



Title:
Test Management and Distributed Test Process Control

Version: 1.0
Date : 20/11/08
Pages : 33

Author:

To:
D-MINT Consortium

The D-MINT Consortium consists of:
Nokia Siemens Networks, ABB, Åbo Akademi, Conformiq, Daimler, DATAPIXEL, ELIKO, Elvior, Fraunhofer IESE, Fraunhofer FOKUS, IDEKO, Innovalia Association, INSPIRE, iXtronics GmbH, Nethawk Oyj, PikeTec GmbH, SQS, SORALUCE, Tallinn University of Technology, Testing Technologies IST GmbH, TRIMEK, VTT Technical Research Centre of Finland, ETSI, TANDBERG, Simula Research

Printed on:
30-10-2008 12:02:00

Status:

- Draft
- To be reviewed
- Proposal
- Final / Released

Confidentiality:

- Public - Intended for public use
- Restricted - Intended for D-MINT consortium only
- Confidential - Intended for individual partner only

Deliverable ID: **D.3.2.v1.0**

Title:
Test Management and Distributed Test Process Control

Summary / Contents:

This document addresses the management testing procedures and motivates the industrial needs of distributed environments. Particular attention will be devoted to the current weaknesses in test management and test process control. One important challenge to be addressed by this document will be the introduction of a quality assurance strategy embedding the test management architectures presented above for both standalone systems and embedded devices.



	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 2 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

TABLE OF CONTENTS

1.	Introduction to Testing Methods	6
1.1	Conformance Testing	6
1.2	Interoperability Testing	6
1.3	Load Testing.....	7
1.4	Local and Distributed Testing	7
1.5	A Short Overview of TTCN-3	9
2.	Test Management	11
2.1	Introduction.....	11
2.2	Automatic Test Management - An Example.....	12
2.2.1	Architecture.....	12
2.2.2	Dynamics	13
3.	Quality Assurance Systems.....	15
3.1	Introduction.....	15
3.2	Architecture	15
3.3	An Example Application.....	16
3.4	Future Steps.....	18
4.	Distributed Test Process Control	19
4.1	Introduction, Motivation.....	19
4.1.1	Distributed Test System for Load Tests.....	20
4.1.2	Heterogeneous Interfaces	20
4.1.3	Distributed System Under Test.....	20
4.2	General Structure of a Distributed TTCN-3 Test System	21
4.3	Distributed Test System Requirements.....	23
4.4	Architecture	23
4.4.1	Containers.....	24
4.4.2	Test Console.....	24
4.4.3	Daemons	24
4.4.4	Session Manager	25
4.4.5	Test System Entities	25
4.4.6	Why CORBA as Middleware	25
4.4.7	Container Configuration File.....	26
4.4.8	Test Component Distribution Language	26
5.	Requirements for The Environment of Testing Distributed Embedded Software.....	27
5.1	General Outline	27
5.1.1	Purpose	27
5.1.2	Rationales.....	27
5.2	Requirements.....	28
6.	Conclusion	33
7.	References	33


	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 3 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

CHANGE LOG

Vers.	Date	Author	Description
0.1	25/07/08	Bogdan (TestingTech)	TOC draft
0.2	20/10/08	Alex (TestingTech)	Merging / Adapting Elvior Contents
0.3	28/10/08	Alex (TestingTech)	Merging / Adapting PikeTec Contents
0.4	31/10/08	Alex (TestingTech)	Submitted for Review
0.5	19/11/08	Stephan (TestingTech)	Incorporated review comments from Jon, Dragos and Detlef
1.0	20/11/08	Alex (TestingTech)	Adaption to Final Version


APPLICABLE DOCUMENT LIST

Ref.	Title, author, source, date, status	Identification

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 4 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

Abbreviations

ATS	Automated Test Suite / Abstract Test Suite
AUI	Automatic User Interaction
CD	Codec
CH	Component Handling
CORBA	Common Object Request Broker Architecture
CP	Coordination Point
DoS	Denial of Service
ETS	Executable Test Suite
ETSI	European Telecommunications Standards Institute
IUT	Implementation Under Test
MTC	Main Test Component
ORB	Object Request Broker
PA	Platform Adapter
PCO	Point of Control and Observation
PTC	Parallel Test Component
QAS	Quality Assurance System
QC	Quality Center
SA	System Adapter
SC	System Component
SCT	System Component Test
SCT ENV	Simulation and Testing Environment
SUT	System Under Test
TA	Test Adapter
TCI	TTCN-3 Control Interface
TE	Test Executable
TM	Test Management
TPT	Time Partition Testing
TRI	TTCN-3 Runtime Interface
TTCN-3	Testing and Test Control Notation version 3

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 5 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

EXECUTIVE SUMMARY

This document addresses common as well as specific problems in test management and test process control regarding industrial needs for testing of standalone systems and embedded devices.


Chapter 1 presents an overview of the different testing methodologies, motivating their usage in different scenarios.

Chapter 2 addresses the common problem of how to handle necessary interaction between the tester and the SUT in automated test execution. A solution is proposed for thus interaction, implemented in TTworkbench tool by Testing Technologies.

Chapter 3 motivates the need for Quality Assurance Systems (QAS) and presents the integration of the TPT test management by PikeTec into the HP Quality Center.

Chapter 4 addresses typical management problems in a distributed test environment and introduces TTmex, a highly standardized architecture based on TTCN-3, maintained by Testing Technologies.

Chapter 5 analyzes problems that may occur while testing distributed embedded software and proposes a set of requirements for the respective test environments.

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 6 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

1. INTRODUCTION TO TESTING METHODS

Testing is one of the fundamental issues a vendor has to take care before, while, and after releasing a product. Testing should not be relegated to a specific stage of the development phase, but has to be performed during the whole development cycle.

In the early phase, mostly functional testing is performed in order to verify that the product works according to the specification. After the product reaches a stable state, load tests (or stress tests) are performed.

Testing of systems is done to increase quality, safety and reliability of the systems and thus to ensure that implementations from different vendors will be able to work together and exchange information in integrated environments. The implementations have to fulfill various conditions such as:

- The implementations should meet the specifications.
- The implementations should be able to interoperate with each other.
- The implementation should perform their task correctly under various load conditions.

Therefore, different types of testing have been developed. The most common examples are:

- Conformance Testing
- Interoperability Testing
- Load Testing

1.1 CONFORMANCE TESTING

Conformance testing is a validation that an implementation meets the formal requirements of the referenced standards and, more precisely, that it meets the conformance clauses contained in the standards.


The primary objective of conformance testing is to increase the probability that different product implementations actually interoperate. An implementation of a system is declared conformant if its capabilities and behavior are shown to satisfy the requirements of a defined set of capabilities or options in the referenced standards.

1.2 INTEROPERABILITY TESTING

Interoperability is the ability of different systems or implementations, typically from different vendors, to work together. Interoperability testing is checking whether an implementation can interwork with another implementation.

The European Telecommunications Standards Institute (ETSI) defines four interoperability classifications:

- Protocol Interoperability: The ability of a distributed system to interchange Protocol Data Units via the communications platform.
- Service Interoperability: The ability of a distributed system to support a subset of the distributed service. The distributed service is offered through the service interfaces.

	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 7 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

- Application Interoperability: The ability of a distributed system to provide a consistent implementation of the syntax and semantics of the data which is interchanged.
- User Perceived Interoperability: The ability of the service user (human, application, machine) to exchange information via the distributed system.

1.3 LOAD TESTING

Load Testing evaluates the behavior of a system under load conditions with regard to adherence to the requirements and enables developers to isolate bottlenecks in the system. In general, load testing involves two kinds of test equipment:

- A generator to generate different patterns of traffic and traffic loads. This traffic has the task of emulating many hundreds or thousands of users which interact with the System Under Test (SUT).
- An analyzer to measure the performance parameters (e.g. the response time of the SUT).

There are different kinds of load tests:

- Performance Testing evaluates an implementation under different traffic and load conditions. It consists of measuring performance-oriented quality-of-service parameters such as delays, throughput and error rates under well known conditions.
- Stress Testing determines the maximum number of concurrent application users or clients the SUT can manage, i.e. the SUT is brought up to the overload limit
- Robustness Testing is load testing over extended periods to validate an application's stability and reliability.

1.4 LOCAL AND DISTRIBUTED TESTING

There are two different methods of implementing a test: local and distributed. These methods differ from each other based on where the test components reside, i.e. if all test components reside in the same testing device or not.

In Figure 1, a generic local testing configuration is shown. A test system contains a main test component (MTC), which coordinates the test execution. The test execution is performed in the parallel test components (PTC). They can exchange coordination messages (CM) among each other and with the main test component via the coordination points (CP). The test components are interconnected with the Implementation Under Test (IUT) at the Point of Control and Observation (PCO). If only one test component is required, then there is no need for parallel test components, i.e. the configuration contains only one test component - the MTC.

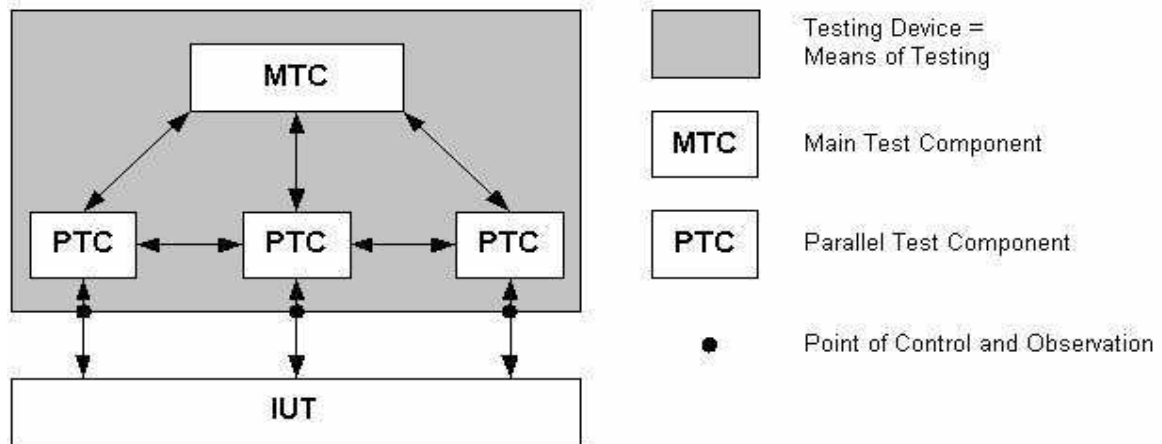


Figure 1: Generic Local Testing Configuration

The term 'local' reflects the fact that all the test components reside in the same testing device, i.e. they are local to the testing device. The main point in local testing is that all test components (MTC and PTCs) share the same resources (like time and memory).

Because of this they all have access to the same clock (testing device's clock) and as such, there is no need of time synchronization between the test components and the main test component.

A generic configuration for distributed testing is shown in Figure 2. Here, every test entity (MTC and PTCs) may reside on a separate testing device. Except for sharing the communication link, they are not sharing any other resources. This applies also to time. So every testing device and thus every test entity has its own notion of time. Because of this, time synchronization may be needed, but it depends on what has to be tested, e.g. conformance testing has less important requirements on time synchronization, whereas the latter plays an important role in performance testing. In addition to time synchronization, functional synchronization has to be done. Functional synchronization represents the procedures needed to perform test setup, maintenance, and clearing. But also test execution and test reporting belong to this category. Thus, in a distributed testing system a lot of synchronization work has to be performed and thus, the approach is much more elaborated than in local testing. Chapter 4 and 5 state the requirements which have to be fulfilled by a distributed test system.

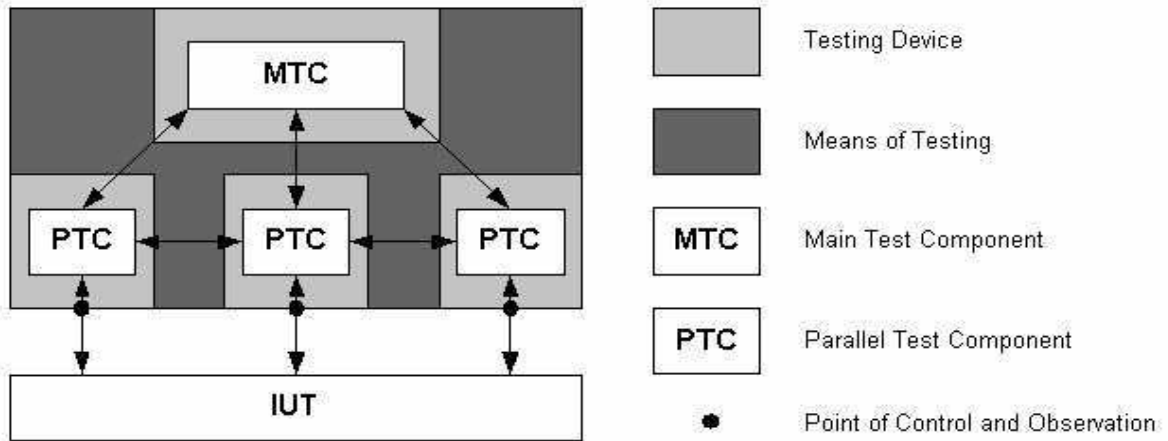


Figure 2: Generic Distributed Testing Configuration

1.5 A SHORT OVERVIEW OF TTCN-3

The Testing and Test Control Notation version 3 (TTCN-3) is a rather young language, designed for test specification and test implementation. In the following, a general introduction to TTCN-3 will be given, along with a description of the structure of a TTCN-3 test system.

TTCN-3 covers a wide range of testing possibilities for distributed and communication-based systems. The most common test types are:


- Functional and Conformance Testing
- Load Testing
- Scalability Testing
- Interoperability Testing

The development of the language was first initiated in 1998. The standardization process started in 2000, and in 2002, the standardization body "European Telecommunications Standards Institute" (ETSI) introduced the first complete version. This version has since then been continuously enhanced on the basis of improvement suggestions made by the industry. Up to date, the TTCN-3 standard has reached version 3.4.1.

TTCN-3 is a complete revision of the older TTCN standard (then known as "Tree and Tabular Combined Notation"), widely used for testing in the areas of GSM and ISDN telecommunication. It consists of a textual core language, with possibilities for interfaces to different data description languages. A number of these interfaces are meanwhile part of the standard, namely ASN.1, IDL and XML.

Besides the possibility to describe formalized test specifications with different presentation formats, TTCN-3 also offers a reference implementation architecture, which enables test solution providers, like test device manufacturers, to implement this standard and provide it to testers.

The incorporated presentation formats are, in addition to the already mentioned textual core language, a tabular and a graphical presentation format. The graphical format represents test sequences as specialized sequence diagrams. This format is used for test script documentation. The tabular format was mainly added to the standard for compatibility reasons to the above mentioned precedent TTCN version and it is of no practical relevance. Despite the existence of other presentation formats, practice has shown that test developers as well as standardization bodies mainly use the textual core language for test specification.

	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 10 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

There is also the possibility to define own presentation formats, making TTCN-3 adaptable to different application areas.

Although having its roots in the area of telecommunication, its versatility makes TTCN-3 applicable wherever reactive systems with different local or distributed communication interfaces are to be tested. TTCN-3 supports asynchronous, message-based communication as well as synchronous, procedure-based communication. The former is typically applied to protocol testing (e.g. Internet Protocols), whereas the latter is used for service-oriented testing (e.g. Web Services and Middleware Platforms). Since TTCN-3 has the capability of combining these two communication principles, it is also possible to use it for testing heterogeneous interfaces, which are rather common in IT and telecommunication, controlled by a single centralized test logic.

Figure 3 shows the above mentioned reference architecture, which is used for concrete implementation, together with its two standardized interfaces, the TTCN-3 Runtime Interface (TRI) and the TTCN-3 Control Interfaces (TCI). The purpose of these interfaces is to enable easy adaptation of a TTCN-3 test suite to a concrete test infrastructure. TRI is used to align the communication interfaces of the test system and the SUT (System under Test), whereas the TCI handles the test system internal communication.

The System Adaptor (SA) provides the actual implementation of TTCN-3 functionality on the respective platform. The Platform Adaptor (PA) provides platform specific operations for the implementation of timers and external functionality which is outside the scope of TTCN-3. Both SA and PA together are often referred to as Test Adaptor (TA).

The TCI is responsible for the execution of a TTCN-3 test suite, called Test Management (TM). It also realizes the distribution of test components, referred to as Component Handling (CH), and the Test Logging (TL). Additionally, the TCI accounts for encoding and decoding of test data, which is done in a specific CoDec (CD).

In the last two years, various standardization bodies have chosen TTCN-3 as their test specification language (e.g. 3GPP, WiMAX Forum and AUTOSAR), which can be seen a sign of its growing acceptance among test developers.

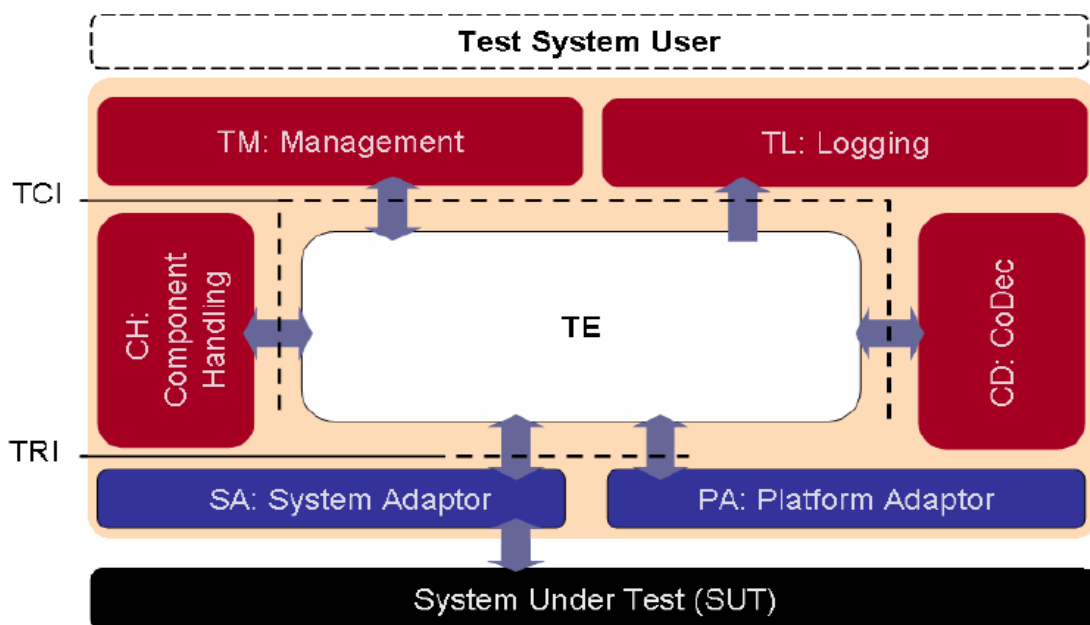


Figure 3: TTCN-3 Reference Implementation Architecture

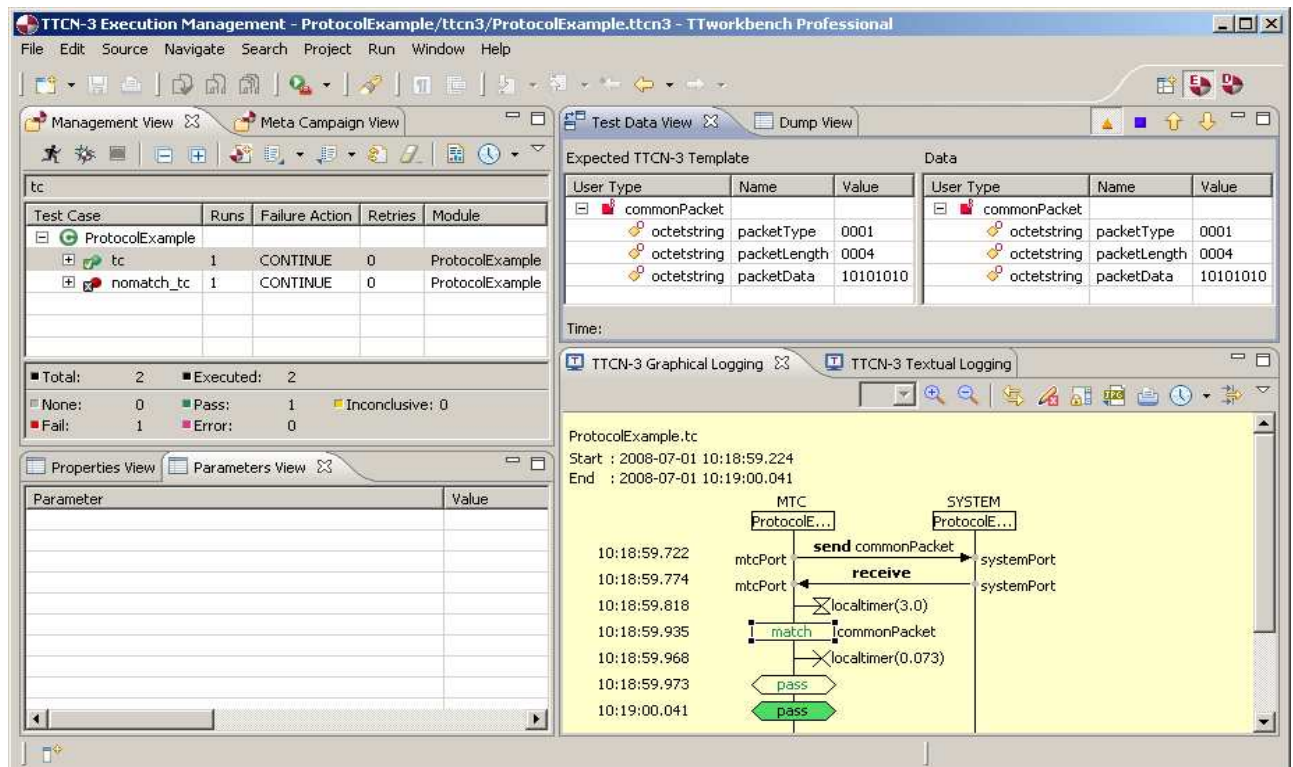
2. TEST MANAGEMENT

2.1 INTRODUCTION

A test management component is the core part of a test system when it comes to the actual application of a test suite. Its main tasks are:

- Configuration, Execution, Saving and Resuming of Test Campaigns
- Logging of Test Execution
- Analysis of Test Runs
- Generation of Test Reports

Figure 4 shows an example of the test management component in TTworkbench, a TTCN-3 based test development and execution IDE.



The screenshot displays the TTCN-3 Execution Management interface. The 'Management View' shows a table of test cases:

Test Case	Runs	Failure Action	Retries	Module
ProtocolExample				
tc	1	CONTINUE	0	ProtocolExample
nomatch_tc	1	CONTINUE	0	ProtocolExample

The 'Test Data View' shows the 'Expected TTCN-3 Template' and 'Data' tables:


User Type	Name	Value	User Type	Name	Value
commonPacket	packetType	0001	commonPacket	packetType	0001
octetstring	packetLength	0004	octetstring	packetLength	0004
octetstring	packetData	10101010	octetstring	packetData	10101010

The 'TTCN-3 Graphical Logging' window shows a sequence diagram for 'ProtocolExample.tc' with the following timeline:

```

sequenceDiagram
    participant MTC as ProtocolE...
    participant SYSTEM as ProtocolE...
    Note over MTC: 10:18:59.722
    MTC->>SYSTEM: send commonPacket
    Note over SYSTEM: 10:18:59.774
    SYSTEM-->>MTC: receive
    Note over MTC: 10:18:59.818
    MTC->>MTC: localtimer(3.0)
    Note over MTC: 10:18:59.935
    MTC->>MTC: match commonPacket
    Note over MTC: 10:18:59.968
    MTC->>MTC: localtimer(0.073)
    Note over MTC: 10:18:59.973
    MTC->>MTC: pass
    Note over MTC: 10:19:00.041
    MTC->>MTC: pass
  
```

Figure 4: TTCN-3 Test Management Component in TTworkbench

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 12 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

The figure shows the management perspective with a test case view, a statistics view, a test case parameter view, a test data view, and a graphical logging view currently active.

Besides standard manual test management, there are numerous scenarios where automatic test management is an interesting option, which will be described in the next section.

2.2 AUTOMATIC TEST MANAGEMENT - AN EXAMPLE

A lot of test management components already offer the possibility to automatically execute test cases. To take test management automation to the next level, features for automatic configuration of test cases and the automation of user interactions were developed at Testing Technologies. The current implementation is applied to a TTCN-3 test system, but is not restricted to this environment.

Automatic User Interaction (AUI) is especially useful for test suites which incorporate test cases where the tester has to manually change the state of the SUT (e.g. multi-band frequency tests). This is called *user interaction*. Normally this is done via a user dialog, asking the user to perform the requested action (e.g. "Please change the frequency to 10.000 Hz.") and click a button, after which the test suite execution continues.

In the approach taken to develop a test system which can automate both configuration and execution of simple test cases as well as test cases requiring user interaction, a client-server architecture is used, as explained in the following section.

2.2.1 Architecture

Figure 5 shows the architecture used to automate test case configuration, execution and user interaction.

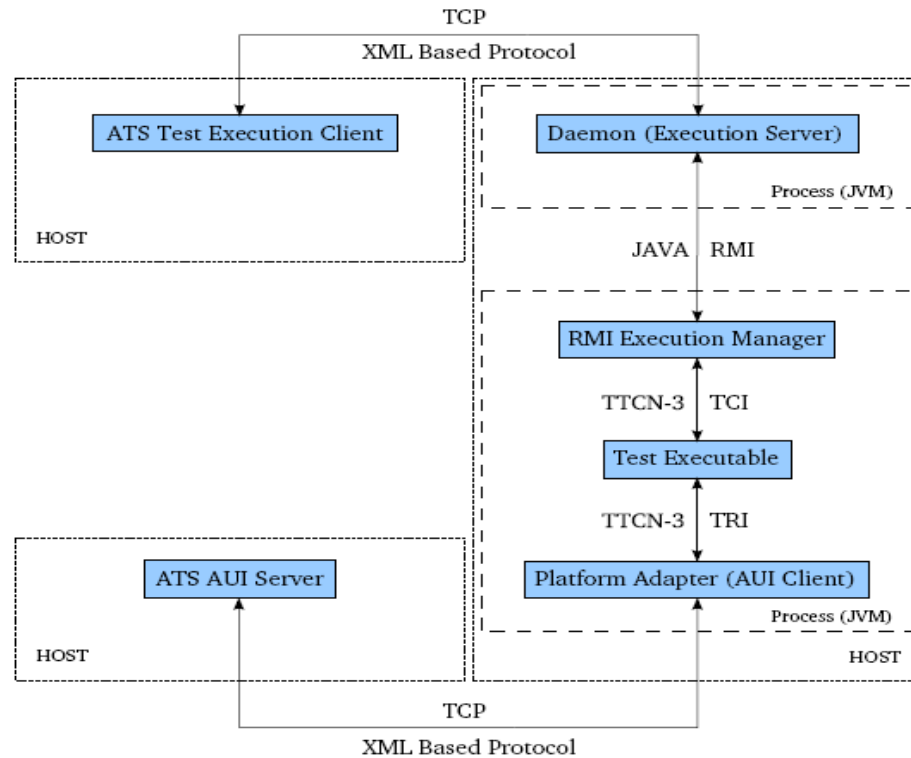


Figure 5: Automatic Test Management Architecture

The tester is simulated by two instances: The ATS (Automated Test Suite) Client and the ATS AUI server. The client processes script files and controls the execution server, which in turn controls the execution. Whenever a test case with user interaction is run, the test execution process queries the AUI server, which runs the respective script. This script triggers the requested change in SUT via an appropriate interface. This scenario only works with SUTs which can be controlled via a control interface.

A tester has to prepare two kinds of scripts in order to utilize this kind of management:

- The test execution scripts specify the test suite configuration, test case parameterization, run order etc. and are evaluated by the ATS Client.
- The AUI scripts utilize SUT interfaces to trigger changes in the SUT and are called by the AUI server.

2.2.2 Dynamics

An example of a test case execution with automated user interaction and an asynchronously sent query can be seen in Figure 6.

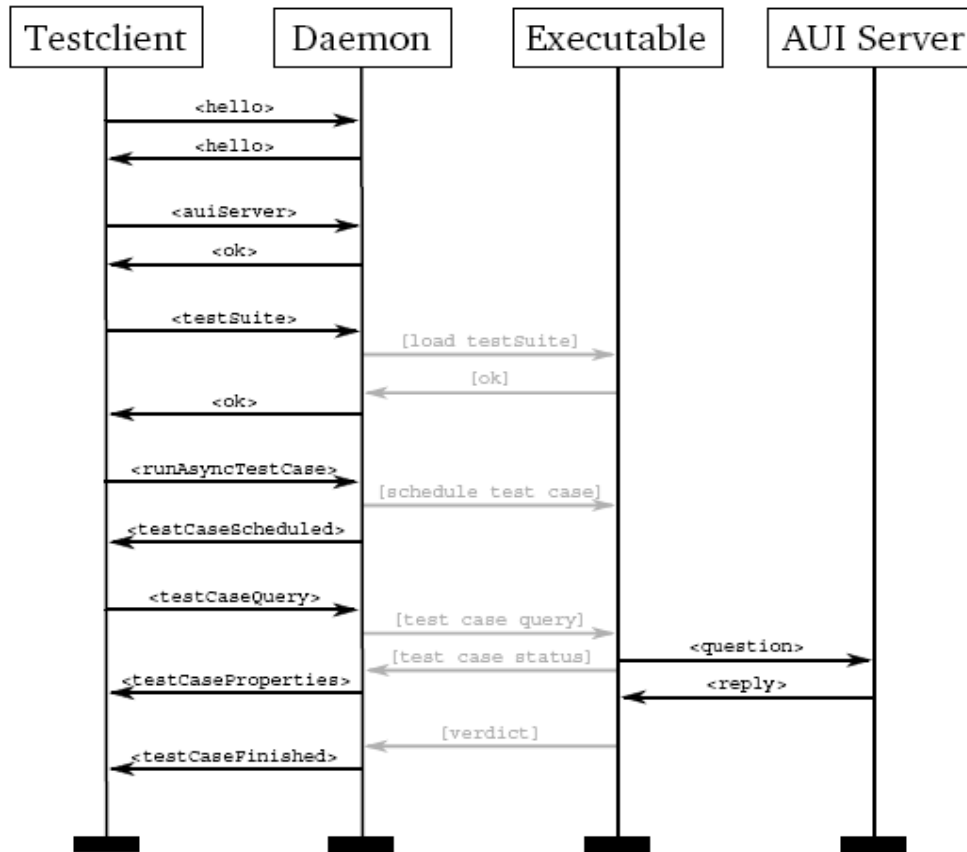



Figure 6: Example of a Test Case Execution

After a handshake with the daemon, the test client registers the AUI server and tells the daemon to load a test suite. After that, test case execution is scheduled. The client then queries the status of the test case and receives a reply with information whether the test case is running or scheduled etc. The test case incorporates a user interaction as question, which is passed to the AUI server and a reply is given. Upon test case termination, the verdict is received by the daemon and relaxed to the client for logging and evaluation.

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 15 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

3. QUALITY ASSURANCE SYSTEMS

3.1 INTRODUCTION

Test models are abstract descriptions of test problems and, in general, are related to a system specification and/or system models. From this perspective, test models are the central artifacts in model based testing and describe the quality assurance from a functional perspective. In addition, another way of looking at the quality assurance is the test process' perspective, as shown in Figure 7.

3.2 ARCHITECTURE

The test management is the central task in the test process that covers definition of test concepts, test purpose, test procedures, quality criteria, test strategy, and test organization as well as the resource management, the creation of a concrete test plan, and the controlling and monitoring of all testing activities. In addition, the test management has an interface to the test documentation because the test management should define how the different test artifacts have to be documented, stored, and archived for durable access.

The test management has a broad view on the testing activities and constitutes the interface to the overall project management. Test management has a look at all testing activities for a system development project as a whole, covering all test levels (module, integration, system tests) and testing artifacts.

However, as mentioned before, test management has the focus on the testing process and not on the test products, thus, not on the test models representing the relevant test aspects with respect to system requirements and/or system models. Therefore, it is reasonable to create a link between high level test artifacts from a test management perspective and low level test artifacts from a test modeling perspective. This link should guarantee a consistent exchange of test information over a project's life cycle.

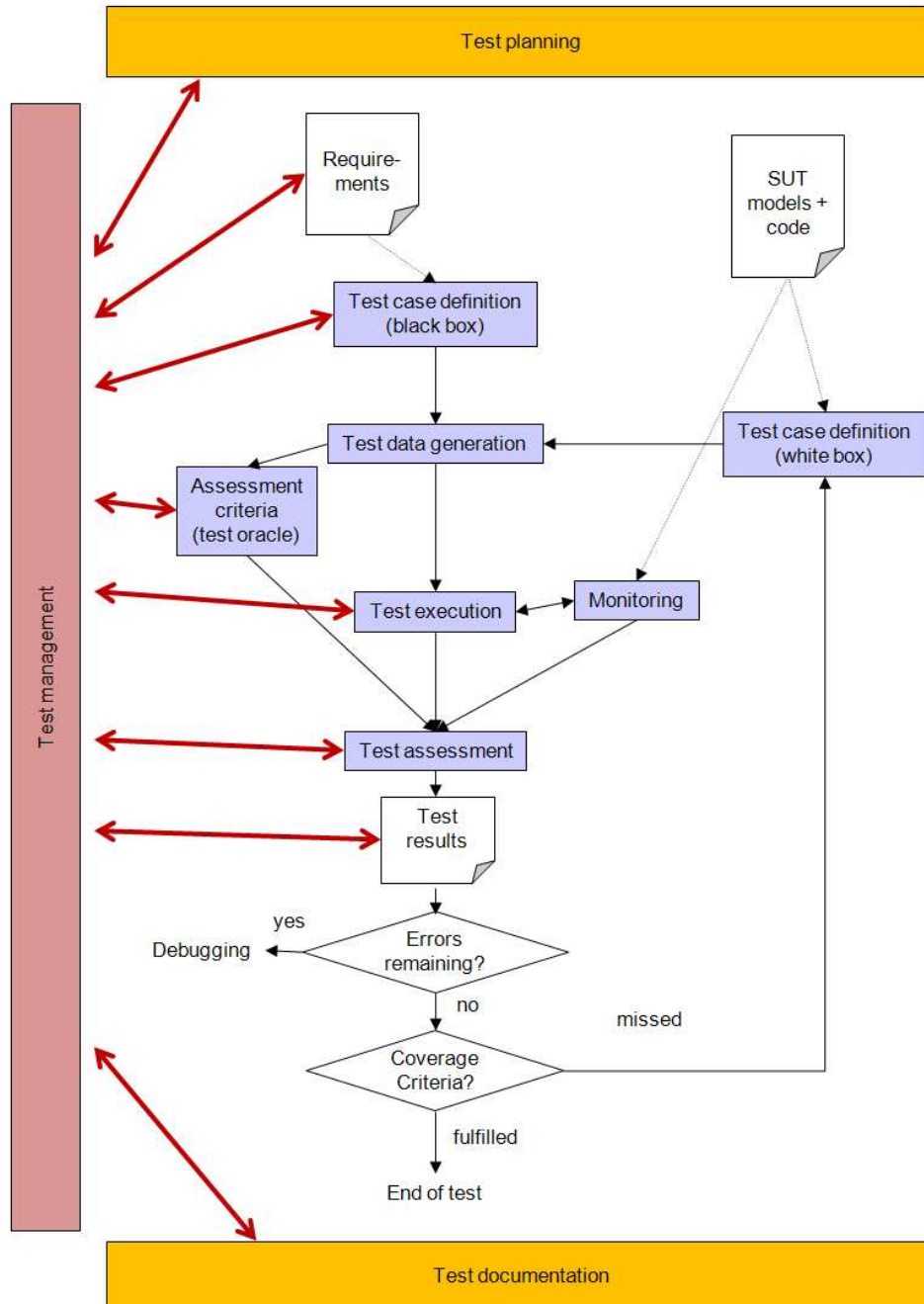


Figure 7: A Test Process incl. Relationship of Core to Test Planning, Management and Documentation

3.3 AN EXAMPLE APPLICATION

A practical example is shown in Figure 8. It demonstrates how test scenarios in TPT (as instances of TPT test models) are linked to test cases in Quality Center (which is a test management solution from Hewlett Packard). Test scenarios in TPT are linked with test cases in Quality Center. Since both tools provide unique IDs for test scenarios/test cases, the identification can be managed consistently over the lifetime of a project

by means of ID-pairing. Both views can be synchronized on user request without semantic conflicts. The ordering criteria for test cases in Quality Center (QC) and test scenarios in TPT may differ (due to different aspects of ordering test cases on both levels). Therefore, ordering is not synchronized automatically.

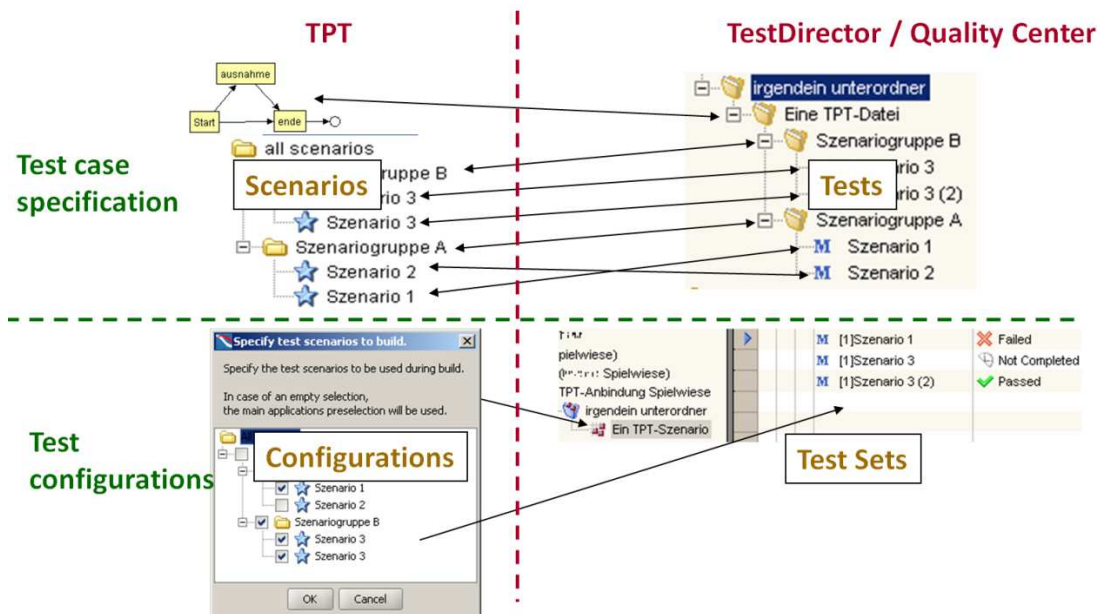



Figure 8: Mapping between test specifications and test configurations

New test cases/test scenarios can be created on both sides: on the abstract level (QC) from a management perspective and on the modeling level (TPT) from a test implementation perspective. Therefore, collaboration between test management and test modeling is possible without restrictions.

A similar linkage is available for test configurations: In both tools, test sets can be configured to specify configurations as a basis for test and resource planning. Test sets can be created, modified or removed in QC and TPT in parallel.

Apart from this, QC operates as the central repository for test results. After test execution with TPT the results are transferred to the QC repository automatically together with information such as:

- QC test case ID
- Test case version
- QC test configuration ID
- Test group
- Test platform
- Verdict
- Tester ID
- Date and Time of test execution
- Estimated runtime
- Workflow status
- Data and report (attachments)

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 18 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

This ensures that QC can serve as a central data repository server for all testing activities that supports project controlling and status reporting even for large scale projects.

3.4 FUTURE STEPS

The current state of integration of TPT (as a test modeling tool suite) and Quality Center (as a test management solution) does not cover aspects such as (functional) requirements tracing or defect tracking. However, both areas have strong relations to the test modeling, so future developments will elaborate on such issues. In general, Quality center addresses the following features:

- **Defining Test Requirements**

Building a requirements tree; creating and defining requirements; tracking the status of requirements.

- **Test Planning**

Building a test plan tree; creating tests; Linking tests and requirements; designing test steps; using parameters in tests; configuring a test to call other tests; generating test scripts (if applicable); monitoring the status of test plans.

- **Test Execution**

Building a test sets tree; creating test sets; organizing tests in a test set; defining and scheduling test execution flows; configure automated test rerun and cleanup rules; executing manual and automated tests; recording and reviewing test execution results; tracking and reviewing test runs.

- **Defect Tracking**

Logging defects; searching and reviewing defects; associating defects to tests; tracking the status of defects.

- **Reporting and Analysis**

Filtering and organizing data for reports and graphs; generating reports and graphs; generating formatted project documentations.

- **Importing and Exporting Data**

Formatting requirements, test plan, and defects data in Microsoft Word and Excel files; importing data from Microsoft Word, Excel, and many others to Quality Center.

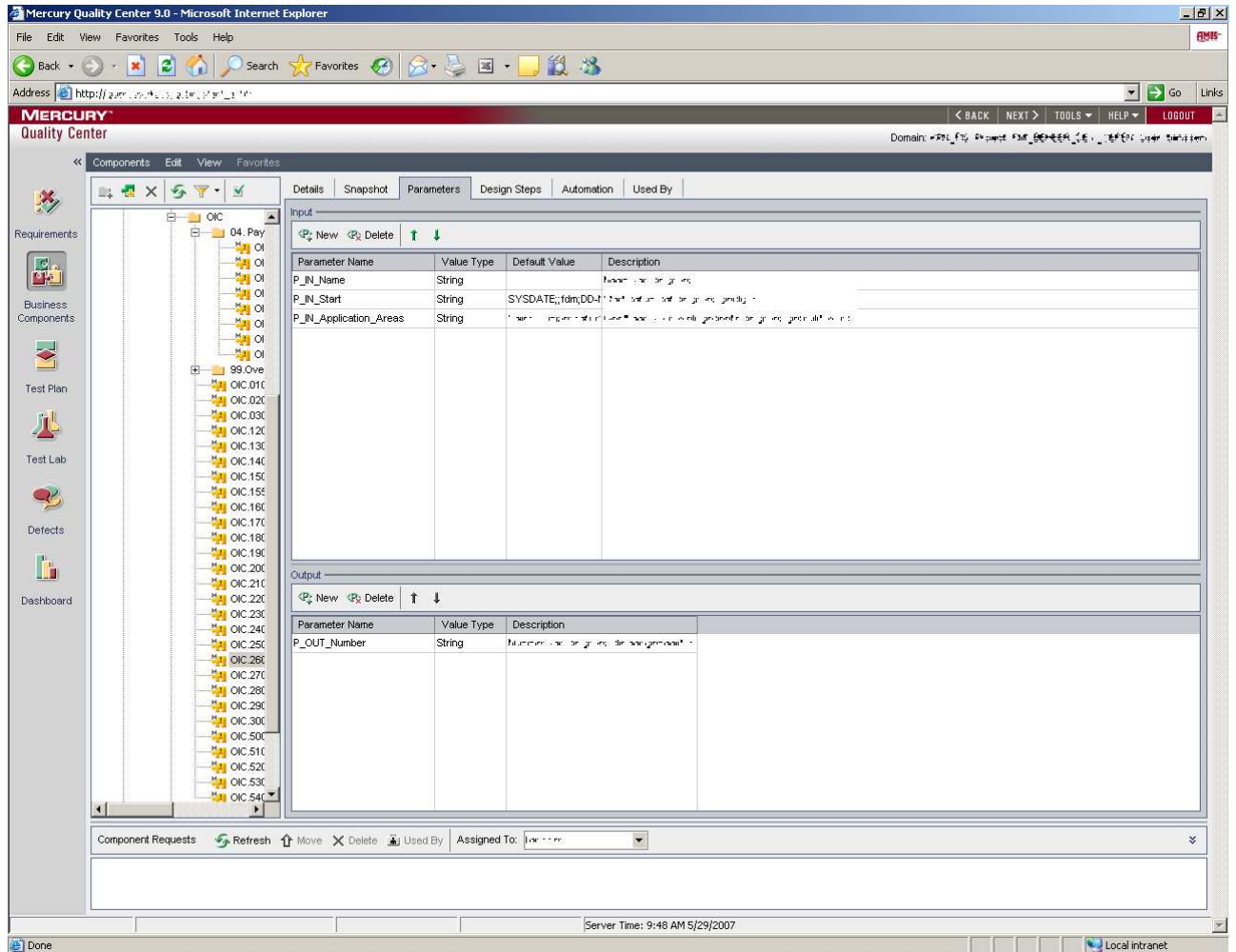



Figure 9: Quality Center Screenshot

4. DISTRIBUTED TEST PROCESS CONTROL

4.1 INTRODUCTION, MOTIVATION

As already mentioned in section 1.4, distributed testing means that the test entities (MTC and PTCs) reside on separate test systems. But why is this needed? There are several reasons why a distributed test setup can be chosen:

- The traffic needed for load tests cannot be produced by only one tester.
- The System Under Test has heterogeneous interfaces.
- The System Under Test itself is distributed

	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 20 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

4.1.1 Distributed Test System for Load Tests

When performing load tests, a huge number of requests have to be produced. These requests should arrive at the SUT in parallel (especially in stress tests) to simulate many hundreds of users who are trying to access the SUT at the same time. When thinking of many hundreds or thousands of requests, this is really difficult to achieve. Of course, there are many kinds of load generators on the market performing exactly this, but it is extra hardware that has to be bought. Also, these traffic generators are very specialized and cannot be used for every needed SUT without causing new investments.

Instead of spending money on buying expensive additional hardware that has to be integrated first into the test system, existing infrastructure can be used. Using enough machines, it is possible to produce also heavy load. A simple example of how powerful this approach can be, are the different successful Denial of Service (DoS) attacks that have been thrown at big Internet companies in the past. Some examples are the DoS attacks on Yahoo, Ebay, Amazon and CNN that have been accomplished on February 2000. The result of the attacks was that the web sites of these companies were not accessible for hours and this caused a loss of many millions of dollars. This happened by using a huge number of computers around the world and attacking these web sites simultaneously, so that they were not able to handle these requests anymore.

Taking advantage of the numerous machines that test laboratories normally own, the previous approach would be a cheaper way to get a similar result. Of course, in cases where really huge numbers of requests have to be produced and not enough machines can be comprised, tests would have to be reverted to the expensive hardware solution (which in this case would probably also be distributed, i.e. more than one load generator).


4.1.2 Heterogeneous Interfaces

In some cases, more than one hardware interface may be needed in order to be able to communicate with the system under test, e.g. the SUT is distributed and every subsystem has a different hardware interface, or the SUT is local, but there are multiple hardware interfaces needed in order to establish a communication (e.g. a telecommunication switching center that has to be tested for handling analogous calls as well as ISDN calls correctly). As not every test system can be equipped with every hardware interface required, there must be a possibility to distribute the tasks to different test devices that support the needed type of communication. This applies to local, as well as to distributed SUTs.

4.1.3 Distributed System Under Test

Often, the SUT itself is distributed and complex which leads to the fact that it cannot be tested using a local test system. Of course, a solution would be to run tests for each subsystem of the whole complex system, which is often done in reality. But it is better to spread the test components between the SUT subsystems in order to stimulate and observe all the facets of the complex distributed system and to minimize signal delays due to long distances between the tester and the components of the distributed SUT. That means that routing the information back to the tester is replaced by the distribution of a test component to the remote testing location.

Especially in interworking scenarios it is necessary to place test components between the different subsystems that have to be observed to be able to get the needed results. One can think of a telecommunication company that wants to test all of its communication interfaces, e.g. UMTS, GPRS, GSM, SS7, and ISDN connections.

	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 21 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

4.2 GENERAL STRUCTURE OF A DISTRIBUTED 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 represented by the TTCN-3 Test 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 Testing Technologies’ TTthree compiler, it is an executable test suite, i.e. the compiled abstract test suite.

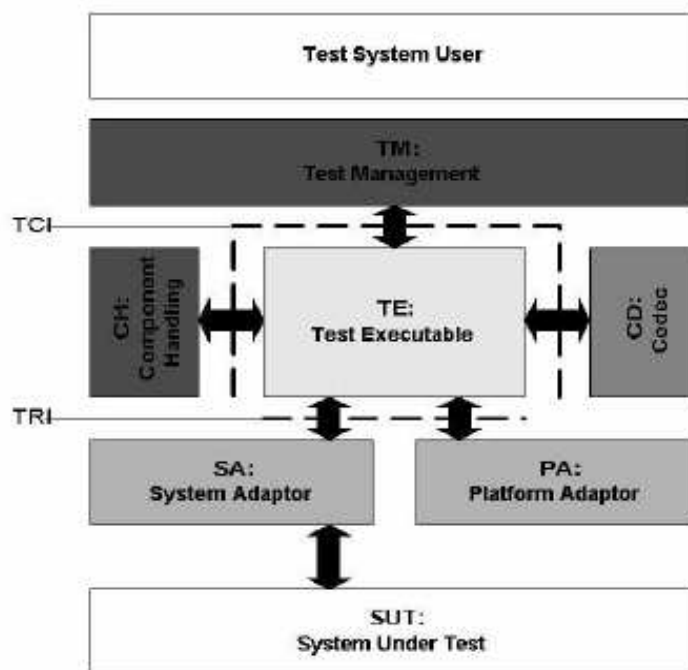


Figure 10: General Structure of a TTCN-3 Test System

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 Adaptor (SA) and Platform Adaptor (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 10, a TTCN-3 test system has two major interfaces, the TTCN-3 Control Interfaces (TCI) and the TTCN-3 Runtime Interface (TRI). The TCI 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 specifies the interfaces between the TE and Platform/SUT Adaptor entities, respectively.

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 11.

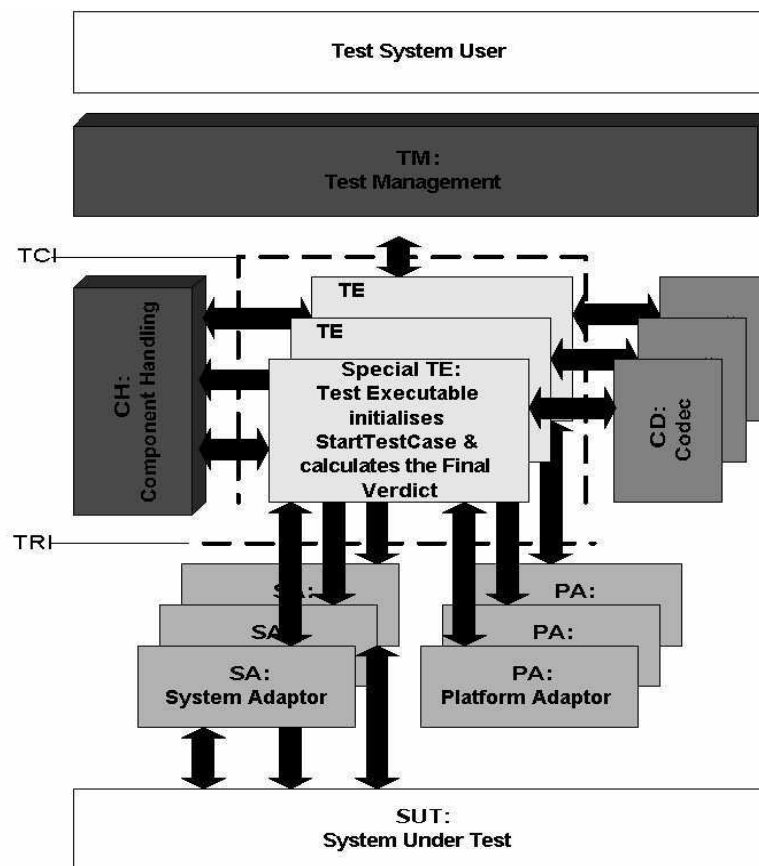



Figure 11: General Structure of a Distributed TTCN-3 Test System

On each node, a TE together with one SA, one PA and one CD exists. The entities CH and TM 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. Besides this, all TEs are handled the same way.

The CH controls the creation of MTC, PTC, and control components in TEs. The system component has a special role, as it exists only conceptually and not as a running test component in a TE. 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 means 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 for forwarding send request operations from a single TTCN-3

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 23 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

component that resides within a certain TE to the targeted component residing potentially in a 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.

Additionally, 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.

4.3 DISTRIBUTED TEST SYSTEM REQUIREMENTS

Before introducing the architecture of a distributed test system, the following requirements are needed as basic prerequisites:

- A middleware that undertakes the task of establishing and performing the communication between the distributed nodes
- An entity that contains the test system and that is part of the middleware
- A possibility to initialize all nodes participating in this test
- A description language that allows specifying the participating nodes and the sources which have to be deployed on these nodes in order to be able to execute a component's behavior.
- A description language that allows to specify where each component has to be deployed.
- An entity that processes the deployment configuration and the different distribution algorithms in order to distinguish the destination node of a new creating component.
- A possibility to distinguish between different test sessions in order to be able to run several sessions either by one user or also by different users

4.4 ARCHITECTURE

After the analysis of the needs in the previous sections it is time to present a distributed approach. The architecture of a distributed TTCN-3 Test System is displayed in Figure 12. It has been named TTmex (TTCN-3 Management and Execution platform) and its basic components are:

- Containers
- Daemons
- Session Manager
- Test Console

- Test System Entities (TE, TM, CH, CD, SA and PA)
- CORBA as middleware for the communication between all these components

4.4.1 Containers

Containers are the heart of the whole architecture. They contain the different test system entities TE, TM, CH, 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. The components cannot be accessed except via well-defined interfaces. The information about created components and their physical locations is stored in a repository within the containers. Containers are part of the middleware and are used for the communication between test components on different nodes.

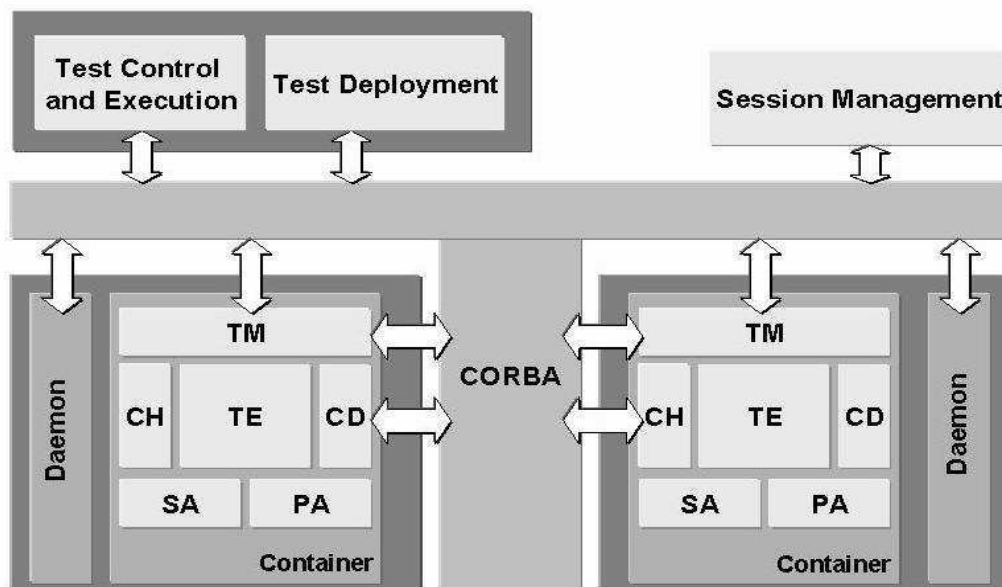



Figure 12: TTmex Architecture

4.4.2 Test Console

The Test Console is the control point of the platform and provides support for specifying TTCN-3 test cases, creating test sessions, deploying test suites and needed libraries into containers and controlling the test execution (start and stop of the test). It is also used for collecting the logs from different nodes.

4.4.3 Daemons

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 belonging to different sessions.

	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 25 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

4.4.4 Session Manager

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 for it, and all containers participating in 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 the 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 the destination node of a new component based on the appropriate distribution algorithm. 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 of the Special Test Container of the session.

4.4.5 Test System Entities

The test system entities are the different entities of a TTCN-3 test system. Their functionality has been described in the first part of this chapter. They reside within containers in order to be isolated from outside and thus, a container is their target execution environment.


4.4.6 Why CORBA as Middleware

As different middleware platforms are available on the market it had to be thought about which of them can be selected. The three architectures that have been under consideration:

- OMG's Common Object Request Broker Architecture (CORBA)
- SUN's Remote Method Invocation (Java RMI)
- Microsoft's Distributed Component Object Model (DCOM)

CORBA has been chosen 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 one to use clients and servers implemented in different programming languages (server in Java, client in C++); this is the main reason why Java RMI has not been chosen.
- There is already much experience with it due to the usage within other projects.
- CORBA is an open standard defined by many global key players (DCOM is Microsoft proprietary).
- It is available for the target operating systems Linux and Windows (DCOM is available only for Windows).

	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 26 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

A choice had to be made between different Object Request Brokers (ORBs) from different vendors. The ORB is the distributed service that implements the requests to the remote objects. Finally, the Java 2 ORB has been selected for the following reasons:

- The ORB had to be available for free without any licensing for usage within industrial products.
- As the whole TTthree runtime environment is specified in Java the ORB should also be in Java.
- The Java ORB is a standard part of Sun's Java 2 SDK from version 1.4 and upward and can thus be easily integrated into the whole architecture.
- It provides the needed CORBA services (Naming Service, Persistent State Service, and Concurrency Service).

4.4.7 Container Configuration File


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

4.4.8 Test Component Distribution Language

The test component distribution language (TCDL) is used to specify how components have to be distributed to the different containers. More concretely, it allows one to specify:

- Component description (Type)
- Component assembly
- Component mapping rules
- Component 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).

	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 27 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

5. REQUIREMENTS FOR THE ENVIRONMENT OF TESTING DISTRIBUTED EMBEDDED SOFTWARE

5.1 GENERAL OUTLINE

The following chapter discusses how the distributed test process control introduced in chapter 4 can be applied to testing distributed embedded software and proposes a list of general and special requirements in amendment to the ones presented in section 4.3.

5.1.1 Purpose

Distributed Software Simulation and Testing Environment (further denoted as SCT ENV) is needed to test embedded software in a workstation environment. This chapter outlines requirements for easing the implementation of proprietary testing environments for distributed embedded software.

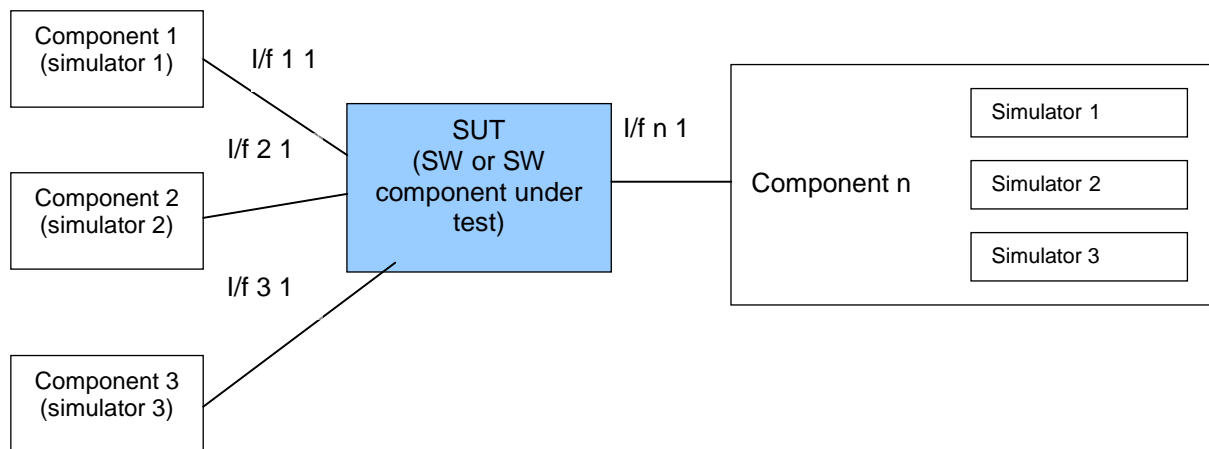



Figure 13: Distributed Software Simulation and Testing Environment

5.1.2 Rationales

When developing software for distributed embedded systems, there are numerous situations and reasons that call for a distributed test environment, like for instance:

- It is very difficult to automate and repeat tests in the target environment.
- It is impossible to test the single System Component (SC) independently from other system components.
- When performing system component testing (SCT) in the target environment, the tests depend heavily on other SCs.

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 28 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

- It is very hard to force neighbor SCs to the desired states in the target environment to run all possible test cases.
- It is expensive to support all possible and needed (different) hardware configurations in the target environment.

As a result, the test coverage is weak and many bugs remain in the distributed software after SCT when testing only in the target environment.

5.2 REQUIREMENTS

This chapter describes the requirements for an SCT test environment. It covers all needed prerequisites that the test environment has to meet. The implementation is an incremental process, where the requirements are fulfilled step-by-step.

The following levels of priority are assigned to the requirements: “MUST”, “SHOULD”, and “MAY”. These priorities can be used to justify the implementation targets for each requirement.

REQ#1: Single System Component software should be tested stand-alone from other SCs.

The test scope is Single System Component software and its interfaces towards other SCs. The entire SC software, or that of a particular hardware unit MUST be possible to test stand-alone from other SCs.

REQ#2: SCT should be carried out in a workstation environment.

Software development tools used in a workstation environment are more developed than tools used in the target environment. Debugging and memory usage analysis tools are easier to use in a workstation environment. Usage of a workstation environment is cheaper because there is no need for having lots of different hardware units for testing purposes as would be needed for target tests.

The test environment SHOULD be possible to run on a single workstation. The design decisions for minimizing host resource requirements (processing power, memory consumption and amount of windows) should be taken.


It MUST be possible to distribute the test environment to many workstations if needed.

REQ#3: General core components

Core components of an SCT environment MUST be general (meaning that it must be possible to use them for testing SUTs from different areas).

REQ#4: Only functional behavior of the software can be tested.

The environment MUST be used to test only the logic of the software (functional requirements). It is not required or even possible to test software real-time features in the SCT environment.

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 29 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

REQ#5: Exploratory testing

In addition to scripted automated testing the environment **MUST** enable manual intervention by the user in order to perform exploratory testing.

Exploratory software testing is a powerful approach, yet widely misunderstood. In some situations, it can be much more productive than scripted testing. All testers practice some form of exploratory testing, unless they simply don't create tests at all.

Exploratory testing is simultaneous learning, test design, and test execution. In other words, exploratory testing is any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests.

REQ#6: Debugging should be possible.

Source code debuggers, message level monitoring, heap monitoring, and RTOS level debugging should be possible to use. Stack usage of software threads should be supported.

Debugging without a burst of events must be provided by a synchronous timing model, where time in the system does not advance while any component/application is being debugged.

REQ#7: Analysis of memory leakage **MUST be possible.**

Memory leak denotes a software behavior, where dynamically allocated memory blocks are not returned to the memory pool after the program has stopped to use them.

When such behavior is carried out frequently (considering the duration of the program), and in relevant amounts (considering the size of the memory pool), this behavior forms a big risk to the stability of the software execution.

REQ#8: Usage of test coverage tools **SHOULD be possible.**

Rationale: Test coverage analysis tools give a feedback which areas of software are covered by test cases and which additional test cases should be created.


REQ#9: Hierarchical test cases **MUST be possible.**

Hierarchical test cases **MUST** be possible to use. It should be possible to start lower level test cases from the main test case and after completing the lower level test case, the test case control returns back to the main test case. The main test case can include many lower level test cases.

REQ#10: Master simulator

The whole test process is controlled by a master simulator, which has a special role compared to other simulators.

Rationale: Distributed embedded software is used in different hardware deployments. This deployment is achieved by a master simulator (means it is not hard-coded in any way)

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 30 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

REQ#11: Supporting different encoding rules and communication protocols by test tools

Test tools SHOULD support necessary codecs and communication protocols in order to minimize need for SUT specific test software development.

REQ#12: Script controlled testing, no manual intervention

Test scripts SHOULD be possible to execute from start to end without manual intervention by the test engineer. Manual intervention is needed in exploratory test cases only.

REQ#13: It SHOULD be possible to analyze test results automatically.

SCT environment can be used for regression tests to ensure that not only new features work well, but also old features still work after changes. Large numbers of regression test cases must be run during regression tests. The test cases must be well automated; otherwise they will not be used in practice. An important feature for automating the tests is automatic test results validation. The tester must receive the test case verdict in the form of 'passed'/'not passed'.

Note: Input events sequence and message parameters in different regression test runs should be fixed to meet the requirement.

REQ#14: Running tests MUST be done automatically.

Regression test case scripts running and results verification MUST be done automatically "by pressing one key" to run all tests.

REQ#15: SUT build MUST be as close to the target build as possible

Production code MUST be taken from the same storage as the official build.

REQ#16: The same data files as in the target environment MUST be used


Data files created for the target environment MUST be usable without any changes in simulated environment.

REQ#17: Simulators design principles

Simulators design should enable either using one simulator instance for all instances of particular unit or SC in SCT ENV, or easy duplication of already existing simulators to serve other units or SC instances.

REQ#18: Timing models

The timing model specifies how the local clocks of SCT ENV applications are synchronized and how the clock ticks are generated. An SCT environment must support synchronous and asynchronous timing models. Both timing models MUST rely on the central clock that ticks for all SCT ENV applications. The clock must send time control messages to all SCT ENV applications.

	Test Management and Distributed Test Process Control Deliverable ID: D.3.2.v1.0	Page : 31 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

REQ#19: Timing models MUST tolerate components leaving and joining the test environment.

The timing models MUST tolerate SCT ENV applications connection breaks because SCT ENV applications can join and leave the environment at any time. The connection breaks are caused either by connection failures or SCT ENV applications leaving the environment normally. The failures may be caused either by transport connection failure or SCT ENV application failure.

REQ#20: Asynchronous timing model

In an asynchronous timing model the timer ticks are generated asynchronously, that is, independently of the state of SCT ENV application. This means that simulation time advances even while the application is doing its processing.

In an asynchronous model, the time scale determines the time interval between clock ticks. This corresponds well to the real world, where a SUT process may not have completed its processing before the next tick occurs. Thus, it is possible for a high priority process to pre-empt a low priority process. An asynchronous model allows testing of free-running SUT processes (i.e., processes that never wait for anything).

The disadvantages of asynchronous timing model are:

- The tick interval is constant. If the SUT real-time dependent behavior changes due to increasing host load, increasing instrumentation or other factors, then it would be good if the tick interval is changed so that all operations triggered by an external event can be executed during the same tick. The optimal tick interval is hard to estimate and therefore a long enough and safe tick interval is chosen in practice. This may slow down the simulation environment significantly. If the workstation processor load is too high, the tests can fail to run.
- It is impossible to repeat tests so that events order and timestamps remain the same in subsequent successive test runs.
- Most of the clock ticks are totally useless, because they do not trigger any new execution neither in the SUT nor in the simulators. This means that some execution time is just uselessly wasted to advance the clock tick-by-tick.
- It depends on the workstation hardware configuration and load conditions.

The SCT ENV central clock sends new current time values to SCT ENV applications. The current time is the simulated time value that is 0 at the beginning of the testing environment start-up. After each *timeInterval* the central clock advances the time by *timeIncrement* milliseconds according to the following formula:

$$\text{current time [simulation step } n + 1 \text{]} = \text{current time [simulation step } n \text{]} + \text{timeIncrement}$$

The *timeInterval* is a real-time interval between two 'time' messages in milliseconds.

'Time' messages are generated for all SCT ENV applications at the same time.

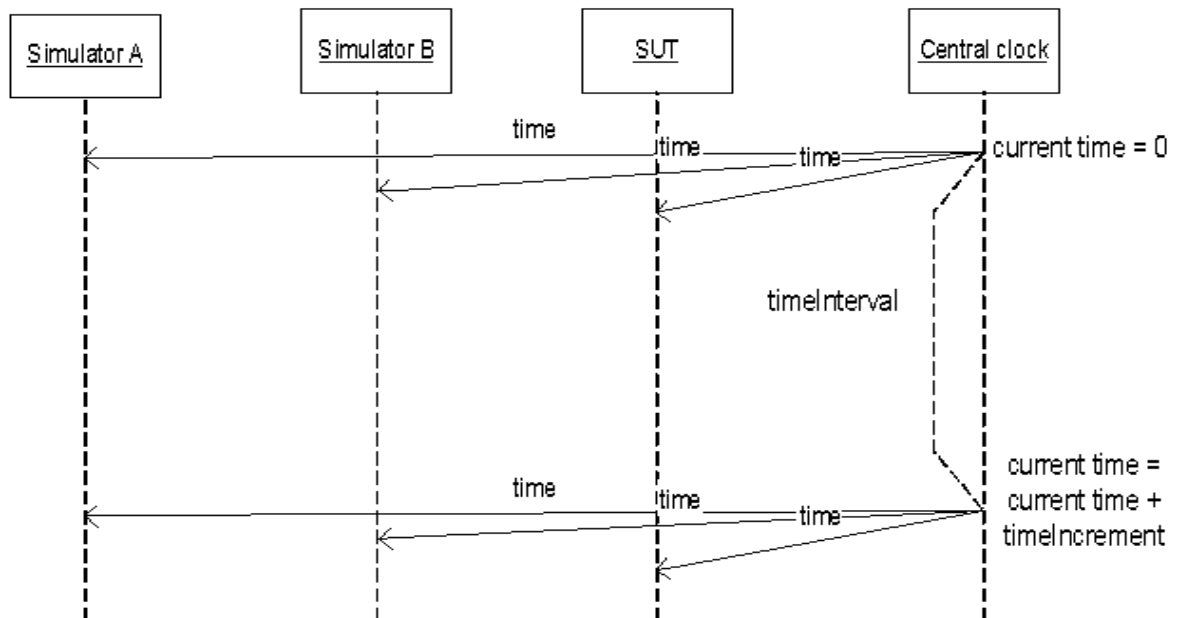



Figure 14: Asynchronous Timing Model Sequence Diagram

SCT ENV applications can join and leave the environment (launched or exited by the master simulator) at any 'current time' moment. Therefore SCT ENV applications have to cope with the initial time different than 0. Simulators, which need to react on time, should use relative time in scripts instead of absolute time. For example, when initial time received by the application is e.g. 8200, then it does not mean that 8200 ms has passed in life of the application.

REQ#21: Synchronous timing model

The synchronous timing model is based on an "infinitely fast processor model". Time advances only when all SCT ENV applications are waiting. This model does not allow free-running SUT processes, because time is not advancing at all during the processing of the free-running process. High priority SUT processes that might in the real world would be ready for execution before low priority SUT processes are not preempting the low priority processes in a synchronous model because time does not advance during execution of the low priority processes. This may sound like a serious limitation, but for testing SUT functionality, this model actually serves best.

The synchronous model is free from the disadvantages that asynchronous models suffer from. A synchronous model requires that each SCT ENV application knows its next event timer, which can be used to define the next step of simulation time. The synchronous model works in a way that the length of a simulation cycle is changed dynamically during execution according to the next event timer. If compared to the asynchronous model, the synchronous model makes the whole environment working much faster because there are no unnecessary simulated time cycles to wait for. The synchronous model provides exactly the same timestamps every time a test runs. This happens regardless of how heavily the workstation is loaded during the test.

	<p>Test Management and Distributed Test Process Control</p> <p>Deliverable ID: D.3.2.v1.0</p>	Page : 33 of 33
		Version: 1.0 Date : 12/11/2009
		Status : Final Confid : Public

In a synchronous model the defects due to the pre-preemptive nature of a real-time operating system are not detected, and free-running SUT processes are not allowed.

6. CONCLUSION

This document gave an overview of test management and test process control in regard to industrial testing needs for standalone systems and embedded devices.

Methodologies have been introduced and problems of test automation and distributed testing have been discussed, along with solution proposals.

The integration of Quality Assurance Systems into test processing tools has been demonstrated by a comprehensible example.

Finally, problems of a distributed embedded testing framework have been presented and respective requirements to prevent them have been proposed.

7. REFERENCES

- Elvior <http://www.elvior.ee>
- HP Quality Center <http://h50281.www5.hp.com/software/index.html>
- PikeTec TPT <http://www.piketec.com>
- Testing Technologies TTworkbench <http://www.testingtech.com>