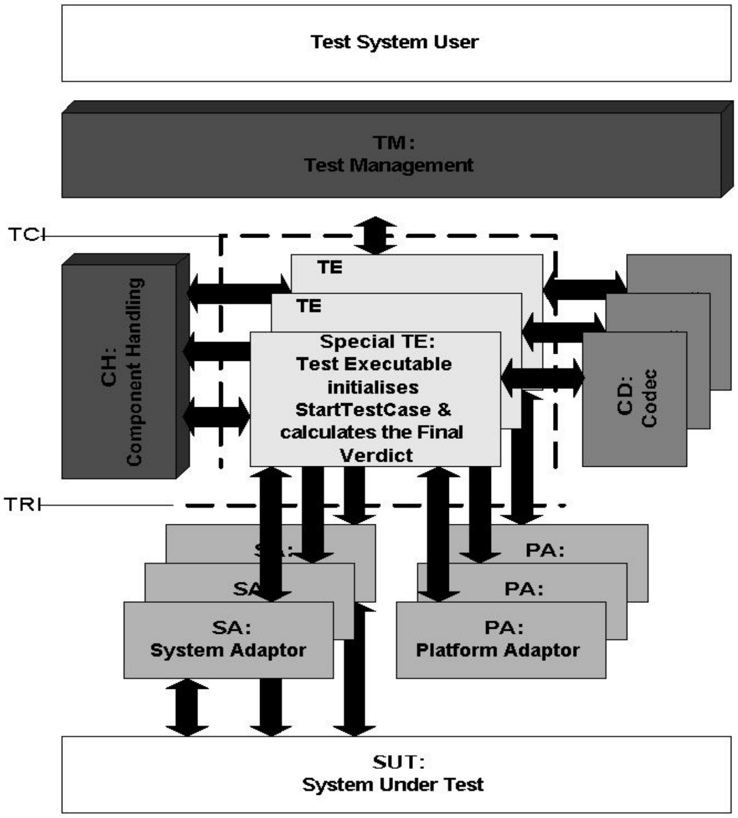


Figure 8. General Structure of a Distributed TTCN-3 Test System

Conceptually on each node, a TE together with SA, PA and CD exist. The entities CH and TM[1] mediate the test management and test component handling between the TEs on each node. There is a special TE that is identified to be the TE that started a test case and that is responsible for calculating the final verdict of that test case[2]. Besides this, all TEs are handled the same. The CH controls the creation of mtc, ptc and control components in TEs. Furthermore in TTCN-3 there is a system component that has a special role, as it exists only conceptually and not as a running test component in a TE (in fact it represents the SUT). System ports (i.e. the ports of the conceptual system component) may be distributed over several nodes. Further, test components on different nodes may have access to the same physical ports of the SUT. The access to the remote real SUT ports can e.g. be realized by TEs via local proxies.

Communication is on the one hand the message or procedure based communication between TTCN-3 components. Therefore, the CH adapts message and procedure based communication of TTCN-3 components to the particular execution platform of the test system. It is aware of connections between TTCN-3 test component communication ports. It is responsible to forward send request operations from a single TTCN-3 component that resides within a certain TE to the targeted component residing potentially in a

[1] Note: An implementation may realize the CH and TM in a distributed way, i.e. CH and TM may be distributed over all participating nodes.

[2] Note: The TM is aware of the special TE.

different instance of the same TE on a different test device. It then notifies the TE of any received test events by enqueueing them in the port queues of the TE.

Procedure based communication operations between TTCN-3 components are also visible at the CH. The CH has the task to distinguish between the different messages within procedure-based communication (i.e., call, reply, and exception) and to propagate them in the appropriate manner to the targeted component TE. TTCN-3 procedure based communication semantics, i.e., the effect of such operation on TTCN-3 test component execution, are to be handled in the TE.

On the other hand, there is additional test component management communication necessary in order to implement the distribution of test components between several test devices. This communication includes the indication of the creation of test components, the starting of execution of a test component, verdict distribution as well as component termination indication.

The CH does not implement the behavior of TTCN-3 components but the communication between several components that are implemented within the TE.

4.3 INTERFACES AND ADAPTORS

In order to execute a test, different adaptations have to be made, like the implementation of methods from the TTCN-3 runtime interface where the communication with the SUT has to be established and handled, or the implementation of methods from the TCI, e.g. methods needed to encode and decode TTCN-3 values to bitstrings and vice versa. Therefore, the methods from each interface will be presented shortly.

4.3.1 TCI

Each interface is divided into two subinterfaces (provided and required). The provided interface specifies operations that a test system shall provide to the TTCN-3 Executable and the required interface inversely the operations the TTCN-3 Executable to a test system. The terms „required“ and „provided“ reflect the fact that this specification defines the requirements on a TTCN-3 executable from a user point of view. The user, or maybe better, the implementor, „requires“ from a TTCN-3 Executable certain functionality to build a complete TTCN-3 based test system. To fulfil its task the TTCN-3 Executable has to inform the implementor on certain events where the implementor has to provide this possibility to the TTCN-3 Executable.

a) The TCI-TM interface

The TCI Test Management Interface (TCI-TM) contains operations a TTCN-3 executable is required to implement and the operations the test management implementation has to provide to the TE. It consists of the TCI-TM *Required* interface that specifies the operations the TM requires from the TE and the TCI-TM *Provided* interface that specifies the operations the TM has to provide to the TE.

The TCI-TM *Required* interface operations are used on the one hand to get information about the executing module. As the TE cannot know which module has to be used it has to be set first. This has to be done by using the method tciRootModule. After this, specific information can be resolved from this module on imported modules, parameters, test cases (parameters), test system interface ports: tciGetImportedModules, tciGetModuleParameters, tciGetTestCases, tciGetTestCaseParameters, and tciGetTestCaseTSI.

On the other hand, there are operations for handling start and stop of the control part or a specific test case: tciStartTestCase (starts a test case in the currently selected module with the given parameters), tciStopTestCase (stops the test case that is currently being executed), tciStartControl (starts the control part of the selected module), tciStopControl (stops the execution of the control part).

The operations of the TCI-TM *Provided* interface are used to inform the test management about the current status of the test executable. The TE can inform the management that a test case has been started (tciTestCaseStarted) or has been terminated (tciTestCaseTerminated) and also that the control part has been terminated (tciControlTerminated). It forwards the logs to the management where they can be displayed (tciLog) and it informs the management about errors occurred during the runtime (tciError). There is only one method that is used by the TE to get information from the TM and this is tciGetModulePar. As the user can change module parameters in the management the TE needs the opportunity to get these changes.

b) The TCI-CD interface

The TCI Codec Interface (TCI-CD) describes the operations a TTCN-3 Executable is required to implement and the operations a codec implementation for a certain encoding scheme shall provide to the TE. A codec implementation encodes TTCN-3 values according to the encoding attribute into a bitstring and decodes a bitstring according to the decoding hypothesis. The CD needs certain functionality from the TE in order to decode a bitstring into a TTCN-3 value. Inversely, the CD provides encoding and decoding functionality to the TTCN-3 Executable. The CD consists of the TCI-CD Required interface that specifies the operations the CD requires from the TE and the TCI-CD Provided interface that specifies the operations the CD has to provide to the TE.

The task of the TCI-CD Required interface is to return a specific type that can be used to create new instances of values. It has methods for getting predefined base types but it is also possible to get a type that has been defined in the TTCN-3 module. Methods for getting the predefined base types return type values representing a TTCN-3 types (e.g. *getFloat* returns a type value representing a TTCN-3 float type). It is also possible to get a type (also structured types) defined in the TTCN-3 module. Therefore, getTypeForName has to be used. When an unrecoverable error occurs in the CD, as well as in any of the other TTCN-3 Control interfaces, *tciErrorReq* has to be called and has to forward the error indication to the test management to inform it about this error situation.

The TCI-CD Provided needs only the following two methods: *encode* since values have to be encoded as they are abstract types and they cannot be understood by the SUT (they have to be encoded to a representation format that can be understood by the specific SUT; the method returns a TriMessage which is a bitstring), and *decode* is used whenever the TE has to decode an encoded value (the given message shall be decoded based on the given decoding hypothesis).

c) The TCI-CH interface

The TCI Component Handling Interface (TCI-CH) contains operations a TTCN-3 Executable is required to implement and the operations a component handling implementation has to provide to the TE. It consists of the TCI-CH Required that specifies the operations the CH requires from the TE and the TCI-CH Provided interface that specifies the operations the CH has to provide to the TE. As already stated, the TCI-CH interface handles the distribution of components and the appropriate configuration operations, e.g. if an component has to be created the CH decides on which TE it has to be done. Then the create request will be sent to the appropriate TE where the component will be created.

The interface which makes such decisions is the TCI-CH Provided Interface. The methods for handling components are: tciCreateTestComponentReq (called by the TE whenever a component has to be created, either explicitly when the TTCN-3 *create* operation is called or implicitly when the master or control component has to be created), tciStartTestComponentReq (TE has to execute start operation), tciStopTestComponentReq (TE has to execute the TTCN-3 stop operations).

Methods for the connection of test components and mapping of test components with the SUT are: tciConnectReq (to execute the connect operations), tciDisconnectReq (to execute disconnect operations), tciMapReq (to execute a map operation), tciUnmapReq (to execute the unmap operation).

The TCI-CH Provided interface is also responsible for forwarding message based and procedure based calls between components, i.e. only intercomponent communication is handled by the CH. Communication with the SUT is realized by the TRI. Appropriate methods are tciSendConnected (sending asynchronous message between two connected ports of two components), tciCallConnected, tciReplyConnected, tciRaiseConnected (to execute a call, reply or raise operation on a component port, which has been connected to another component port).

Other methods are needed for receiving the status of a component, i.e. if the component is running (tciTestComponentRunningReq) or if has finished its execution (tciTestComponentDoneReq). Also the termination of a component is shown to the appropriate TE by tciTestComponentTerminatedReq. As the MTC is maybe needed by a remote TE it can be requested by tciGetMTCReq. tciExecuteTestCaseReq is used to notify the remote TEs that have system ports of the indicated test case about the execution of the test case. This is done in order to set up static connections and the initialization of communication means for TSI ports. The last method tciResetReq is used to notify all participating TEs that an unrecoverable error has occurred and that they have to finish immediately their execution. It is also called when a test case is stopped by the user.

As the TCI-CH Provided interface has only the task to pass all method calls to the appropriate TE, the methods within TCI-CH Required interface are the same as in CH Provided. An example is the tciStartTestComponent method. It is the analog to tciStartTestComponentReq from TCI-CH Provided, and therefore, it has also the same parameters. The same happens for all methods from the CH Required interface.

For the communication methods (message and procedure based) different methods have been specified: *tciEnqueueMsgConnected* (enqueueing an asynchronous message into the local port queue of the indicated receiver component, called by the CH at the local TE when at a remote TE a provided tciSendConnected has been called), *tciEnqueueCallConnected* (enqueueing a call; called by the CH at the local TE when at a remote TE a provided tciCallConnected has been called), *tciEnqueueReplyConnected* (enqueueing a reply; called by the CH at the local TE when at a remote TE a provided tciReplyConnected has been called, *tciEnqueueRaiseConnected* (enqueueing an exception; called by the CH at the local TE when at a remote TE a provided tciRaiseConnected has been called).

4.3.2 TRI

The TRI interface consists of the two subinterfaces TriCommunicationSA and TriPlatformPA. The TriCommunicationSA interface consists of operations that are necessary to implement the communication of the TTCN-3 executable with the SUT. It includes operations to initialize the Test System Interface (TSI), establish connections to the SUT, and handle message and procedure based communication with the SUT. In addition, the TriCommunication interface offers an operation to reset the SUT Adapter (SA).

The *TriCommunicationSA* interface handles the communication from the TE to the SA, i.e. the SA has to implement the operation defined here: *triExecuteTestcase* (called by the TE immediately before the execution of any test case), *triMap* (SA can establish a dynamic connection to the SUT for the referenced TSI port, called by the TE when it executes a TTCN-3 map operation), *triUnmap* (SA shall close a dynamic connection to the SUT for the referenced TSI port, called by the TE when it executes any TTCN-3 unmap operation), *triSend* (SA can send a message to the SUT, called by the TE when it executes a TTCN-3 send operation on a component port, which has been mapped to a TSI port; called for all TTCN-3 send operations if no system component has been specified; the encoding of message has to be done in the TE prior to this TRI operation call), *triCall* (called by the TE when it executes a TTCN-3 call operation on a component port, which has been mapped to a TSI port; for all TTCN-3 call operations if no system component has been), *triReply* (called by the TE when it executes a reply operation on a component port which has been mapped to a TSI port; for all TTCN-3 reply operations if no system component has been specified), *triRaise* (called by the TE when it executes a raise operation on a component port; for all TTCN-3 raise operations if no system component has been specified), *triSutActionInformal* (called by the TE when it executes a SUT action operation, contains a string only), *triSutActionTemplate* (called by the TE when it executes a TTCN-3 SUT action operation, which uses a template, the encoding of the action template value has to be done in the TE prior to this TRI operation call), *triSAReset* (called by the TE at any time to reset the SA).

triEnqueueMsg, triEnqueueCall, *triEnqueueReply*, *triEnqueueException* are called by the SA after it has received a message/reply/procedure-call/exception from the SUT that has to be indicated to the TE.

The other interface, *TriPlatformPA*, mainly address the control of TTCN-3 external functions and timers and provides the following methods: *triExternalFunction* (operation is called by the TE; all in and inout function parameters contain encoded values; all out function parameters shall contain the distinct value of null since they are only of relevance in the return from the external function but not in its invocation; no error shall be indicated by the PA in case the value of any out parameter is non-null). *triStartTimer*, *triStopTimer* (called by the TE when a timer needs to be started/stopped), *triReadTimer*, *triTimerRunning* (called by the TE when a TTCN-3 read/running timer operation is to be executed). On the other hand *triTimeout* is called by the PA after a timer has expired. Finally there is *triPAReset* an operation that can be called by the TE at any time to reset the PA.

4.4 DISTRIBUTED EXECUTION ENVIRONMENT

The architecture of a possible realization of the distributed TTCN-3 Test System is displayed in the following figure. It has been named TTCN-3 Management and Execution platform and its base components are: containers (including test system entities TE, TM, CH, CD, SA and PA, daemons, session manager, and test console. CORBA is proposed as middleware for the communication between all these components.

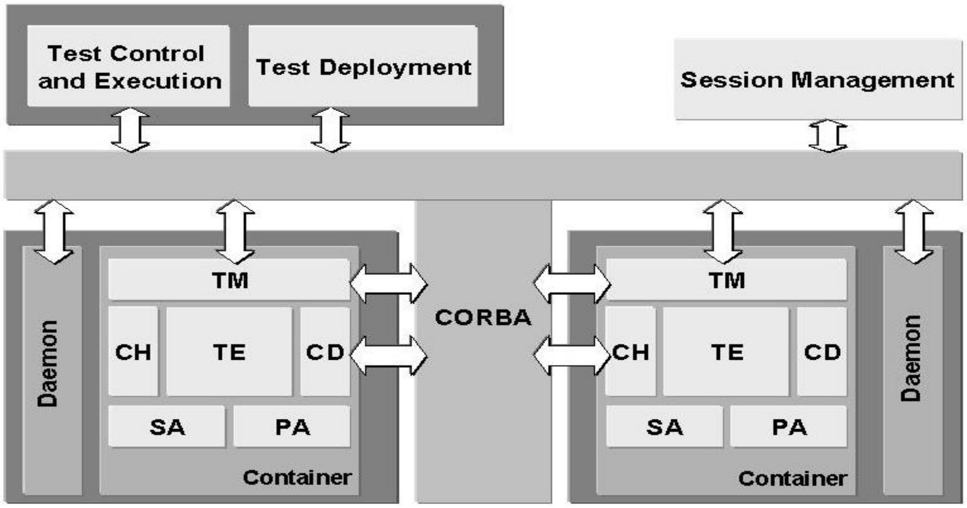


Figure 9. CORBA communication within the test system

4.4.1 Architectural elements

Containers are the heart of the whole architecture. They contain the different test system entities TE, TM, CH[3], CD, SA and PA and are used to handle the communication between them and the system entities on other nodes. Test components are created within the containers and are thus isolated from outside (a container is their target execution environment). The components cannot be accessed except via well defined interfaces. The information about created components and their physical locations is stored in a

[3] Note: As mentioned before conceptually CH and TM are single components, but for implementation reasons they may be distributed over all participating nodes.

repository within the containers. Containers are part of the middleware and are used for the communication between test components on different nodes.

The Test Console (illustrated by test control, excecution and deployment) is the control point of the communication platform and provides support to specify TTCN-3 test cases, create test sessions, deploy test suites and needed libraries into containers and control the test execution (start and stop of the test). It is also used for collecting of the logs from the different nodes.

Daemons are standalone processes installed on every host and used for the registration of a node to the test execution environment. They manage the containers which belong to different sessions.

To be able to execute multiple test sessions on a machine (either by one or by many users) the concept of session ids has been introduced. Whenever a new test session starts, a new id is created therefore and all containers participating to this test are associated with this id. Communication is allowed only between components with the same session id. As stated, there must be a central administration point that creates and manages these ids (and therefore, communication must be allowed also between this entity and the containers). This entity is the Session Manager.

The Session Manager is the centralized entity that has the global view of the test execution. On the one side it manages the different test sessions (creation and management of session Ids), and on the other side it processes the deployment configuration file and computes, based on the appropriate distribution algorithm, the destination node of a new component. Every CH has access to the session manager in order to get the needed information. It is also a central storage space for information like the location of the MTC or the Special TE of the session.

4.4.2 Supporting utilities

A middleware platform for the communication between distributed communicating test components has to be selected. Three architectures available at the market appears as the most suitable ones: OMG‘s Common Object Request Broker Architecture (CORBA), SUN’s Remote Method Invocation (Java RMI), and Microsoft’s Distributed Component Object Model (DCOM). CORBA is proposed as middleware for the communication between the containers on the different nodes for the following reasons:

- it is the standard architecture for distributed object systems,
- it allows to use clients and servers implemented in different programming languages (server in Java, client in C++); this is the main reason why Java RMI is not proposed,
- CORBA is an open standard defined by many global key players (DCOM is Microsoft proprietary),
- it is available for target operating systems Linux and Windows (DCOM is available only for Windows).

Furthermore a particular ORB, i.e. the distributed service that implements the requests to the remote objects, has to be selected. The Java 2 ORB is proposed since it is available without any licensing for the usage within industrial products, the whole TTthree runtime is specified in Java, it is a standard part of Sun's Java 2 SDK from version 1.4 and upward and can thus easily be integrated in the whole architecture, and it provides the needed CORBA services (Naming Service, Persistent State Service, Concurrency Service).

The introduction of a container configuration file (ccf) gives the possibility to describe the participating nodes and the sources that have to be deployed there. Every node might be identified by an IP address and a logical name. The logical name is used to simplify later needed referencing for a user. The sources that have to be specified include the ETS created by the TTthree compiler, the SA together with the appropriate codecs, and all other libraries needed by the test suite.

A test component distribution language (tcdl) is proposed to specify how components have to be distributed to the different containers. More concretely, it should allow specifying component descriptions (types), assemblies, mapping rules, and distribution algorithms

It is possible to choose between manual deployment (just to configure which type of component goes where) and automatic (constraints between components must be specified; the Session Manager processes the constraints and provides an adequate configuration).

4.5 LOGGING

Logging of test executions is a central element of the test infrastructure. Logging can be textually or graphically. In a first approach, a graphical logging by means of an open source TraceViewer developed by Lucent is investigated.

The focus of this tool is the graphical visualization of observations and activities that happened during the execution of TTCN-3 tests. The TTthree logging interface has been adapted to the Visualiser properties of the TraceViewer tool for TTCN-3 purposes.

The main goal is to show the client in the form of a web graphic all those processes, which exist during testing. We should also be able to distinguish between different types of components existing during test process. Generally there exists always the SUT as a Component being tested and additional Components like: mtc - Main Test Component, ptc -Parallel Test Component, control and so on. According to the specific role of the SUT every port to the SUT is represented separately. Moreover different operations implicated by TTCN3 code must be presented. These are to be shown in the Graphical User Interface.

The architecture is based on the Lucent Trace Viewer and Trace Server, however events to be visualized are produced due to the execution of TTCN-3 tests using a test manager (e.g. GrMuTTman, a GUI version of μTTman in the toolchain TTthree).

4.5.1 Technical Approach

The selection of the Lucent Trace tool for TTCN-3 test excecution purposes has been made due to its existing API that allows an easy access via CORBA, its extensibility for new graphics required for TTCN-3 purposes and its open source availability. Furthermore the viewer part of the tool that is based on a web browser offers excellent navigation and information filtering facilities.

The first part of the work address the connection of two independent software products, it is required to connect:

- the TraceViewer (i.e. TraceServer) with
- the logging interface of TTthree / GrMuTTman.

Excecuted TTCN-3 events in TTCN-3 test suite cause log events in an application of the logging interface, i.e. the implementation of EventLoggingImpl.java. By means of a CORBA connection (using e.g. Orbacus – as an easiest case) the Lucent trace server tool gets inputs.

The following figure presents an overview on this integration from the engineering viewpoint.

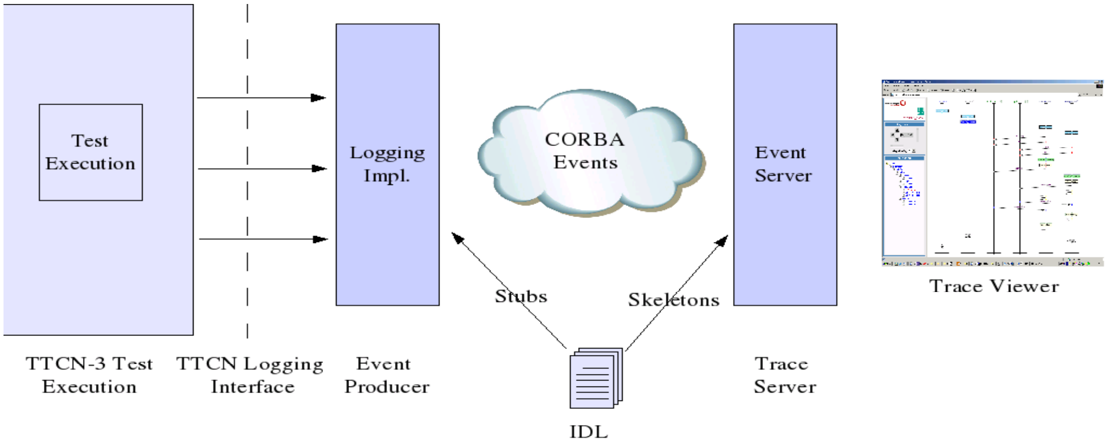


Figure 10. Implementation architecture

Besides the representation of the test components and ports several symbols are needed to present the operations done during the test case.

In the following the most important symbols used for the visualization of running test cases will be listed:

- the representation of component instances
- the creation, start and termination of test components
- the execution (start / stop) of test cases
- the timers within statements: start, stop, timeout.

Furthermore we provide a list of the symbols that will be also of major importance.

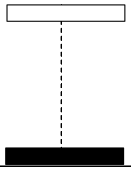
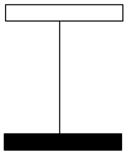
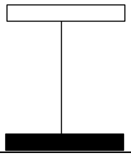
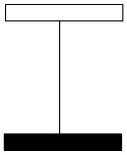
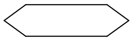
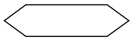
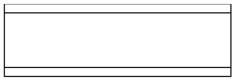
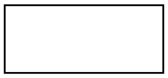
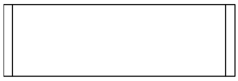
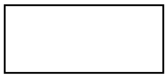
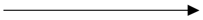
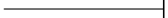
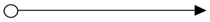
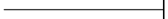
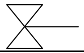
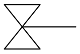
- verdict information
- start control
- stop control
- logStatements, which implicates the position of user in the program code (phase)
- particular ports of the System Under Testing
- name of the protocol
- protocol behaviour
- (un)map
- (dis)connect
- send message
- message enqueue
- template matching
- interruptions (e.g. ext. function calls)

According to the potential of the logging interface in particular cases (e.g. port mapping) a distinction was possible and has been made for the request of an operation (e.g. reqmap) and the operation itself (mapped). Due to the filtering facilities of the trace viewer the user is able to restrict the presented information to its particular interest.

The next steps of the approach combines Scalable Vector Graphics (SVG) in javascripts code. In particular the Viewer will be adapted to specific needs according to the particular TTCN-3 test behaviour events. SVG and javascripts are used so as to visualise the information provided by the trace server. What we want to develop, are new graphical templates so as to show the user the important parts of communication between the TTCN-3 components and SUT ports (System Under Test) in an appropriate manner.

Requirements analysis concerning graphical representation of particular parts considers the concepts of Graphical Test Specification (The Graphical Format of TTCN-3) symbols and Scalar Vector Graphics possibilities.

In the table below selected symbols of GFT elements in the relation to proposed SVG elements are shown.

GFT Element	Symbol	Description	SVG representation in the Viewer
Port instance symbol		Used to represent port instances	
Component instance symbol		Used to represent test components and the control instance	
Condition symbol		Used for textual TTCN-3 boolean expressions, verdict setting, port operations (start, stop and clear) and the done statement, to be attached to a component symbol	
Create symbol		Used for TTCN-3 create statement, to be attached to a component symbol	
Start symbol		Used for TTCN-3 start statement, to be attached to a component symbol	
Message symbol		Used for TTCN-3 send, call, reply, raise, receive, getcall, getreply, catch, trigger and check statement, to be attached to a component symbol and a port symbol	
Found symbol		Used for representing TTCN-3 receive, getcall, getreply, catch, trigger and check from any port, to be attached to a component symbol	
Start timer symbol		Used for TTCN-3 start timer operation, to be attached to a component symbol	

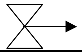
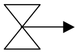
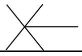
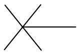
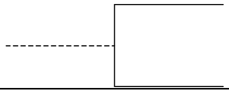
Timeout timer symbol		Used for TTCN-3 timeout operation, to be attached to a component symbol	
Stop timer symbol		Used for TTCN-3 stop timer operation, to be attached to a component symbol	
Event comment symbol		Used for TTCN-3 comments associated to events, to be attached to events on component instance or port instance symbols	Not implemented yet

Figure 11. GFT format versus SVG possibilities

We used rectangles for additional observations (like message enqueue) that are not covered in GFT. SVG offers a lot of dynamic possibilities, which have not been used because we considered that the animations did not improve the presentation of the events.

In the following figure the whole design concept of proper operations visualization is shown. A detailed view has been used which allows distinction of different ports even at the main and parallel test components.

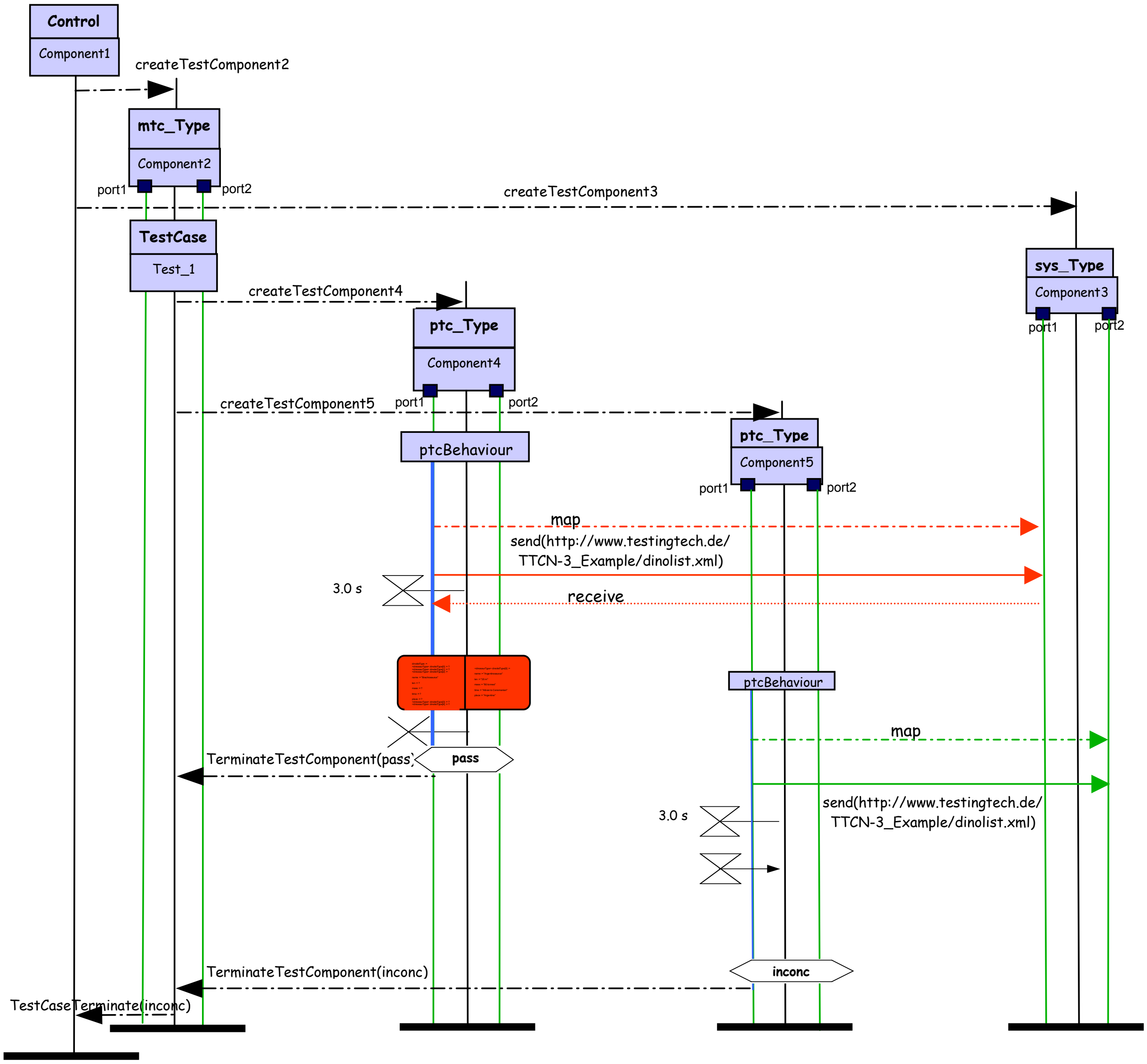


Figure 12. Design concept for TTCN3 tracing visualization

4.5.2 Implementation

The next figure address an example test suite (TTsuite-SIP) that has been used to illustrate the trace viewer features considering TTCN-3 with TTthree. It shows the execution traces of the test procedure. Requests in the form of TTCN3 operations are sent to the server.

Appropriate lifelines of the components are obtained. Later on, the components and their ports are created and the test case started, which causes presentation of arrows that are due to the map operation, indication about the start of the test components and their behaviour.

The timers are started (appropriate symbols) and stopped (relatively timeout) after enqueueing of a received message. Operations like matching of an incoming message and verdicts creation are due to the test specification. Test components and test case terminate with verdict settings.

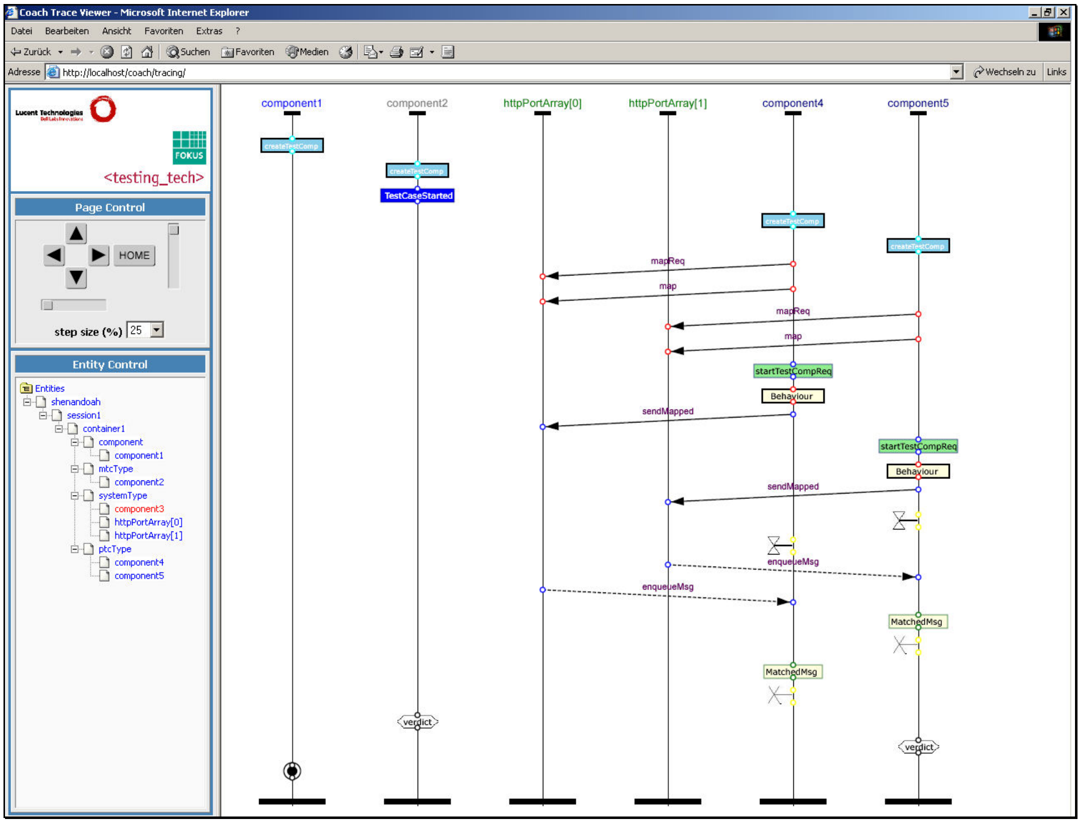


Figure 13. Snapshot from the TraceViewer implementation

4.6 IDL INTEGRATION

To support the testing of CORBA [11] based applications two main tasks need to be solved: Firstly, the IDL definitions of the interface of the SUT need to be made available when writing the test cases. This can be done by either importing the IDL definitions directly to the test suite or by first converting the IDL definitions explicitly to TTCN-3 and then to import the resulting TTCN-3 definitions. Secondly, procedure calls in the test suite that are executed need to relayed via some ORB to the SUT. Vice versa, the SUT must be able to call operations on the test system and these operation calls can then be accepted in the executing test cases by a `getcall` operation. This document focuses on test system execution, therefore we consider here only the second of the tasks: Supporting the exchange of procedure calls with the SUT.

Note that both tasks are supported by the TestingTechnologies/FOKUS tool `TTthree`.

4.6.1 Architectural Overview

To call a procedure on the SUT in the test system, the SA needs to contain an ORB. This is needed to establish communication at all between the SUT and the test system. In Figure 14. the ORB can be seen at the bottom of several layers. The ORB is directly connected to the SUT.

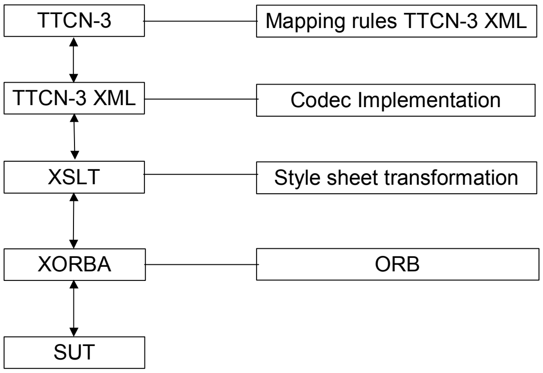


Figure 14. Architectural Overview of SA for testing CORBA applications

The ORB chosen in the SA is named XORBA [12], where the 'X' indicates that the ORB has an XML based API. This means that procedure calls, replies, and exceptions are exchanged with the application using the ORB in an XML defined format.

When looking at the upper part of these layers, then when executing a procedure call in the test suite the parameters of the test suite have to be encoded when they are passed from the TE to the SA. The TTCN-3 tool `TTthree` provides a default encoding of parameters that again is based on XML.

Unfortunately, these two XML schema definitions are not the same. An intermediate style sheet transformation is to used to convert the XML encoded data from one format to the other.

To support the decoding of incoming procedure calls, i.e. calls of the SUT on the test system, the decoder uses a description of the TTCN-3 signatures. These descriptions contain the name of the signatures and the types of their parameters, return values, and their exception types. This description is read from a file, which again can be generated by TTthree as a byproduct of importing the IDL definitions to TTCN-3 or of converting the IDL definitions to TTCN-3.

Note, that even some information about the signatures is read from file, the SA is a generic solution. Given some IDL definitions, there is no need to develop specific code to support the testing of applications hat either implement or use this interface.

4.6.2 Addressing Objects

To either call a procedure on the SUT, or to allow the SUT call a procedure on the test system, the SA must be able to route the call to the relevant objects. For this purpose, the SA implements the concept of *connectors*. A connector can be configured via an external function from within TTCN-3. The connector is then associated with a port at the test system interface. All subsequent calls from the test system on the SUT are forwarded to the corresponding connector. Vice versa, calls from the SUT on the test system will show up on one of this connectors and are then forwarded to the corresponding port at the test system interface.

At the moment of writing, connectors are implemented that utilize the CORBA naming service. These connectors are configured with the hostname and portnumber where a name server can be addressed, the name of a service, and the role of the object in the SUT. If the role of the SUT object is a client, then the service name used to configure the connector is the name under which the connector is registered at the name server. The SUT can then perform a lookup on the name server which object provides the needed service. If such an object – the connector in the SA – is found, the SUT can call a procedure on this object. This will then forward the call to the test system via the corresponding port. The call can be accepted in the test suite by a `getcall` statement, answer to the call can be provided by either `reply` or `raise` statements on the same port. The reply or exception will the be forwarded by the connector to the calling object. On the other hand, if the object in the SUT acts as a server, then the connector uses the service name to look up a reference of the object in the naming service and to forward procedure calls to this object.
A second implementation of a connector is currently under implementation, which does not use the naming service, but works directly with interoperable object references (IORs). More detail will be provided in the next revision of the report.

Note that in addition to using service names or IORs to address objects, both of them can be used as parameters of signatures. The name of a service can be exchanged between the test system and the SUT as well as IORs can be exchanged. While the actually exchange is easy, both service names and IORs can be defined in TTCN-3 as charstrings, there is also the need to get access to the IORs of the connectors. Especially in case that the test system contains a parallel test component that acts as a server, it might be needed that an IOR for this server is passed to the SUT and the SUT later on calls procedures on this server. Most naturally the IOR of the connector is used for this purpose. This means that the IOR of a connector must be made available in a test case. The details how this is achieved are not known at the moment because the connectors using IORs are currently implemented.

4.6.3 Exception Handling

Each IDL operation and its corresponding TTCN-3 signature can have a list of exceptions that can be raised by the server when processing a call. In addition to these explicitly defined exceptions in IDL definitions, each call in a CORBA application can result in one of a set of CORBA Exceptions. There are exceptions that are defined already in the CORBA standard, but further vendor specific Exceptions are also possible.

The SA must be able to handle these CORBA exceptions in various ways. Note that the SA at the moment of writing does not support such exceptions at all, therefore the remainder of this subsection presents the various ways how CORBA exceptions can occur in a test case.

In the first situation, assume that the test system calls a procedure on the SUT. Both the ORB of the test system as well as the ORB of the test system can identify some problem with the procedure call and raise one of the CORBA exceptions. In both cases the SA has to enqueue the TTCN-3 representation of a CORBA exception at the test system. The test case can then react correspondingly in whatever manner is considered correspondingly. One can imagine that in this case it would be sufficient that the SA does not raise an exception, but actually sets the verdict of the test case to `error`. Something has gone wrong, and most probably the test case cannot be continued in a meaningful way.

In the second situation, assume that the SUT calls a procedure on the SUT. Again, both the ORB of the test system or already the one of the SUT can raise a CORBA Exception. The exception would be provided to the calling object in the SUT, but the procedure call would not show up at all in the test system. This means that actually in this situation the exception does not become visible in the test case.

In the third scenario, both ORBs work fine, but in the test case a CORBA exception is raised explicitly to test e.g. error handling routines in the SUT. This kind of raising exceptions is similar to raising any of the explicitly defined exceptions of a signature. In this case, the SA has to forward the exception to the SUT. In this scenario, the CORBA exception must be explicitly named as an exception of the TTCN-3 signature. Otherwise the TTCN-3 code would simply be type incorrect.

4.7 XML INTEGRATION

There are various Systems under Test that describe the information to be exchanged using XML-based data type definitions (e.g. SOAP [15], MOST [13]). Therefore we need to elaborate an approach to support the integration of such information in the test system. In particular the mapping rules of XML data types, i.e. XML Schema definitions, to the Testing and Test Control Notation (TTCN-3) and a tool implementation for this task is introduced in the following.

4.7.1 XML to TTCN-3 Mapping

In [16] a generic approach for XML to TTCN-3 has been investigated. The main issues of this approach are the mapping of the XML built-in types to TTCN-3 types (e.g. decimal to TTCN-3 float), the consideration of XML facets (e.g. maxInclusive leads to TTCN-3 value restrictions), and the generation of structured TTCN-3 types from XML ComplexType specifications (e.g. choice is translated to TTCN-3 union). An auxiliary TTCN-3 module has been introduced for the mapping of all built-in types without facets to support short references within TTCN-3.

The structure of the mapping follows somehow the structure of XML Schema. First we look into the built-in datatypes, and afterwards into the components that the language offers.

Built-in datatypes are structured into primitive ones, and derived ones. The latter are derived from the primitive ones, by means of restrictions, like: length, size, list, range etc. These restrictions are called facets. For every simpleType, primitive or derived, facets can be applied, and a new type will be available. So for every built-in type there is a list of possible facets that can be applied to it, and depending on the facet, the correspondence in TTCN-3 is defined. Every built-in type, without any facets, is mapped with its built-in name, in a module called XSDAUX. And whenever a new simpleType is defined, with the base type a built-in one; it will be mapped using dot notation.

Example:

```
<xs:simpleType name="I">
    <restriction base="xs:integer"/>
</xs:simpleType>
```

Becomes:

```
type XSDAUX.integerXSD I;
```

Every built-in type is first mapped into the TTCN-3 definition of a simple type, and afterwards for the definition of the type, with every available facet. After mapping the basic layer of XML Schema, i.e. the built-in types, the mapping of the wrapping structures follow. For that, every structure that can appear, globally or not, will have a corresponding mapping into TTCN-3.

SimpleType components are used to define new simple types by three means: restricting a built-in type by applying a facet to it (as discussed in the previous section), building lists or unions of other simple types. SimpleTypes can be defined globally, which means the parent is <schema>, and the <name> attribute is mandatory, and they will be mapped to the new TTCN-3 type using that name. And so they can be reused in other definitions. Or they can be defined locally, i.e. the <name> attribute does not appear, so they will be mapped with an automatic generated name, but they will not be reused in other definitions. Here we will give just another small example[4]:

```
    <simpleType>
        <restriction base=”positiveInteger”>
            <minInclusive value=”8”/>
            <maxInclusive value=”72”/>
        </restriction>
    </simpleType>
    <simpleType name=”eRestriction”>
        <restriction base=”string”>
            <pattern value=”\d{3}-[A-Z]{2}”/>
        </restriction>
    </simpleType>
```

Becomes:

```
    type XSDAUX.positiveInteger simpleType__1 (8..72);
    type XSDAUX.string eRestriction (pattern "[0-9]#(3)-[A-Z]#(2)");
```

The complexType is used for creating new types that contain other elements and attributes. This is in contrast with the simpleTypes that cannot contain attributes. Just like simpleTypes, complexTypes can be defined globally, which means the possible parents are: <schema> and <redefine>. When they are defined globally, the "name" attribute is mandatory, so the new types will be mapped under the given value of that attribute. And they can be defined locally, in which case the name attribute cannot appear, they will be mapped using an automatic generated name, but they will not be used in other complexType definitions. In other words, they cannot be referenced from other definitions, as there purpose was locally.

For the mapping, the idea is to take separately every child that it can have, and assign the corresponding TTCN-3 code. For example the *complexType* can hold optional an *annotation* and after that follows the content. This content could be *simpleContent* element, *complexContent* element or a definition. The *simpleContent/ complexContent* describe a restriction or an extention of an existing *simpleType/ complexType*. The definition for example can be specified with a *group*, *choice* or an *sequence* and some attributes.

4.7.2 Implementation approach

There is also a prototype implementation available that has been integrated into the TTCN-3 compiler TTthree. The implementation allows both an implicit mapping of XML structures into an internal tree representation of the XML definitions and also the explicit translation into TTCN-3 core notation syntax due to the application of some TTthree printing features in a second step.

The translation from XMLSchema to TTCN-3 could have been a stand-alone tool, but it is needed for the import handle into TTCN-3, so it has been included in the TTthree compiler. For the integration in the TTthree compiler, the mapping is divided in two parts: "implicit" mapping, that translates XMLSchema to the TDOM (Typed Document Object Module) structure, and "explicit" mapping that translates from the TDOM structures to TTCN-3 syntax. The first mapping is obtained from the XMLSchema parser that will be defined below, and for the second one is used another tool, also integrated in the TTthree compiler,

[4] It is recommended to use the name attribute since a reorderinig of definitions may lead to renamed type definitions.

Syntax2Template.java. The TTthree tool is a Java application that comes with its own Java Runtime Environment (JRE 1.3.1) and Java compiler (IBM's Java compiler Jikes v. 1.16).

Creating the TDOM is one step in the import of XSD types into TTCN-3. The new types are assigned to the tree, and any mistake in the compilation will be noticed if events are not sent in the right order. But to verify that the translation from XMLSchema to TTCN-3 was correct, it would be too clumsy just to watch the tree. So a tool (Syntax2Template.java) has been used, for translating from TDOM to TTCN-3 syntax. And the verification of the translation now can be done really easy. So the short version is: from XMLSchema to TTCN-3 TDOM, in the TTthree compiler, and from TDOM to TTCN-3 syntax, using Syntax2Template.

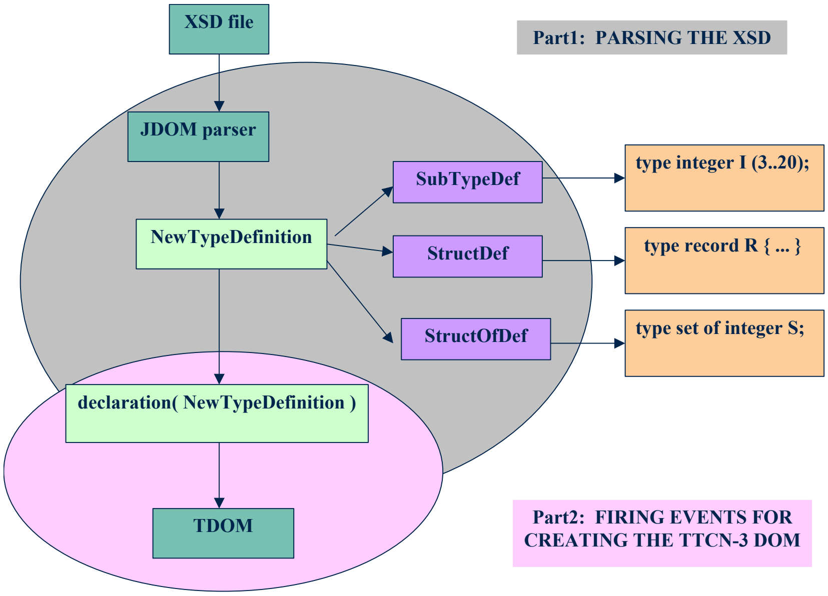


Figure 15. Architectural Overview on XSD integration in TTthree

4.7.3 XML processing prior to TTCN-3 Mapping

In the following section a case study is presented to explain that the pure XML to TTCN-3 mapping – as described before - may be not sufficient in some industrial domains for the test suite development process even if there is an XML-based data type description available. The MOST functional catalog (Fcat) from the automotive case study has been selected as an application of the XML mapping implementation since it comprises the definitions of MOST devices and functions in standardized XML documents. Each MOST function may have different operation types with a related parameter list. Beside the English text version and different printed tables of the catalog the MOST cooperation provides two machine readable documents of Fcat: a DTD file on the structure of function blocks and function interfaces (fcat.dtd) and a corresponding data file most.xml which gives information on the concrete function signatures and parameter types.

It is important to be aware about the distribution of the Fcat information among the two files. The type definition file provides a tree structure describing the set of MOST functional blocks and functions on a high abstraction level, i.e. it introduces the scope of information related to all MOST functions without addressing any particular function (identifier, parameters etc.). Beside some general elements in this tree about the identification and version of a function the main structure addresses the possibility of function properties and methods. Especially the latter introduces e.g. different operation types for method commands and reports. Furthermore it provides the list of allowed parameter types in MOST but not the relation of parameters to functions.

Function and parameter names and its textual descriptions as well as the details on the parameter types (e.g. value scope restrictions) are part of the XML file only. The XML file contains the information on any parameter position within a MOST function. Furthermore it provides some information for the protocol data coding (e.g. FBlockID, FunctionID).

It is obvious that testing MOST-based communication has to consider the data structures given in the DTD and XML files. It should be possible to address the Fcat contents within the TTCN-3 test suite behaviour and data templates. Therefore an automatic translation of the MOST Fcat into TTCN-3 is required.

To get the most flexibility a general approach for the MOST Fcat transformation has been selected: First, the DTD definition will be translated into the popular XML schema definition (XSD) format and secondly a generic XSD to TTCN-3 translation needs to be performed. To translate the DTD definition one can use any DTD to XML Schema Converter (e.g. the freely available DTD to XML Schema Converter “dtd2xs” from Syntext). For the transformation to TTCN-3 we’ve applied our XSD to TTCN-3 translator. Due to the nature of the MOST Fcat contents the resulting TTCN-3 data types have been quite complex for human reading and a modified approach for the integration and usage of the MOST definitions will be proposed as follows:

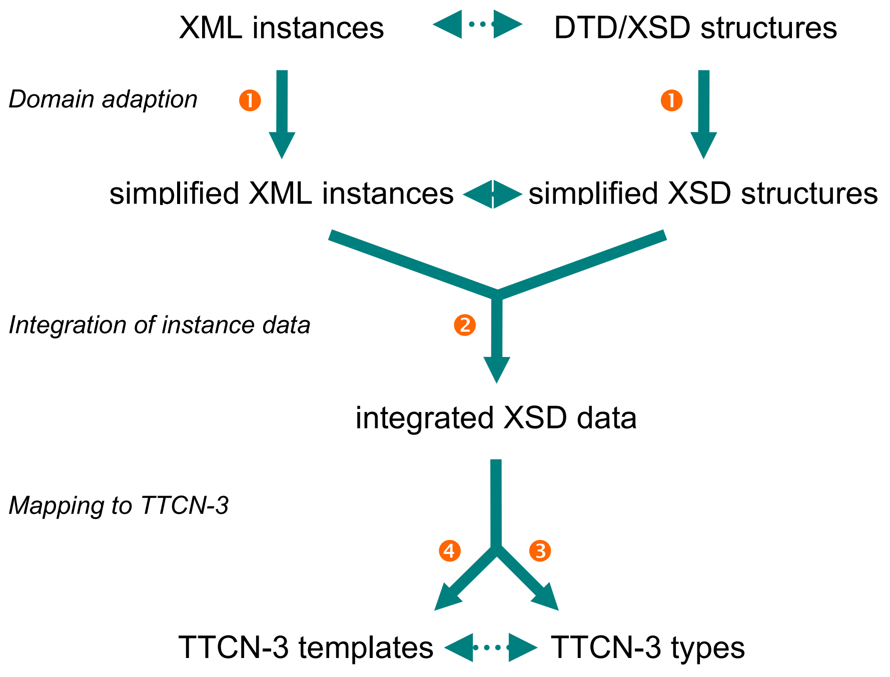


Figure 16. MOST mapping approach

Following the approach illustrated in Figure 16. four steps have to be considered: ❶ some restriction of the DTD/XSD structure to those data elements that are observable during testing, ❷ an integration of XML data into the DTD/XSD structure definition, ❸ the mapping to TTCN-3 data types that will be used for the declaration of data templates, and ❹ the generation of parametrisied TTCN-3 data templates.
Step ❶ and ❷ describe the process prior to the TTCN-3 Mapping. Domain adaption comprises four internal actions: Strip XML data from futile ballast (e.g. remove documentation elements), unify elements with aquivalent structure, decide for every element if it should defined globally or locally and simplify structures by mapping them to basic XML Schema types. The Integration of instance data has two main tasks: use the protocol structure from MOST and flatten the structure by using attributes and combining multiple elements. In principle these transformations of XML data structures are done automatically, i.e. only step ❶ for XSD structures is done manually.

In the following we present MOST mapping approach by a fragment from the AmFmTuner function block:

```
<!-- Function: Notification -->
 <Function>
  <FunctionID FunctionSection="Coordination">0x001</FunctionID>
  <FunctionName>Notification</FunctionName>
  <FunctionDescription>[…]</FunctionDescription>
  <FunctionVersion Access="public">[…]</FunctionVersion>
  <FunctionClass ClassRef="class_unclassified_property">
     <FunctionClassDesc>Unclassified Property </FunctionClassDesc>
     <Property>
      <PUnclassified Length="1">
       <!-- Control -->
       <PUParam>
        <ParamName ParamIdx="1">Control</ParamName>
        <ParamDescription>[…]</ParamDescription>
        <PUParamOPType>
           <PUCommand>
            <PCmdSet OPTypeRef="PCmdSet">
             <ParamPos>1</ParamPos>
            </PCmdSet>
           </PUCommand>
           <PUReport/>
        </PUParamOPType>
        <PUParamType>
           <TEnum TypeRef="type_enum" TEnumMax="3">
            <TEnumValue Code="0x00">SetAll</TEnumValue>
            <TEnumValue Code="0x01">SetFunction</TEnumValue>
            <TEnumValue Code="0x02">ClearAll </TEnumValue>
            <TEnumValue Code="0x03">ClearFunction</TEnumValue>
           </TEnum>
        </PUParamType>
       </PUParam>

      … more PUParam definitions …

      </PUnclassified>
     </Property>
  </FunctionClass>
 </Function>
```

After preprosessing, the Function *Notification* in the integrated XSD data has some operation elements like *AmFmTuner_Notification_PCmdSet* and all used type definitions (e.g. *AmFmTuner_Control_Type*). An important factor is, that protocol and definition structure have been reduced to a nesting depth less then 5:

```
 <xs:element name="AmFmTuner_Notification_PCmdSet">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="FBlockId" type="MOST_FBLOCK_ID" fixed="0x40" />
    <xs:element name="InstanceId" type="MOST_INSTANCE_ID" />
    <xs:element name="FunctionId" type="MOST_FUNCTION_ID" fixed="0x001" />
    <xs:element name="OpType" type="MOST_OPTYPE" fixed="0x0" />
    <xs:element name="Control" type="AmFmTuner_Control_Type" />
    <xs:element name="DeviceID" type="MOST_UWORD" />
    <xs:element name="FktIDList" type="AmFmTuner_FktIDList_Type" />
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:simpleType name="AmFmTuner_Control_Type">
  <xs:restriction base="MOST_UBYTE">
   <xs:enumeration value="0">
    <xs:annotation>
     <xs:documentation>SetAll</xs:documentation>
    </xs:annotation>
   </xs:enumeration>
   <xs:enumeration value="1">
    <xs:annotation>
     <xs:documentation>SetFunction</xs:documentation>
    </xs:annotation>
   </xs:enumeration>
   <xs:enumeration value="2">
    <xs:annotation>
     <xs:documentation>ClearAll</xs:documentation>
    </xs:annotation>
   </xs:enumeration>
   <xs:enumeration value="3">
    <xs:annotation>
     <xs:documentation>ClearFunction</xs:documentation>
    </xs:annotation>
   </xs:enumeration>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="AmFmTuner_FktIDList_Type">
  <xs:list itemType="MOST_UWORD">
   <xs:maxLength value="8" fixed="true" />
  </xs:list>
 </xs:simpleType>
```

The resulting TTCN-3 data types and templates have been produced with the XSD to TTCN-3 translator and can now be referenced in test cases. Every Operation has one element definition with the protocol structure and a corresponding parametrized template:

```
type record AmFmTuner_Notification_PCmdSet__Element {
 MOST_Types.MOST_FBLOCK_ID FBlockId,
  MOST_Types.MOST_INSTANCE_ID InstanceId,
  MOST_Types.MOST_FUNCTION_ID FunctionId,
  MOST_Types.MOST_OPTYPE OpType,
  AmFmTuner_Control_Type Control,
  MOST_Types.MOST_UWORD DeviceID,
  AmFmTuner_FktIDList_Type FktIDList
}

type MOST_Types.MOST_UBYTE AmFmTuner_Control_Type (0, 1, 2, 3);
type set of MOST_UWORD AmFmTuner_FktIDList_Type;

template
AmFmTuner_Notification_PCmdSet__Element AmFmTuner_Notification_PCmdSet__Template
(
  template MOST_Types.MOST_INSTANCE_ID param_InstanceId,
  template AmFmTuner_Control_Type param_Control,
  template MOST_Types.MOST_UWORD param_DeviceID,
  template AmFmTuner_FktIDList_Type param_FktIDList
) :=
{
  FBlockId := '40'H,
  InstanceId := param_InstanceId,
  FunctionId := '001'H,
  OpType := '0'H,
  Control := param_Control,
  DeviceID := param_DeviceID,
  FktIDList := param_FktIDList
}
```

4.8 C++ TO TTCN-3 INTEGRATION

In the following subsection, we provide an introduction to the C++ to TTCN-3 language mapping by describing the C++ elements to be mapped in the course of the TT-Medal project. To illustrate the mapping, an example is provided, as well. The second subsection outlines an architecture for TTCN-3 test systems to test C++ components and interfaces in a platform independent way. In the third subsection, a realization of a predefined adaptation is discussed.

4.8.1 C++ to TTCN-3 Mapping

A typical test execution harness for C++ based component testing includes entities to call the functions of component under test (i.e., test drivers) and to simulate the non-existent software components (i.e., stubs), which the component under test depends on. To directly access the provided and required interfaces of the component under test, a TTCN-3 based test system requires representation of these interfaces. An approach to represent the interface at the TTCN-3 language level is to define an explicit mapping, e.g., C++ header file elements are translated into TTCN-3 language elements by means of dedicated mapping rules. The represented interfaces can then be used by TTCN-3 test cases to call the functions of C++ component under test (CUT) and to simulate non-existent C++ software components. The mapping to be defined in the course of the TT-Medal project shall follow the explicit mapping, which should include the items defined in Table 4.1.

Table 4.1. C++ Header file elements

Named namespaces	**namespace** N { /* ... */ }
Type definitions	**struct** Point {**int** x, y;}
Template declarations	**template**<**class** T> **class** Z;
Template definitions	**template**<**class** T> **class** V {/*...*/};
Function declarations	**extern int** strlen(**const char***);
Inline function definitions	**inline char** get(**char*** p) {**return** *p++;}
Data declarations	**extern int** a;
Constant definitions	**const float** pi = 3.141593;
Enumerations	**enum** Light {red, yellow, green};
Name declarations	**class** Matrix;
Include directives	**#include** <algorithm>
Macro definitions	**#define** VERSION 12
Conditional compilation directives	**#ifdef** __cplusplus
Comments	/* check for end of file ***/**

An example of C++ to TTCN-3 mapping is provided in Table 4.2. In the example, a CFile class is mapped to TTCN-3 following mapping rules defined in [25].

Table 4.2. An example of C++ to TTCN-3 mapping

C++	TTCN-3

class CFile : **public** CObject {
public:
	module CFile {
	// include definitions from “super class”
	import from CObject **all**;
	// include C++ basic types e.g., CppInt
	import from Cpp **all**;
	// include definitions for types that are
	// implied by CFile.h e.g., CString and
	// UINT
	import from OtherTypes **all**;

// member functions	// member functions are mapped to TTCN-3
	// signatures
	// - constructor returns a unique value
	// for a new object
	// - DefaultException is used if no
	// exception is declared by the C++
	// function in question
	// - inherited functions are defined
	// explicitly

CFile();	**signature** CFile_no_params()
	return CFilePtr
	exception(DefaultException);

CFile(LPCTSTR lpszFileName, UINT nOpenFlags);	**signature** CFile_two_params(**in** LPCTSTR lpszFileName, **in** UINT nOpenFlags) **return** CFilePtr **exception**(DefaultException);
virtual UINT Read(**void*** lpBuf, UINT nCount);	**signature** Read(**in** CFilePtr this /* the object */, **in** CppPtr lpBuf, **in** UINT nCount) **return** UINT **exception**(DefaultException);
static BOOL GetStatus(LPCTSTR lpszFileName, CFileStatus& rStatus);	**signature** GetStatus(/* no this */ **in** LPCTSTR lpszFileName, **in** CFileStatusRef rStatus) **return** BOOL **exception**(DefaultException);
virtual ~CFile();	**signature** DelCFile(**in** CFilePtr this) **exception**(DefaultException);
	// “inherited” **signature** CObject_IsSerializable(**in** CFilePtr this) **return** BOOL **exception**(DefaultException);

	// we need to declare the port for the
	// signatures
	type port CFilePort **procedure** {
	// test system makes the calls -> out
	out CFile_no_params,
	out CFile_two_params,
	out Read,
	out GetStatus,
	out DelCFile,
	out CObject_IsSerializable
	}

// enums
enum OpenFlags {
modeRead = 0x0000,
modeWrite = 0x0001
}
	// enums are mapped to constant integers
	const CppInt modeRead:= **hex2int**(‘0000’H);
	const CppInt modeWrite:=**hex2int**(‘0001’H);

// member variables
UINT m_hFile;
 	
protected:
// member variables
BOOL m_bCloseOnDelete;
CString m_strFileName;
};
	// member variables are represented by
	// a record type. also the super class
	// needs to be included
	type record CFileType{
	UINT m_hFile,
	BOOL m_bCloseOnDelete,
	CStringType m_strFileName,
	CObjectType super

	}
	
	// we need to map the C++ pointer
	// operators. signatures or external functions
	// can be used. utilizing external functions
	// reflects the fact the these functions are not
	// “tested” but rather they used to set and
	// retrieve the test data
	type octetstring CFilePtr;
	external function NewCFile()
	    return CFilePtr;
	external function SetCFile(
	    in CFilePtr this,
	    in CFileType newValue);
	external function GetCFile(
	    in CFilePtr this)
	    return CFileType;
	external function DeleteCFile(
	    in CFilePtr this);
	}

The aim of the standardized mapping rules is to provide rules where the interface of CUT can be represented one to one with TTCN-3. This result in a similar usage of TTCN-3 compared with a situation where test suites are implemented with a programming language and therefore supports wide variety of test cases. Sometimes it is reasonable to represent types on a more abstract level than depicted by the mapping rules although this might lead to situation where the automatic generation of static part of TTCN-3 test suites, which relies on unambiguous information, is no longer possible. Also, increasing the abstraction level leads in a situation where all the details cannot be tested since the information is hidden into the adaptation layer

The abstract rules cannot be standardized as they derive their requirements from test purposes which are always case specific. Nevertheless, the abstract presentation of types enhances flexibility, reusability and readability of the test suites and makes the test suite development more efficient as the most excruciating details are not included in the test suites. This should be more beneficial compared to automatic generation of the information. For instance, mapping a `vector<MyClass*>` to TTCN-3 could lead to unnecessary complicates TTCN-3 code although the test suite writer only needed an array of classes which can be mapped as `type record of MyClass`. Table 4.1 illustrates this.

Table 4.1. A simplified C++ to TTCN-3 language mapping

Standardized mapping	Abstract mapping

testcase MyTestCase() runs on MyTester {
  :
  // the functions used in this test case are
  // external or TTCN-3 functions. the TTCN-3
  // functions shall further utilize signatures
  // or external functions to address its
  // purpose. code excludes also error handling
  // code
 	
  // create the pointers
  var VectorMyClassPtr list :=
      ef_NewVectorMyClass();
  var MyClassPtr myClass := ef_NewMyClass();
  var MyClassPtr myClass2 := ef_NewMyClass();
  // set the fields of myClass and myClass2
  ef_SetMyClass(myClass, {1, 3.14});
  ef_SetMyClass(myClass2, {0, 3.14});
  // add the myClass and myClass2 to the list
  f_AddVectorMyClass(list, myClass);
  f_AddVectorMyClass(list, myClass2);
  // call the function under test
  myPort.call(s_MyFunction:{list}, nowait);
  :
	testcase MyTestCase() runs on MyTester {
	  :
	  var MyClassArray list :=
	      {{1, 3.14}, {0, 3.14}};
	  // call the function under test
	  myPort.call(s_MyFunction:{list}, nowait);
	  :

This same concept can be also applied to functions by representing the signatures on more abstract level. Instead of mapping the signatures one to one, one should consider whether the types of parameters, return values, and exceptions could be depicted in one way that satisfies many usages to enhance the reusability. An example of this is where same function is implemented both with Java or C++ and one common representation is described in TTCN-3 code for these functions.

4.8.2 TTCN-3 Test System for C++ Components

In order to integrate C++ based software with the Java based TTthree infrastructure, the integration defines a distributed TTCN-3 runtime interface where the test system does not share memory with the CUT, but the communication between the test system and the CUT is realized by the means of an interprocess communication mechanism. As a matter of fact, this architecture can be utilized to test C++ components whose functionality is implemented over many operating system processes (i.e., the CUT is also distributed). The architecture is depicted in Figure 17.

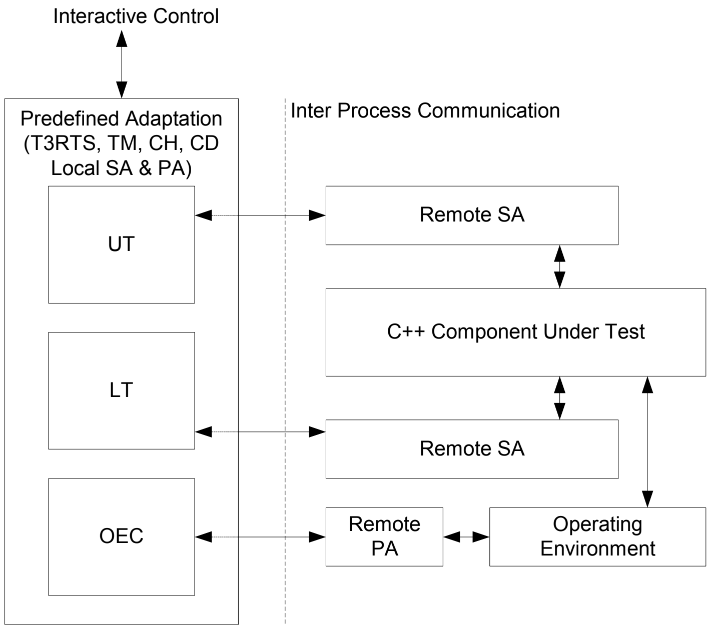


Figure 17. Test system architecture for C++ component testing

The test system includes an Upper Tester (UT) to control and observe the upper interface and the Lower Tester (LT) is for the lower interface of the Component Under Test (CUT). The UT and the LT roughly correspond to driver functionality and stub functionality of general purpose software testing, respectively. However, UT can receive calls from the CUT (e.g., call backs) and LT can stimulate the lower interface of CUT. Therefore UT and LT can contain also stub and driver functionality. Also, a distinction between LT and operating environment control (OEC) is made. Whereas OEC provides control and observation of the real operating environment of the CUT, LT is used to simulate non-existent components. In other words, the OEC only interacts with operating environment which further interfaces with the CUT while the LT directly interfaces with the CUT. UT, LT, and OEC are implemented using TTCN-3 test language where C++ to TTCN-3 mapping rules are utilized to represent of the required and provided interfaces of CUT.

The left hand side of the figure includes features to execute test cases (T3RTS), control and monitor test execution (TM), and report results (TM). CH is used to administer test components locally. Local SA, and PA support the distribution of the TTCN-3 runtime interface over test system process (i.e., left hand-side of Figure 17) and CUT process(es) (i.e., right hand-side of Figure 17). CD is used to define encoding and decoding mechanisms for TTCN-3 values to be sent between test system and system under test processes.

Remote SAs are responsible for calling the functions of CUT and also provide functions that are called by the CUT. For instance, when a UT makes a call, the remote SA is activated to convert encoded values into C++ values and call the function of the CUT. This in turn might generate a call back to the UT and a call to the LT. In this case, remote SAs are called by the CUT, and the C++ values are converted to the values CD is able to decode. Finally, the call is forwarded back to LT or UT by the remote SAs. Of course, this kind of sequence can be activated in reverse order by LT.

OEC utilizes remote PA to control and monitor the real operating environment of CUT. For instance, the state of the operating environment can be set-up to interesting values prior to making the function calls. Alternatively, remote PA might provide an external function to set-up the function parameters of the CUT which in turn are utilized by UT or LT in down or up calls. For example, C++ to TTCN-3 language mapping uses external functions to represent pointers. These external functions are called when function parameters which have type of pointer are initialized and passed to function under test.

4.8.3 Approach to Realize the Adaptation

The TTCN-3 test system for C++ component testing should consist of generic adaptation components. Only the remote SAs and PAs are the interface specific elements and therefore need to be implemented for each C++ interface and for each operating environments of CUT which are accessed from TTCN-3 tests. In order to test distributed CUTs, remote SAs and PAs need to be implemented in each operating system processes where parts of CUTs are executed. The parts of the remote adapter can be generated from header files of C++ programs instead of manually writing them. However, implementing the specific parts manually provides necessary flexibility, reusability, and readability for test system and test cases development which cannot be necessarily achieved with generation techniques as discussed previously. Table 4.4 describes the reusable elements in TTCN-3 test system in order to enable C++ component testing.

Table 4.4. Realization of predefined adaptation for C++ component testing

Test system entity	Functionality
T3RTS	Executes UT, LT, and OEC by implementing parts of TCI and TRI (TE) interfaces. T3RTS is always realized by TTCN-3 tooling.
TM	Controls and monitors (i.e., logging) test execution. This should be included in TTCN-3 tooling.
CH	Administrates test components within a test system process. This should be included in TTCN-3 tooling.
CD	Encodes TTCN-3 values to format which can be passed over inter process communication channel to remote SAs and PAs. Decodes values from remote SAs and PAs to TTCN-3 values. This encoding is described below the table.
Local SA	Supports distribution of TRI procedural based operations to multiple remote SAs. This can be realized by utilizing TT-Datagram protocol.

Local PA	Supports distribution of external functions to multiple remote PAs. This can be realized by utilizing TT-Datagram protocol.

The remote SAs and PAs shall include generic elements such as parsing the incoming encoded values and constructing values to be decoded by a CD. The generic elements shall also provide convenient means to send and receive information over inter process communication channel. Furthermore, these generic parts should provide an API for test system developer to implement the specific part of the remote adapter. To utilize advanced reuse techniques, the generic parts of the remote adaptations can be utilized in multiple test systems and interface specific parts, which are left for test system developer, are introduced as extension to those generic parts. The API offered for the extensions by the generic part provides means to register the specific part in question into the generic part, calls appropriate signature adapter which does the actual function call, and implements functions to extract C++ values from encoded TTCN-3 values, create return values or exceptions from C++ values and enqueue calls and parameters back to TTCN-3 tests in cases of stubs. This facilitates efficient TTCN-3 test system adapter development for C++ component testing. The enhanced architecture is depicted in Figure 18.

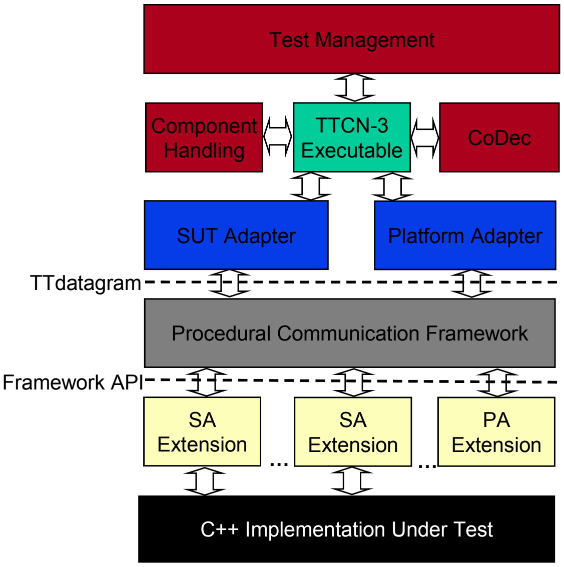


Figure 18. Refined test system architecture for C++ components

Encoding

Signature parameters, return values, and exceptions are exchanged between local- and remote part of the test system as byte (8 bit) strings. These values are sent and received between local- and remote parts of the test system and therefore mutual consistent encoding rules are needed. TT-Datagram protocol defines the rules for exchanging TRI related information, but exchanging the actual test data is yet to be defined. In this section we define encoding rules to send and receive test data between local- and remote parts of the test system. This encoding shall be implemented by the generic CD and also the generic part in the remote adapter.

The encoded byte string includes a type identifier element (TypeID), a field identifier element (FieldID), a field count element (FieldCount) and a value element (Value) as depicted in Table 4.5 below.

Table 4.5 Encoded TTCN-3 value

TypeID Length Value	FieldID Length Value	FieldCount 32 bits	Value Length Value

- The length field found in TypeID, FieldID, and Value elements has size of four bytes in big-endian byte order.

- Type identifier element includes a value field which is a US-ASCII encoded string of the type name defined in a TTCN-3 module and a length field which defines the variable length of the value.

- Field identifier element includes a value field which is a US-ASCII encoded string of the field name in a TTCN-3 type and a length field which defines the variable length of the value.

- Field count element defines the number of fields in a structured type. This element includes four bytes in big-endian byte order. 2^32 is used to indicate that the value is a simple type and therefore contains no fields.

- In case of simple types, a value element includes a value field which is an US-ASCII encoded string of the TTCN-3 value and a length field which defines the variable length of the value.

- In case of structured types (i.e., record, union, and array), a value element includes a byte string according to these encoding rules. I.e., the length field has size of four bytes in big-endian byte order which defines the variable length of the value field and the value field is itself encoded as defined the above encoding rules. Enumeration and set types do not need to be encoded as the C++ to TTCN-3 language mapping do not utilize them.

4.9 TT-DATAGRAM PROTOCOL

Following the presented TTCN-3 infrastructure there is a need in providing a bridge between a system adapter (SA) and the actual SUT interface in case of some technical incompatibility with the operating system, the programming language API etc. To overcome such issues a TTCN-3 datagram protocol is proposed and introduced in the following.

The TTCN-3 system architecture is a well defined architecture describing the different parts needed by a TTCN-3 based test system, e.g. the TRI. Normally all these parts are developed on a single platform using a certain programming language. But there are no well defined ways described to distribute the different parts in a network. For the adaptation to a specific SUT often an implementation exist that might be written in other programming language than the one used for the test environment (TE) or it is necessary to distribute the parts in the network.

The TTdatagram was designed to be able to implement the TRI interface on different target platforms. The result was the definition of a protocol where its protocol data units are capable of transporting all the information that is available at the TRI interface. In fact the TTdatagram implements a light-weighted RPC (Remote Procedure Call) type of functionality. Instead of using a standardized RPC mechanism, like CORBA, a self-defined protocol has been defined to be able to achieve one of the main tasks, interoperability among different system architectures, and to minimize the overhead, which can occur.

The TTdatagram as a protocol is independent from the transporting mechanism and the programming language used to interact with the TTdatagram protocol data. So it can be adapted for the use of TCP or UDP and can be designed in every programming language.

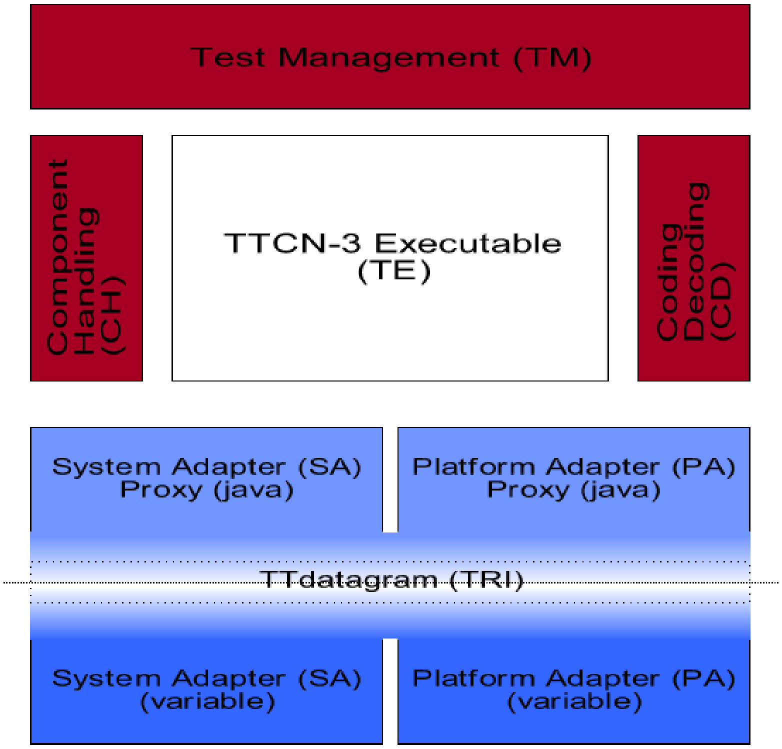

Test Management (TM)
Component Handling (CH)
TTCN-3 Executable (TE)
Coding Decoding (CD)
System Adapter (SA) Proxy (java)
Platform Adapter (PA) Proxy (java)
TTdatagram (TRI)
System Adapter (SA) (variable)
Platform Adapter (PA) (variable)

Figure 19. TT-datagram architecture

The TTdatagram protocol is a four byte aligned protocol which is used via a dedicated transport mechanism, further transport layers can be implemented in the future. The operations and parameters defined in [4] have their representation in the TTdatagram protocol. Each operation is handled as command, indicated in a common header which consist of a two bit Protocol tag, a six bit Version tag, the eight bit Command tag, a Transfer Identifier and the message length.

Currently there are implementations of the TTdatagram protocol in Java and C using UDP as transport mechanism.

4.9.1 The TTdatagram protocol message encoding

A TTdatagram represents a binary encoding of an operation call and its response (if given).

The TTdatagram has the following structure:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| P |    Version|S|  Command    |        Transfer Id             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       PayloadLength                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                       Payload                                  .
.                                                                .
```

P	- 2 Bit Protocol descriminator
Version	- 6 Bit Version number
S	- 1 Bit Status flag (0 = command, 1 = response)
Command	- 8 Bit Command identifier
Transfer Id	- 16 Bit random Transfer identifier
PayloadLength	- 32 Bit unsigned integer Payload Length
Payload	- arbitary length payload
Padding	- 0...3 “null” bytes (to get 4 bytes alignment of the message)

Supported Protocols (P):

TRI_PROT:	'00'B
TCI_PROT:	'01'B
Reserved:	'10'B

Command handling and status:

List of commands:

SA_RESET, MAP, UNMMAP, SEND, ENQUEUE_MSG, CALL, REPLY, RAISE

ENQUEUE_CALL, ENQUEUE_REPLY, ENQUEUE_EXCEPTION

SUT_ACTION_INFORMAL, SUT_ACTION_TEMPLATE

PA_RESET

START_TIIMER, STOP_TIMER, READ_TIMER, TIMER_RUNNING, TIMEOUT

EXTERNAL_FUNCTION

A command will be acknowledged by copying the P, Version, Command and the Transfer Id. The status tag S will be set to 1. The Payload Length is set to 0, if no error occured, or indicates the length of the followed Payload Data representing an error string.

Payload:
The payload of the TTdatagram consists of the parameter of the specified command. Each parameter has its representation in an integer and/or string encoding whereby the integer can be used for the indication of the following elements or length of the string. An integer is represented as a four byte value (most significant bit first). According to the conventions in C each string is closed by a null character (that is part of the payload field) and will always be padded to fulfill four byte alignment if necessary.

The following example will describe the parameter representation of the TriPortIdList used e.g. for the parameter tsiList of the command triExecuteTestCase.

TriPortIdList:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        length(integer)                         |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        port 1(TriPortId)                      |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      /                                                               /
      \                                                               \
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        port n(TriPortId)                      |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Encoding of the TriPortId:

```
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        compInst(TriComponentId)               |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        portName(Char)                         |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        portIndex(integer)                     |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        portType(QualifiedName)                |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Encoding of TriComponentId:

```
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        compInst(BinaryString)                 |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        compType(QualifiedName)                |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Encoding of BinaryString:

```
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        bits(integer)                          |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                    data(unsigned char)                        |
      .                                                               .
      .                                                               .
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Encoding of QualifiedName:

```
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        moduleName(Char)                        |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        objectName(Char)                        |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Encoding of Char:

```
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        length(integer)                        |
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
      |                        String                                 |
      .                                                               .
      .                                                               .
      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4.9.2 The Java DatagramTestAdapter

The intention of the Datagram TestAdapter is the adaptation of the TTCN-3 runtime environment to an SUT specific adaptation layer (e.g. C-based) via socket communication using the TTdatagram protocol.

The implementation of the TTdatagram protocol in the DatagramTestAdapter supports all defined TRI functions.

The DatagramTestAdapter creates a UDP socket. UDP is adequate for transport due to the fact that the DatagramTestAdapter was implemented for use on one single machine. The transport mechanism can be overwritten by a TestAdapter extending the DatagramTestAdapter to choose the adequate transport mechanism for a specific implementation.

There is a differentiation between modal and non-modal commands. Modal commands are commands where a response for the call is expected, non-modal have no response. For all modal commands a so called TRIDatagramProcessor has to be created to await the expected response. For the detailed signature of the operations represented by the commands please see [4].

5. TEST SYSTEM ARCHITECTURES USED IN THE CASE STUDIES

The general test system architecture as shown in Figure 4 has been instantiated differently in the various case studies. The main difference has been in the implementations of the SA and the PA. In some of the case studies the SA has been implemented as an executable of its own. In some other case studies, it has been implemented as part of the executable containing the TE, too. The main difference regarding the PA has been the use of simulated time in one of the case studies as opposed to real time by the other case studies. None of the case studies made use of the distribution possibilities offered by the CH.

5.1 SA AS A SEPARATE EXECUTABLE

In the automotive case study, the SMLC case study, and the bearer in a box case study, the SA has been implemented as a separate executable. In the bearer in a box case study, a major part of the SA has been dedicated hardware to access the SUT either via a 2.5G or a 3G air interface. For this case study a different TTCN-3 tool as in the other case studies has been used, therefore the architecture of this case study will not be presented here.

In the other two case studies using a separate executable for the SA the same approach has been taken to connect the SA with the TE. In both cases the SA has been written in C, thereby using the TRI with the C-language mapping. The TE offers the TRI, but with a Java language mapping. TestingTechnologies provided a small adaptation layer which forwards calls at the Java-TRI via UDP to the C-TRI in the SA executable and vice versa. The protocol to forward the calls has been named TTDatagram. This architecture is shown for the SMLC case study in Figure 20. The TCP/Ipv4 connection in this diagram is for message exchange with the SUT and is not related to this architectural split between TE and SA.

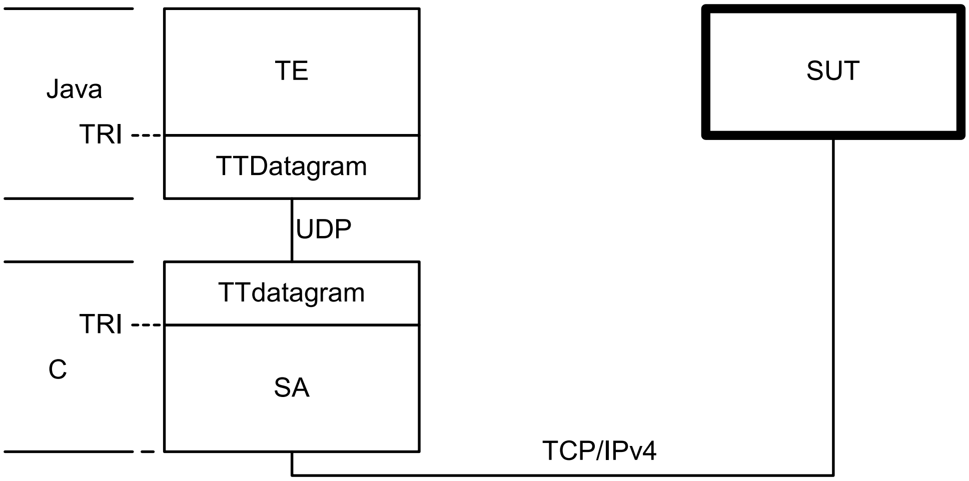


Figure 20. Separate SA executable in the SMLC case study

In this case study, the split between the TE and SA has been done because there existed an SA that provided configuration possibilities for the connection to the SUT. Without this SA being available, the SA would have been implemented in Java in the same executable as the TE.

In the automotive case study, a similar split between TE and SA has been used. The SA contained dedicated hardware to access the MOST bus. This hardware could be accessed via a C interface, provided as a DLL. Therefore again two separate executables have been used. A single executable using the JNI (Java Native Interface) would have been possible also, but would not have provided any benefit. The separation into two executables has the additional advantage that the TE can be executed on a different machine than the SA, which is tied to the PC with the physical connections to the test hardware.

5.2 SA AND TE WITHIN ONE EXECUTABLE

In the CORBA case study and the case study of the financial domain, the SA has been implemented in the same executable as the TE. Overall the test system will be simpler to execute, because there is just one executable to control.

Taking the CORBA case study as an example, the test system and the SA form a layered system, these layers are shown in Figure 21. A `call`-statement in the TTCN-3 code is processed in the various layers as follows: The signature and the actual parameters are encoded as an XML data structure and passed to the highest layer in the SA. Then, using a style sheet transformation, this XML data structure is transformed into the specific format needed by the 'XORBA' ORB. This ORB then actually issues the call to the SUT.

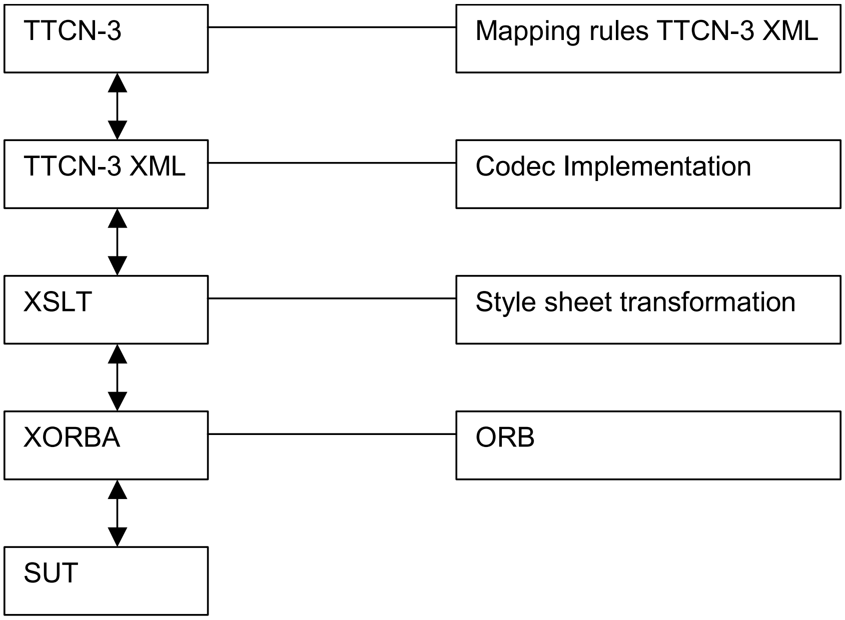


Figure 21. Architecture of the CORBA SA

Replies and exceptions to the call and calls issued by the SUT are processed in the reversed direction. Note, that all these layers including the XORBA ORB are contained in one executable.

5.3 SIMULATED TIME

All the case studies, except the railway case study use an implementation of the timers as real-time timers in the PA. This is the usual case and a default implementation is provided by TTthree.

To implement simulated time for the railway case study it has not been sufficient to implement timers in the PA that can proceed instantaneously to the time of the timer expiring next. The problem here is that the TRI and TCI as they are defined at the moment do not provide enough information to detect idleness of the test system. But timers may advance only, when the test system is idle. It still has been possible to use the general architecture of a test system, but on the level of the TTCN-3 code, for each test component in a test case, there has been an additional test component responsible for idleness detection. These additional components have been connected such that global idleness could be detected.

In the diagram in Figure 22 the additional components to detect idleness are named 'idleness handlers'. Note that these are actually executed inside the TE.

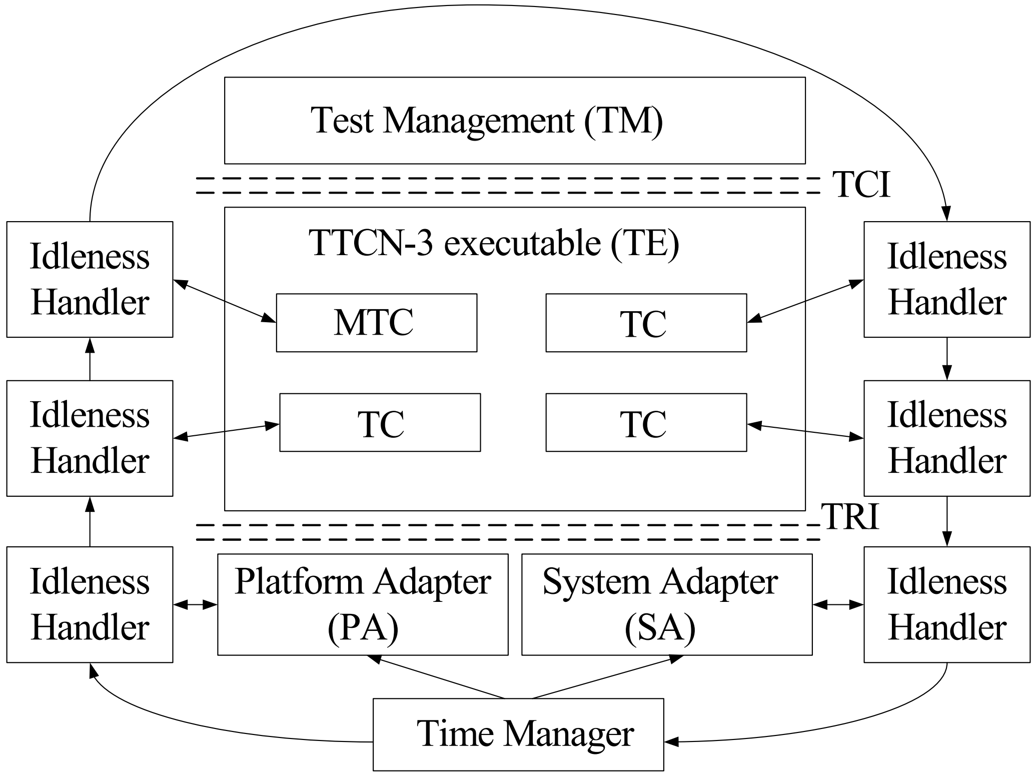


Figure 22. Architecture for Simulated Time [24]

As this architecture becomes visible on the level of the test cases written and requires a specific style of writing communication statements in the actual test behaviour, a change request to the TCI will be proposed which allows to implement simulated time such that it does not become visible to the test cases which notion of time is used when executing the test cases.

6. SUMMARY

The first phase of the TT-Medal project will develop the test infrastructure as described above. In the second phase, additional components for the test infrastructure as well as components for the test platform will be developed.

The test infrastructure will specifically been enhanced with plugins for CORBA based systems and for XML based systems. The development of the CORBA plugin address a realization of the IDL to TTCN-3 mapping (by means of implicit import combined with a visualizer for the generated TTCN-3 structures) and the realization of a TRI/TCI-based CORBA adaptor. The development of the XML plugin addess the definition of an XML to TTCN-3 mapping, a realization of this mapping (by means of implicit import combined with a visualizer for the generated TTCN-3 structures).

In addition, the test infrastructure will be extended with a logging interface. This logging interface will be defined in terms of XML structures for the logging data and in terms of interface operations for the logging events. This interface is subject to standardization activities at ETSI.

Furthermore the infrastructure address means to support SUT interfaces beside of Java. Thus work on C++ mapping and the TTdatagram approach have been investigated and reported.

I. MARKET QUESTIONAIRE

The companies under which the market survey addressed in chapter 2 was held are not only in the financial sector, but also in energy and telecom. Most of these companies are multinationals with offices around the globe.

An overview of companies who participated in the survey is listed in the table. This overview is not intended to be a complete list, but is used to indicate the diversity of the participants.

Company	Country	Business	System under test
Achmea	Netherlands	personal pension plan and insurance company	pension system
Rabobank	Netherlands	banking and insurance company	web applications
Electrabel	Belgium	supplier of energy solutions	SAP based application
Eneco	Netherlands	supplier of energy solutions	integration of Oracle databases
LogicaCMG	Netherlands	telecom vendor	WAP gateway
Vodafone	Sweden	telecom operator	billing application

The market survey made use of the following questionnaire:

Test environment

- What is your System Under Test (SUT)?
- Which interfaces and protocols are used?
- Are tests executed in a laboratory environment or in a live environment?
- Is the SUT simulated in a laboratory environment, and if so with which purpose?
- On which operating systems is the SUT being tested?
- On which operating system is a possible execution tool (e.g. test automation tool) executed?
- On which operating systems are remaining test tools executed?

Test development

- Are test specifications developed manually or generated by an application?
- Are test specifications validated?
- What is the structure of a test; for example is there a separation in test cases and scenarios?
- Which specification language is being used to describe the tests?
- Which features do you appreciate most in the specification language?
- Which features do you miss most in the specification language?
- What kind of tests are being executed: Module test, Integration test, System test, Acceptance test…
- What type of testing is being done: Black box, White box, Once only test, Regression test, Functional test, Scenario test, Load test, Performance test

Test execution

- How is test input data treated? For example, is the input data stored in external files or integrated in the test specification:?

- How is test data offered to the SUT: Web based interface, GUI (Graphical User Interface), Commandline interface, Serial interface (e.g. RS-232), IP based communication, API (Application Programming Interface), RPC (Remote Procedure Call)
- How much time is spent on protocol implementation, encoders and decoders?
- Are tests being automated at the moment? If so, completely?
- Which tools and languages are being used for test automation?
- Does the test automation tool have to work with other tools, for example requirement management tool?
- What are the requirements for performance of the test tool?
- Is the test built in one part of does it exist from several sub-tests? Can possible sub-tests be executed solely?
- How is test logging stored and analyzed?
- What do you miss in the used test tool?

Testing and company processes

- What is the status of testing within the company?
- Is there a need to improve the test process?
- Is there a need to improve the test specification?
- Is there a need to improve the test tooling?
- Is it possible to replace existing tools?
- Is the company willing to invest in test improvements?

II. TERMINOLOGY

The testing terms in this section have been taken from the basic TTCN-3 and testing methodology standards provided by ETSI and ITU-T. Software testing and quality terms defined by international standardization bodies and institutions such as ISO/BSI TMap, IEEE, and the ISTQB have been collected and added.

For better comparison the terms have been assigned to six groups according to general terms, characteristics, documents & artefacts, processes, test tools, and test types.

A. BASIC TERMS

Term	**Explanation**	**Source**
C++ component	A component can be a single class; It is more likely a collection of classes, sometimes also referred to as a module. The required and provided interfaces of C++ components are described in C++ header files.	VTT
Kind of test	One of the test categories. Differentiating functional, conformance, interoperability, robustness, etc. tests from non-functional, performance, scalability, load, stress, etc. tests	
Pass/Fail criteria	Decision rules used to determine wether a software	[18]

	item or a software feature passes or fails a test.	
Quality characteristic	Property[5] of an IT system. Examples are security, time-behaviour, usability	ISO 9126
Software Feature	A distinguishing characteristic of a software item (e.g. Portability)	[18]
System under test	A real system, sub system, or system component which is to be studied by testing. The system under test (SUT) is a part and is the system, subsystem, or component being tested. A SUT can consist of several objects. The SUT is exercised via its public interface operations and signals by the test components. No further information can be obtained from the SUT as it is a black-box.	Modification of ISO/IEC 9646 (CTMF), see *Test object* [23]
Test	A) A set of one or more test cases B) A set of one or more test procedures C) A set of one or more test cases and procedures	[18] [19]
Test categories	Tests can be classified along different categories: the granularity of the SUT (unit, module, integration, system), the approach for the test (white-box, grey-box, black-box) and the kind of tests (see kind of test)	
Test level	A test level is a group of test activities[6] directed and executed collectively. Examples of test levels are component test, integration test, system test and acceptance test.	[10] [19]
Test organisation	A test organisation comprises all of the test functions, facilities, procedures, and activities, including their relationships.	[9]

B. CHARACTERISTICS

Term	**Explanation**	**Source**
Compliance	Attributes of software that make the software adhere to application standards or conventions or regulations in law and similar prescriptions.	[10]
Correctness	The extent to which the system processes the presented input and changes correctly, according to	[9]

[5] Note: The terms characteristic and property are used with a similar meaning (like a feature).

[6] Note: Test activity has not been defined explicitly, but is used in the sence of a test case.

	the specification, into consistent data collections.	
Interoperability	Attributes of the software that bear on its ability to interact with specified systems.	[10]
Maintainability	A set of attributes that bear on the effort needed to make specified modifications.	[10]
Portability	A set of attributes that bear on the ability of software to be transferred from one environment to another.	[10]
Quality	The totality of characteristics of a product or service that bear on its ability to satisfy stated or implied needs.	ISO 8402
Recoverability	Attributes of the software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and the time and effort needed for it.	[10]
Reliability	A set of attributes that bear on the ability of software to maintain its level of performance under stated conditions for a stated period of time.	[10]
Reusability	The extent to which parts of the information system, or of its design, may be reused for the development of other applications.	[10]
Security	Attributes of software that bears on its ability to prevent unauthorised access, whether accidental or deliberate, to programs and data.	ISO 9126
Suitability	Attributes of software that bear on the presence and appropriateness of a set of functions for a specified task.	[10]
Testability	Attributes of software that bear on the effort needed for validating the modified software. Attributes of the software that bear on the effort needed for validating the modified software.	ISO 9126 [10]
Usability	A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.	[10]

C. DOCUMENTS & ARTEFACTS

Term	**Explanation**	**Source**

Abstract test case	A complete and independent specification of the actions required to achieve a specific test purpose.	Modification of ISO/IEC 9646 (CTMF)
Abstract test suite	A test suite composed of abstract test cases.	ISO/IEC 9646 (CTMF), see *Test set*
Executable test case	A realization of an abstract test case	ISO/IEC 9646 (CTMF)
Executable test suite	A test suite composed of executable test cases.	ISO/IEC 9646 (CTMF), see *Test script*
Implementation conformance statement (ICS)	A statement made by the supplier of an implementation or system claimed to conform to a given specification, stating which capabilities have been implemented. The conformance statement is the result of answering the ICS questionnaire.	ISO/IEC 9646 (CTMF)
Implementation extra information for testing (IXIT)	A statement made by a supplier or implementer of an SUT which contains or references all of the information (in addition to that given in the ICS) related to the SUT and its testing environment, which will enable the test laboratory to run an appropriate test suite against the SUT. This information is the result of answering the IXIT questionnaire.	Modification of ISO/IEC 9646 (CTMF)
Test basis	All documents from which the requirements of an information system can be inferred. The documentation on which the test is based. If a document can only be amended by way of the formal amendment procedure, the test basis is called a fixed test basis.	[10]
Test case	A logical or physical description a test that is to be executed, which has a specific test objective and which is related to a specific test unit. A test case is a specification of one case to test the system, including what to test with which input, result, and under which conditions. It is a complete technical specification of how the SUT should be tested for a given test objective. A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.	[10] part of [23] [19]
Test context	A collection of test cases together with a test configuration on the basis of which the test cases are executed.	[23]
Test control	A test control is a specification for the invocation of test cases within a test context. It is a technical	[23]

	specification of how the SUT should be tested with the given test context.	
Test item	A software item (Source code, object code, job control code, control data, or a collection of these items) which is an object of testing.	[18]
Test case specification	A document specifying inputs, predicted results, and set of execution conditions for a test item.	[18]
Test design specification	A document specifying the details of the test approach for a software feature or combination of software features and identifying the associated tests.	[18]
Test framework	A collection of test pattern/test libraries for abstract test cases and of adaptors for executable test cases. Test frameworks can exist for a specific system, a specific technology or a specific application domain.	
Test library	A collection of ready to use (potentially parameterized) abstract test definitions (such as TTCN-3 types, templates, functions, altsteps, and testcases).	
Test log	Sequence of interactions between a test system and an SUT resulting from the execution of a test case. It represents the different messages/calls exchanged between the test components and the SUT and/or the internals of involved test components. A log is associated with a verdict representing the adherence of the SUT to the test objective of the associated test case. A chronological record of relevant details about the execution of tests.	Modification [23] [18] [19]
Test objective	A prose description of a well defined objective of testing of an abstract test suite. A test objective is a named element describing what should be tested. It is associated to a test case.	Modification of ISO/IEC 9646 (CTMF) [23]
Test pattern	Prose description of recurring aspects of test systems. These aspects can be architectural (architectural test pattern), behavioural (behavioural test pattern) or data oriented (data test patterns)	ETSI PTD
Test plan	A test plan contains the overall framework and strategic choices relating to a test that is to be executed. The test plan forms the reference framework during the test execution and also serves as an instrument for communicating with the customer who ordered the test. The test plan is a	[10]

	description of the test project, including a description of the activities and planning; it is not, therefore, a description of the test cases. A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.	[18]
Test procedure specification	A document specifying a sequence of actions for the execution of a test.	[18]
Test purpose	A prose description of a well defined objective of testing of an abstract test case.	Modification of ISO/IEC 9646 (CTMF)
Test repository	Collection of test suites, test libraries or test patterns.	
Test result	A statement about the correctness/incorrectness of an SUT with respect to the test objectives/test purposes of a test suite. Containing typically the test verdicts (pass, fail, inconc or error) of executed test cases.	Modification of ISO/IEC 9646 (CTMF)
Test script	The succession of co-ordinated actions and checks relating to physical test cases, including the order in which they are to be executed. A description of how testing is to proceed.	[10], see *executable test suite*
Test set	A collection of test cases specifically focusing on one or more quality characteristics and one or more test units.	[10], see *abstract test suite*
Test specification	A description of the way in which logical test cases have been selected, as well as a description of the logical test cases. A description of what is to be tested. A document that consists of a test design specification, test case specification and/or test procedure specification.	[10] [19]
Test suite	TTCN-3 module that either explicitly or implicitly through import statements completely specifies all definitions and behaviour descriptions necessary to completely define a set of test cases.	[1]
Test summary report	A document summarizing testing activities and results. It also contains an evaluation of the corresponding test items.	[18]
Test verdict	A statement of pass, fail, or inconclusive, as specified in a test case, concerning conformance of an SUT w.r.t. that test case when it is excecuted. Verdict is the assessment of the correctness of the	Modification of ISO/IEC 9646 (CTMF)

	SUT. Predefined verdict values are pass, fail, inconclusive and error. … An Error verdict shall be used to indicate errors (exceptions) within the test system itself.	Modified [23]
Testware	Artifacts produced during the test process required to plan, design, and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing.	[19]

D. PROCESSES

Term	Explanation	Source
Compilation	Translation of abstract TTCN-3 test suites into executable tests; typically the target of compilation are traditional programming languages such as C, C++ or Java	
Coverage analysis	Measurement of achieved coverage to a specified coverage item during test execution referring to predetermined criteria to determine whether additional testing is required and if so, which test cases are needed.	[19]
Debugging	Analysis of executable TTCN-3 tests by controlling the test execution with break points, watch points, etc. and by checking the correctness of variables, parameters, snapshots, etc. The process of finding, analyzing and removing the causes of failures in software.	[19]
Deployment	The process of making physical representations of tests available on nodes and installing them so that they are ready for execution.	Modification of ITU Z.130 (eODL)
Evaluation	The evaluation and inspection of the various intermediate products and/or processes in the system development cycle.	[9]
Quality control	Operational techniques and activities that are used to fulfil requirements for quality.	ISO 8402
Simulation	Analysis of abstract TTCN-3 tests by generating a test simulation (a special kind of executable test) and by checking the correctness of the tests during that simulation.	
Test composition	Composition of test functionality to gain new/modified/adapted test functionality in new test context, for other kinds of tests or for new test	

	objectives Different means for test composition in TTCN-3 exist and enable the reuse of test data and for test behaviour. Examples are the parameterized invocation of test cases/functions/altsteps, the parameterized use of templates, or template modification.	
Test creation	Manual test development based on a functional specification and system requirements.	TestFrame
Test derivation	The automated synthesis of test skeletons from system models, system scenarios or test purpose models. The skeletons are completed manually.	
Test design	All steps needed to design/architect an abstract test suite, typically including development of test suite structure, test purposes, ICS and IXIT questionnaires, test architecture, test data and test behaviour.	
Test development	All steps needed to develop a test system, typically including test design and test realization (see also test design and test realization)	
Test generation	The automated synthesis of complete abstract tests from system models.	
Test process	The fundamental test process comprises planning, specification, execution, recording and checking for completion.	[19]
Test realization	All steps needed to develop an executable test suite, typically including implementation of abstract test cases (often supported by TTCN-3 compilers), of adaptors and portation to test equipment.	
Test variation	Modification of legacy tests, test patterns or test libraries by means of parameterization, composition, or refinement.	
Testing	Testing is a process of planning, preparing and measuring, aimed at establishing the characteristics of an information system, and indicating the difference between the actual status and the required status. The process of analyzing a software item to detect the differences existing and required conditions (that is, bugs) and to evaluate the features of the software item. The process consisting of all life cycle activities, both static and dynamic, concerned with planning,	[10] [18]

	preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.	[19]
Validation	Confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use have been fulfilled.	BS 7925-1

E. TEST TOOLS

Term	Explanation	Source
Adaptor	An implementation for the various TRI and TCI subinterfaces TRI-SA, TRI-PA, TCI-TM, TCI-CH and TCI-CD. An adaptor can be the implementation for one of these interfaces or for several of these. In many cases, the TRI-SA and the TCI-CD need to be implemented by a user only, as the implementation for TRI-PA, TCI-TM and TCI-CH are reused from the TTCN-3 tooling.	ETSI TTCN-3, part 5 and 6
Adaptor repository	Collection of TRI and TCI adaptors	
Test architecture	Component and port types used to define a test configuration in TTCN-3, which can be changed dynamically during test execution. The set of concepts to specify the structural aspects of a test context covering test components, the SUT, their configuration, etc.	ETSI TTCN-3, part 1 Modified [23]
Test Configuration	The collection of test component objects and of connections between the test component objects and to the SUT. The test configuration defines both (1) test component objects and connections when a test case is started (the initial test configuration) and (2) the maximal number of test component objects and connections during the test execution.	[23]
Test Component	A test component is a class of a test system. Test component objects realize the behavior of a test case. A test component has a set of interfaces via which it may communicate via connections with other test components or with the SUT.	[23]
Test engine (also known as TTCN-3 Executable)	The part of a test system that deals with interpretation or execution of a TTCN-3 executable test suite.	ETSI TTCN-3, part 5 and 6
Test environment	An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test.	[19]

Test equipment	General purpose or test specific device able to run executable tests; being the technical basis of a test system	
Test execution tool	A type of test tool that is able to execute other software using an automated test script, e.g. capture/playback.	[19]
Test harness	A test environment comprised of stubs and drivers needed to conduct a test.	[19]
Test infrastructure	The kernel of a test platform providing means to execute tests. Containing test engine, test equipment and executable tests and additional components such as for logging and tracing. The environment in which the test is performed, consisting of hardware, system software, test tools, procedures, and so on. The organizational artifacts needed to perform testing, consisting of test environments, test tools, office environment and procedures.	TT-Medal FPP [9] [19]
Test manager	An entity which provides a user interface to as well as the administration of the TTCN-3 test system	ETSI TTCN-3, part 6
Test object	The IT system (or part of it) which is to be tested.	[9], see *System under test*
Test platform	A set of components supporting the design, development, analysis and execution of tests. The kernel of a test platform is the test infrastructure used to execute tests.	
Test system	The real system which includes the test engine, executable tests, adaptors, and the test equipment.	Modification of ISO/IEC 9646 (CTMF)
Test unit	A collection of processes, transactions and functions which are tested collectively. The totality of test units defines the test object.	[10]

F. TEST TYPES

Term	**Explanation**	**Source**
Acceptance test	A test conducted by future users and managers to determine whether a system satisfies its acceptance criteria and to enable the user to determine whether the system is ready to be accepted. The test is intended to show that the developed system meets the functional and quality requirements. Formal testing with respect to user needs, requirements, and business processes conducted to	[10]

	determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.	[19]
Black-box test	A test on the externally visible characteristics of an information system, without any knowledge of the system's internal design.	[10]
Dynamic testing	Testing, on the basis of specific test cases, by executing the test object, or by running programs.	[10]
Integration test	A test, executed by the developer in a laboratory environment, that should demonstrate that a logical series of programs meets the requirements as set in the design specifications.	[9]
Legacy tests	Existing tests being available inhouse or from external sources	
Load tests	A test type concerned with measuring the behavior of a component or system with increasing load, e.g. number of parallel users and/or numbers of transactions to determine what load can be handled by the component or system.	[19]
Performance testing	The process of testing to determine the performance (degree to which a system or component accomplishes its designated functions within given constraints regarding processing time and throughput rate) of a software product.	[19]
Regression test	Regression is the phenomenon that the quality of a system decreases as a result of individual adjustments. The aim of a regression test is to check that all parts of a system still function correctly after a change has been made.	[9]
Scalability testing	Testing to determine the scalability (capability of the software product to be upgraded to accommodate increased loads) of the software product.	[19]
Static testing	Testing by checking and examining products without any programs being executed.	[10]
System test	The system test is a test executed by the developer in a (controllable) laboratory environment, which should demonstrate that the developers system or parts of it meet the requirements set in the functional and technical specifications.	[10]
White-box test	A test on the internal properties of a test object with an understanding of the object's internal design.	[10]

Unit test	The test which is executed by the developer in a laboratory environment and must show that a program meets the requirements set in the technical specifications.	[10]

III. GLOSSARY

CAN	Controller Area Network (a bus technology in automotive)
CASE	Computer Aided Software Engineering
CAST	Computer Aided Software Testing
CD	see TCI-CD
CM	see TCI-CM
CORBA	Common Object Request Broker Architecture
CUI	Command-line user interface
CUT	Component under test
ETSI	European Telecommunication Standards Institute (www.etsi.org)
GFT	Graphical format of TTCN-3
GPRS	General Packet Radio Services (a mobile communication technology, generation 2.5)
GSM	Global System for Mobile Communication (a mobile communication technology, generation 2)
GUI	Graphical user interface
ICS	Implementation conformance statement
IDL	Interface Definition Language
IOR	interoperable object reference
ITEA	Information Technology for European Advancement (www.itea-office.org)
IXIT	Implementation extra information for testing
LT	Lower tester
MOST	Media Oriented System Transport (a bus technology in automotive)
OEC	Operating Environment Capabilities
ORB	Object Request Broker

PA	see TRI-PA
SA	see TRI-SA
SVG	Scalable vector graphics
SUT	System under test
TCI	TTCN-3 Control Interfaces
TCI-CD	Coding/Decoding Adaptor (realizing the TCI-CD Interface)
TCI-CH	Component Handling Adaptor (realizing the TCI-CH Interface)
TCI-TM	Test Management Adaptor (realizing the TCI-TM Interface)
TE	Test Engine (also called TTCN-3 Execution Engine or TTCN-3 Executable)
TM	see TCI-TM
TRI	TTCN-3 Runtime Interfaces
TRI-PA	Platform Adaptor (realizing the TRI-PA Interface)
TRI-SA	System Adaptor (realizing the TRI-SA Interface)
TTCN-3	Testing and Test Control Notation
TT-Medal	ITEA project on Tests and Testing Methodologies with Advanced Languages
UMTS	Universal Mobile Telecommunications System (a mobile communication technology, generation 3)
UT	Upper tester
XORBA	XML to CORBA Link/Interface