Testing Real-time Systems using TTCN-3

Mayada Abdel-hak

A thesis
in
The Departement
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science
In Electrical and Computer Engineering
Concordia University
Montreal, Quebec, Canada

August 2006

Your file Votre référence
ISBN: 978-0-494-20738-3
Our file Notre référence
ISBN: 978-0-494-20738-3

ABSTRACT

Testing Real-time Systems using TTCN-3

Mayada Abdel-hak

In system engineering, testing plays an important role to validate systems and system components. As the market time becomes ever shorter and the requirements on system features, reliability, availability, integrity and performance further increase, assured quality of system and its components is very important. Moreover, in real-time systems extreme reliability and safety are the most fundamental requirements. Thus, in order to fulfill the requirements of such systems as well as those of the market, a systematic proficient approach to testing is an essential need. The Testing and Test Control Notation TTCN-3 has been developed by ETSI to address testing needs and to enable systematic specification-based testing for software systems.

This thesis will discuss in particular the use of TTCN-3 for testing real-time software systems. It takes textual test cases that have previously been generated for a particular real-time application. Each of them has input actions, time delays, and output actions. Then, the thesis explains the procedure of transforming the generated test cases into a test suite coded in TTCN-3. After that, it shows the process of creating and implementing the required TTCN-3 interfaces to complete the test system and follows it by running the test system and analysing the results.

To define the black-box testing procedure of these timed test cases, stimuli are applied on the system under test. Then, the reactions are observed and compared with the expected ones with respect to the time delays. According to this comparison, a test verdict assignment is set to determine the test behaviour of each test case. If the expected and the observed responses are matched while respecting the value of the related timers, a test verdict is set to "pass" indicating a successful test case. Otherwise, the verdict is set to "fail" signifying an unsuccessful test case.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Dr. Abdeslam En-Nouaary, for his support, guidance, and encouragement during all the stages of my research. I really appreciate his invaluable advices, directions and inspiring discussions that help me extend and deepen my knowledge in software testing and real-time system field.

In addition, I would like to thank all the members of the Examining Committee for their time and patience in reviewing my thesis.

Finally, I dearly thank my son and my daughter. This thesis is dedicated to them for their understanding, sacrifices, and hard working with a student mother which was not easy at all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

API	Application Programming Interface
ASN.1	Abstract Syntax Notation Number One (based-text notation)
ATS	Abstract Test Suite
CH	Component Handler
CORBA	Common Object Request Broker Architecture
ECD	External CoDecs
ETS	Executable Test Suite
ETSI	European Telecommunications Standards Institute 2005
FIFO	First In First Out
IDL	(COBRA) Interface Definition Language
ISO	International Organization for Standardization
ITU-T	International Telecommunication Union –Telecommunication Standardization Sector
IUT	Implementation Under Test
MSC	Message Sequence Chart
MTC	Main Test Component
MTS	Methods for Testing and Specification
OBSAI	Open Base Station Architecture Initiative
OMG	Object Management Group
OSI	Open System Interconnection
PA	Platform Adapter
PTC	Parallel Test Component

SIP	Session Initiation Protocol
SA	SUT Adapter
SUT	System under Test
TFL	Test Frame Language
TC	Test Control
TCI	TTCN-3 Control Interface
TE	TTCN-3 Executable
TID	Timer Identification
TIOA	Timed Input Output Automaton
TL	Test Logging
TM	Test Management
TRI	TTCN-3 Runtime Interface
TSI	Test System Interface
TTCN	Tree and Tabular Combined Notation
TTCN-3	Testing and Test Control Notation version 3
TT-Medal	Test and Testing Methodologies for Advanced Languages
U2TP	UML 2.0 Testing Profile
UML	Unified Modelling Language
XML	Extensible Mark-up Language

CHAPTER 1
Introduction

In a software life cycle, testing is one of the most significant activities. Its purpose is to make sure of the quality of the product or service being tested and to detect bugs. However, testing is an expensive and time-consuming phase. In addition, the significant growing complexity of systems and the lack of market time have added more exigencies to testing. Accordingly, the test process should satisfy the requirements of quality, reliability, and responding to the complexity of today's systems with minimal time and cost.

Real-time systems are special kind of systems that are restricted by time constraints. When testing such systems, particular difficulties arise due to the system behaviour restricted by these constraints, which adds another testing challenge. A real-time system must react to externally input stimuli within a limited and specified duration. Consequently, the correctness of system implementation depends not only on the logical results, but also on meeting the operational deadlines.

Therefore, the necessity for concrete systematic automated test methods that fulfill all the above requirements and response to the challenges is increasing. The Testing and Test Control Notation – Version 3 (TTCN-3) is a standardized testing language to pace and ease the specification and implementation of test suites for complex systems.

1.1 The Objective of the Thesis

In this research, we are going to develop a methodology for executing Timed Test Suite using TTCN-3 testing language. The starting point of our methodology is from a given text file of Timed Test Suite that has a collection of test cases for a real-time application. Our objective is to execute those test cases through the usage of TTCN-3.

Within the methodology, we will develop a program to automatically convert the test cases into a TTCN-3 Abstract Test Suite (ATS) produced as an output. The purpose of the methodology is to let the program automatically generate ATSs to save time, cost, and possible learning effort to the TTCN-3 language. In addition, we aim to make the program reusable for as many similar applications as possible. Once we compile the generated ATS under a TTCN-3 environment, the TTCN-3 Executable (TE) for that Abstract Test Suite is produced. The TE at this point is still abstract and portable between toolsets of different vendors who support concrete and language mapping. Therefore, we need some supplementary work to be able to run it.

Through the second phase of our development, we will complete the work and render the abstraction concrete. To make the TE and System Under Test (SUT) able to intercommunicate and understand each other, we need to encode/decode the messages exchanged between them from abstract data to real world data and vice versa. This is what we will implement as a part of the TTCN-3 Control Interface. Furthermore, the TE should adapt to a specific platform, test devices, and SUT, which we will also implement in the TTCN-3 Runtime Interface. By doing so, the test system is completed and ready to be applied on a simulated SUT. It will be run to automatically execute the test cases.

We will give more information about testing timed test suite using TTCN-3 in section 1.4 after introducing conformance testing since we are going to perform this type of testing. Thus, in the next section, we will talk about conformance testing and show its procedure as defined in the standardized Conformance Testing Methodology and Framework.

1.2 Conformance Testing

Distributed real-time systems communicate with the environment by exchanging messages. The manners of communication depend on communication protocols. So, a protocol implementation has to be compatible with other implementations [1]. To guarantee such compatibility, all implantation must conform to a well-detailed specification. The principle of testing the implementation is defined as Conformance

Testing. Nevertheless, different developers/testers could achieve this type of testing in different ways or may test the same system more than once. To avoid the previous problems, there is a necessity for a unified approach. The International Organization for Standardization (ISO) has developed and internationally standardized the Conformance Testing Methodology and Framework (CTMF) to cover the aspects of conformance testing. Although it focuses on conformance testing of OSI protocol entities, it has been used with success for functional testing.

Conformance testing objective is to determine whether an implementation conforms to a specification or not. The standard (ISO IS-9646) [2] defines the CTMF framework for conformance testing through specifying its main general procedure of test suites generation, test execution, and test result analysis. It also identifies the representation of test suite and test verdict. The standard recommends this general procedure of testing without targeting any specific testing protocol.

1.2.1 Conformance Testing Procedure

The procedure consists of three stages: test suites development, test execution, and result analysis. In Figure 1-1 [2], we illustrate this recommended conformance testing procedure.

1.2.1.1 Test Suite Development

A test suite is a set of test cases representing the test for particular test purpose. Test purposes are derived from the specification to identify what to test. Each test purpose covers one or a set of conformance requirements specified in the specification. Related requirements may be expressed as a single test purpose to be tested.

When test purposes become available, one generic test suite is derived from each test purpose. A generic test suite explains the high-level test actions to perform a specific test purpose, without taking into account the execution environment or the test method to be used.

At last, in order to generate an abstract test suite from the generic test suite, a specific test method should be considered besides the restrictions implied by the test environment. This is because the generated ATS should be adaptable to prospective real test architecture. An overview of test methods and their architectures is given in section 1.2.2.

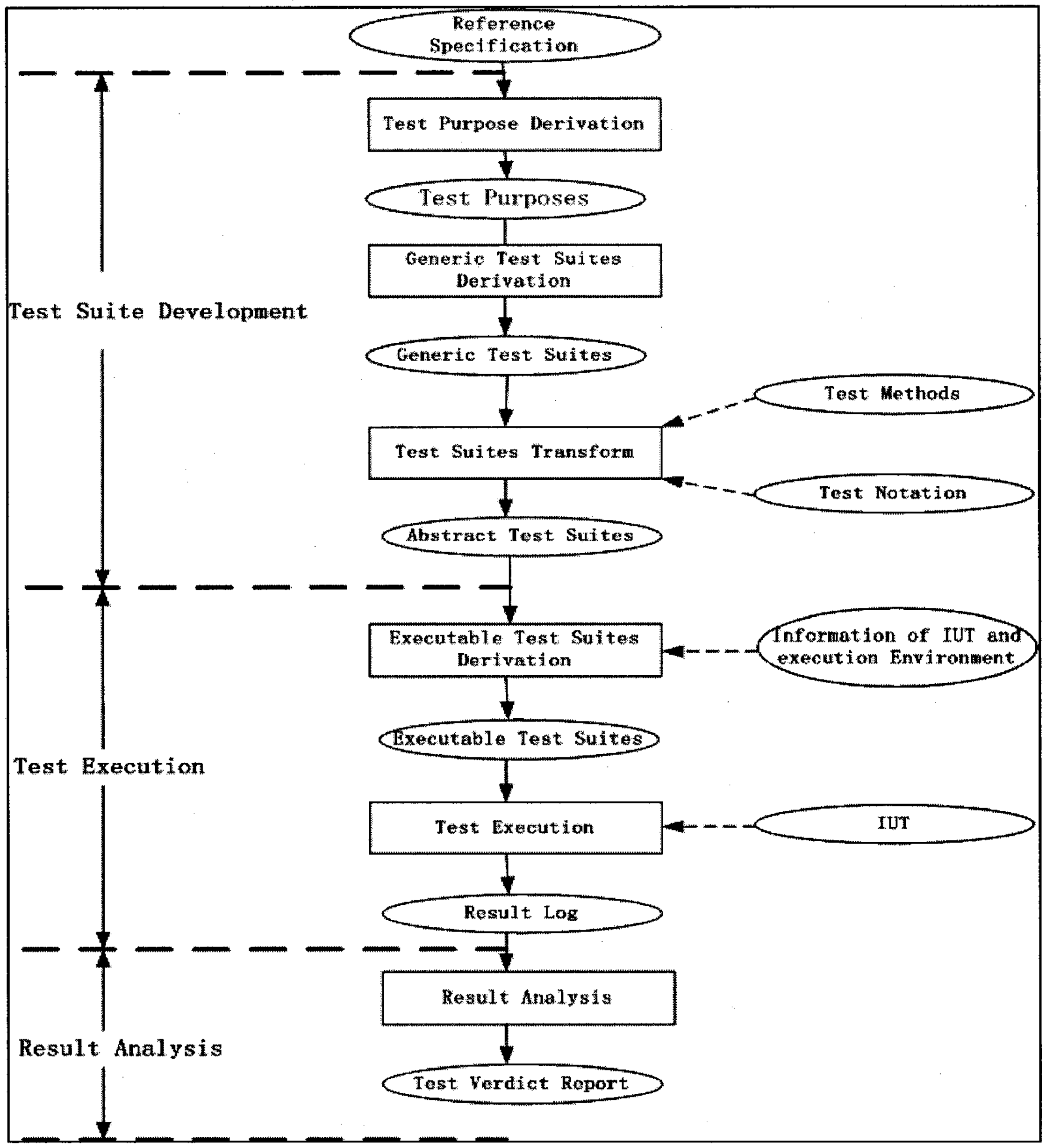


Figure 1-1: Conformance Testing Framework

The resulted ATS is independent of any implementation and the test cases in an

ATS are represented in a well-defined test notation. TTCN, the tree and Tabular Combined Notation, is a semi-formal standard language to specify abstract test suite is suggested by ITU-T in its recommendation [3], as we will see in section 1.4.

1.2.1.2 Test Execution

The ATS is independent of any real time testing environment and any IUT; hence, the data and parameters in the test cases are abstract and not real data. Due to this, the ATS can not be applied on real test devices until it is transformed to executable test suite with concrete data. Therefore, it is time to know about the system under test implementation and the test environment. A test case, for instance, could be passed as the parameters of a function, or the payload of a packet. Additionally, if there are some test cases in the ATS that could be irrelevant to the implementation owing to some options offered in the specification, they should be deselected before the transformation.

When the executable test suite has been created and ready for execution, it is applied to the IUT. The reactions of the IUT are observed and logged.

1.2.1.3 Result Analysis

The recorded reactions of IUT are compared with the expected ones as specified in the test suite, and a verdict report is generated. A verdict can have one of the three values: PASS, INCONCLUSIVE, or FAIL. If the outputs of each test case indicate that the implementation conforms to the specification and test purpose, a PASS verdict is concluded. In case the implementation conforms to the specification but not to the test purpose, INCONCLUSIVE verdict is concluded. However, if the implementation fails to conform to the specification, a FAIL verdict is concluded. If all test cases applied on the IUT lead to the verdict PASS, it means that the IUT conforms to its specification.

1.2.2 Test Methods and Configuration

The IS 9646 standard has recommended the four basic types of test methods. Each method represents different logical concepts of test architecture with regards to the accessibility of the IUT to the tester as depicted in Figure 1-2 [4].

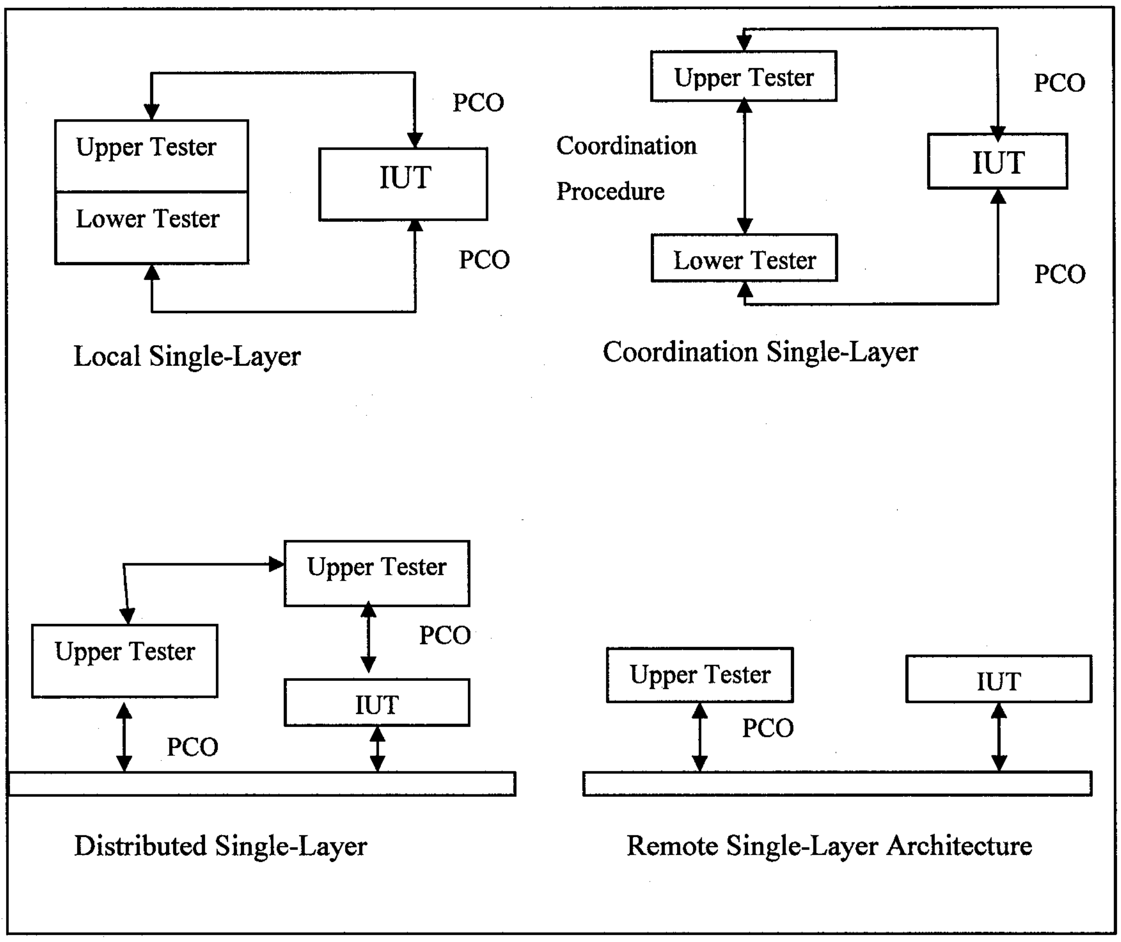


Figure 1-2: Basic Types of Test Methods

For any test architecture, there are an Upper Tester (UT), Lower Tester (LT), Implementation Under Test (IUT), and Points of Control and Observation (PCOs) [4]. The UT behaves as the upper layer of the IUT that enquires services from the IUT via the PCO on the upper boundary. While the LT acts as a lower layer of the IUT and its peer-layer. It creates services and sends signals to control the behaviours of the IUT via PCOs. It also generates the test verdict and test report. PCOs are service access points to enable the tester observing and controlling the IUT behaviour. In a concrete test configuration, the UT and LT representing the functional entities are mapped to one or more test components, e.g. active elements which are executed in parallel. Each test component realizes the functionality of one or more UTs and LTs. The lower tester control function is integrated in the main test component (MTC). The number of test components depends on the degree of concurrency in the behaviour of the tester. In a multi party context, one test component should be defined for each party. However, it should be noted that, in principle, several test components again may be executed on the same physical device.

The four testing methods are: Local Single-Layer (LS), Distributed Single-Layer (DS), Coordinated Single-Layer (CS), and Remote Single-Layer (RS) Method. In the LS method, both lower and upper testers as well as the IUT reside in the same system. In the DS method, the LT is at the remote location and requires communication with IUT through the PCOs of the low layer service provider. It also communicates with the UT through a Test Coordination Procedure in order to apply a proper test suite to the IUT and reports related outcomes. The CS method is similar to the DS one except that both testers are in the same location, but separated and coordinate through the local coordination procedure. The RS method differs from the DS in that there is no upper tester and the remote tester has an access to only one PCO.

1.2.3 Conformance Testing and Specification

The amount of information given about the system implementation determines the way in which tests can be specified. Two types of testing can be distinguished with regards to the implementation: black and white-box testing. Conformance testing is a black-box testing most of the time. This type of testing ignores the internal structure of the implementation. The test cases are generated based on the specification, and then they are applied to the implementation. The implementation is tested against the specification to verify its conformance to it. In addition, the test fault coverage and result analysis are checked with respect to the specification.

Unlike black-box testing, in the white-box testing, the implementation and the internal structure are known. They are used along with the specification for the test cases derivation, fault coverage, and test result analysis.

Another type of testing is called grey-box testing where only high-level module structure of the implementation is acknowledged without any details.

Besides conformance testing, there are many other types of testing such as interoperability, robustness, regression, system, and integration testing.

1.3 Testing Real-Time Software Systems

Real-time systems are usually large and complex systems that need extreme reliability and safety. They have real-time control and concurrent control of separate systems that communicate with each others and with the environment. Hence, these systems must react to input stimuli from the environment within a finite and specified period.

Testing real-time systems is subject to real-time constraints, which are the operational deadlines from event to system response. For instance, emerging multimedia over internet is considered one of its applications. Non real-time systems do not have deadlines, but quick responses are desired and preferred to obtain high performance. In contrast, the correctness of the functionality of a real-time software system depends not only on the logical results of the system, but also on the time at which these results were delivered. Therefore, in these systems, the emphasis is on both scheduling and performance. Failure to respond through a specified delay is as bad as the wrong response.

It is extremely crucial in hard real-time systems that responses happen within the required deadline such as flight control systems, missile guidance systems, medical device monitoring, and power plant control. If a time constraint is violated or a deadline is missed, the failure will be disastrous on human lives as well as on the environment. Such applications of real-time systems are considered mission critical and testing them is one of the challenging research areas.

As mentioned before regarding modern complex systems, testing is a more costly and time-consuming process. Testing scenario will be worst when we test real-time systems because time is not under the control of the tester and the timing factor produces too many states [5]. Before concrete tests can be carried out on a system, big efforts should be spent on specifying what and how to test and on getting the test descriptions in a format that is accepted by the test devices [6].

To reach such kind of testing, three main steps could be recognized in general. First of all, test cases are generated using an efficient method especially for real-time applications called *Automated Timed Test Suite Generation.* This step or method is not part of our work, but its final output will be our starting point. Thus, in chapter 3, we will talk briefly about its process of generation without any implementation or details. This is just to give the reader an idea about it and show how and why our work will be the continuation for it. In this method, timed test cases are generated based on the specification of the system and test purposes. As mentioned before, test purpose is the partial functionality of the system under test. Its objective is to reduce the number of test cases while ensuring acceptable fault coverage. Each test case represents a sequence of interactions among the components of the system and their environment as well as the time constraints on these interactions. The construction of use cases employs timers and time constraints between each pair of events in order to specify the timing behaviour of a real-time system. As a result of this generation, a collection of test cases is written into a file. Each test case consists of input actions, time delays, and output actions. Time delay is the maximum duration allowed between two actions (2 inputs, input-output, or 2 outputs). Our work will start with this file and include the following steps.

In the second step, the generated test cases are applied to the implementation of the system under test. For each time delay in a test case, we set a timer and we wait for its expiration before processing the next event, then the reactions of IUT are observed. Finally, the test results are analyzed and a verdict is concluded. If the outputs of each test case match those expected with respect to corresponding timers value, the implementation is conforming to the specification and test purpose. Otherwise, faults within the implementation must be sited and fixed.

The approach adapted in the first step is not part of our work, but its final output will be our starting point. Thus, in chapter 3, we will briefly talk about its process of timed test suite generation to show the reader how and why our work will be the continuation to it. However, the implementation and details of the generation are out of the scope of this thesis. Then, based on the outcome of this step, we will proceed our development for the second and third steps in this research.

1.4 Testing Timed Test Suite with TTCN-3

The manual testing methods used before cannot ensure the reliability and quality of the actual diverse complex systems within acceptable time and cost. Today, testing of software intensive systems is in many cases done in an ad-hoc fashion with languages not designed specifically for testing. In the third part of Conformance Testing Methodology and Framework, the Tree and Tabular Combined Notation TTCN is defined and recommended as a test specification language [7].

Later, TTCN-3 has been developed and internationally standardised as a test language. It is used to test the specification and implementation of complex test systems (e.g. Distributed and reactive systems) for different kinds of applications. However, TTCN does not include OSI or conformance testing specific constructs in its third version. Instead, it provides many new compelling features as a modern testing language. To tackle the increasing requirements of today's systems, automating testing of test suites using the standardized testing language TTCN-3 is adapted.

Our approach will focus on developing a methodology to execute Timed Test Suite using the TTCN-3 testing language. We are going to apply the methodology on a real-time multimedia application. The program that we will develop in our methodology starts from a provided set of test cases in a file generated as mentioned in the previous section and will be our input. The program will automatically transform the test cases into a TTCN-3 ATS. The aim of our methodology is to be able to use the program as ATSs generator to save time, effort, and cost that would be invested if the ATS is built manually by training developers or testers for this new language. In addition, we want to apply the reusability software concept and reuse the program or the TTCN-3 ATS for as many similar applications as possible with little modification if needed. The output of our program will be a file containing the TTCN-3 ATS. When we compile this ATS under a TTCN-3 environment, we get the TTCN-3 Executable

(TE) for that abstract test Suite. The ATS / TE are intended to be abstract to make the tester focuses on the specification of the test independently of any implementation, language, or platform. Hence, the TE is portable and supported by the toolsets of different vendors who provide language mapping, and that is why the TE cannot be run at this stage.

Therefore, the generation of the ATS with its TE is the core part of our work, but not all. We will complete the work through the second stage of our development when we render the abstraction concrete. To do that, the executable needs adaptation to a specific platform, and system under test, which we will implement through the TTCN-3 Runtime Interface. Additionally, the ATS needs encoding abstract data to real world data and decoding the real world data to abstract. Encoding is required, so that the System Under Test (SUT) be able to understand the stimuli coming from the ATS. On the inverse direction, decoding is needed, so that the responses (to the stimuli) sent from the SUT to the ATS can be understood by the ATS. This is also what we will implement through the TTCN-3 Control Interface. As a result to our development, we will have a complete test system that we apply on a simulated SUT. Then, we run the test system, which will automatically execute all the given test cases.

However, due to the fact that the TTCN-3 testing language is still new, we will talk about it before starting our development. This is to make it easy to the reader to follow our work. Thus, the thesis will first explain the structure of the language and its elements. It also highlights its features and mechanisms such object-oriented like syntax, timer handling, dynamic test configuration and communication, test behaviour, and test verdicts. These concepts will be used by the program generating the TTCN-3 abstract test suite. We will also explain where we get the file of timed test suite and why we consider it our starting point.

1.5 Thesis Outline

This thesis consists of five chapters. The remainder of this thesis is organized as follows: Chapter 2 talks first about TTCN-3 core notation. It starts with a background. Then, it introduces the third version, its objective and applicability, presentation formats, essential capabilities. After that, it explains the TTCN-3 core language structure, main parts, and how it is used to create a TTCN-3 abstract source code and produce the executable. Then, the chapter describes the structure of a whole TTCN-3 test system as a collection of conceptual interacting entities. Each of which has a functionality to perform a specific role in the test system implementation. According to its functionality, an entity belongs to either the executable of the abstract test or to one of the two related interfaces: TTCN-3 Control Interface and TTCN-3 Runtime interface. The chapter will then talk about each involved entity along with its task.

Chapter 3 focuses on our methodology for the development of a TTCN-3 test system to automatically execute given timed test cases. In this chapter, we will first illustrate and give an overview of the process of building a TTCN-3 test system. Then, we will explain the approach of timed case generation that has already been done by the others to generate a file of test cases to start our development with. Then, we will begin our methodology by creating a program to generate a TTCN-3 ATS. After that, the compilation of the ATS and the production of the TTCN-3 Executable are described.

Chapter 4 is devoted to the design, implementation, and execution of the test system. In this chapter, we will complete the development of our test system to be able to run the given test cases of a real-time multimedia system. The chapter starts with explaining the object oriented design to the packages and classes involved in our development. Then, we will give an overview of our test implementation. After that, we will implement the classes used for this development and build them. At the end of the chapter, we show how to run our test system to automatically execute the test cases and analyse the results.

Finally, in chapter 5, we will summarize the thesis and our contribution based on the work and the results of this research. Then, we will discuss the future extensions related to our work. Specifically, we will talk about the *TIMED*TTCN-3 extension to efficiently address the hard real-time requirements and the *TIMED*GFT extension to graphically represent the *TIMED*TTCN-3.

CHAPTER 2
TTCN-3 Core Language and Test System

This chapter first talks about the language history. Section 2.1 gives an idea about TTCN-3 background. Section 2.3 provides an overall view of TTCN-3 including its objectives and applicability, features, presentation formats. Then, section 2.3 explains the structure of the core language according to the functionality of its different elements. The core language enables us to create a TTCN-3 Abstract Test Suite/module and produce its TTCN-3 Executable (TE). The latter is just part of the test implementation and the first step of the process. To be able to execute a TTCN-3 test system, the whole test system structure should be built. Section 2.4 will describe the structure of a TTCN-3 test system based on its various functionalities where the TE is its center as given in the first subsection. Completing the structure requires a TTCN-3 Control Interface (TCI) and a TTCN-3 Runtime Interface (TRI), which will be explained in the other subsections along with their corresponding entities and sub interfaces respectively.

2.1 Background

The Tree and Tabular Combined Notation (TTCN) was developed and standardized as a black-box functional testing language. Its objective was to describe tests in an effective unambiguous manner that is simple to be used by testers. Some enterprises applied the language for critical testing to their products under development and found it effective.

Progressively, its success was the reason for the development of the next version, TTCN-2, using the conformance testing methodology. Standardization organizations and forums such as ETSI and ATM have developed test suites for their specifications using TTCN-2 with success. In addition, many companies started developing their own test suites as a complement to the public test suites. Once again, that success and good experience with the second version led to improve it in order to meet the testing needs for other applications in many different areas like internet and software.

2.2 Overall View of TTCN3

The most recent version of the language, TTCN version 3 (TTCN-3), is developed, standardized, and maintained by European Telecommunication Standards Institute (ETSI: ES 201 873 series). The International Telecommunication Union – Telecommunications (ITU-T) has published the language as the recommendations (ITU-T: Z.140 series) [10] and adopted it as well.

2.2.1 TTCN-3 Objective and Applicability

The Testing and Test Control Notation TTCN-3 has been developed to write detailed test specifications that address testing needs of modern Telecom/Datacom and IT technologies, and to widen the scope of applicability. One of the objectives of TTCN-3 is to enable systematic specification-based testing for software systems based on CORBA, EJB or XML technologies.

TTCN has been used to specify tests for many kinds of applications, including mobile communications (GSM, 3G, TETRA), wireless LANs (Hiperlan/2), cordless phones (DECT), Broadband technologies (B-ISDN, ATM), CORBA-based platforms, internet protocols (IPv6, SIGTRAN, SIP and OSP), software Modules, services testing (including supplementary services), and APIs [11].

TTCN-3 does not include OSI or conformance testing specific constructs, but it provides many other concepts. This is to widen its usability for many types of testing such as interoperability, robustness, regression, system, and integration testing besides conformance testing.

2.2.2 Essential Features

TTCN-3 did not reinvent; instead, it retained the well-proven concepts of TTCN-2, improved the others, and reserved the expertise of version-2 developers. The following outlines these features [20]:

- **Triple C**
    - **Configuration**: Dynamic concurrent test configurations with test components.
    - **Communication**: Various communication mechanisms (synchronous and asynchronous).
    - **Control**: Test case execution and selection mechanisms.
- **Improved**
    - Harmonized with ASN.1.
    - Module concept.
- **Extendibility** via attributes, external function, external data.
- Well-defined object-oriented like syntax (looks like Java/C++).
- Static and operational semantics.
- Different presentation formats.

2.2.3 Different Presentation Formats

The TTCN-3 standard offers three different presentation formats: textual, tabular, and graphical format illustrated in figure 2-1.

Usually, the text-based Core Language (ES 201 873-1) is the normal choice for who got used to use conventional programming environment [13]. With this format, developers can effectively specify test types and values. In addition, it is considered as a standardized interchange format of TTCN-3 test suites between TTCN-3 tools. It also serves as the semantic basis for different presentation formats.

However, there were considerable investment for the already made tabular format in TTCN and ES 201 873-2 [17] of the previous version and there are users familiar with it. Therefore, developers of TTCN-3 were well aware of that and have defined a tabular format (TFT).

Moreover, TTCN-3 came up with the third standard representation, ES 201 873-3 [18] graphical format (GFT). This format is helpful when the focus is on testing purposes and definition of dynamic behaviour in order to draw complex test cases rapidly and create appealing documentation in parallel. GFI is also a good choice to visualize test executions in case errors have been found.

In addition to previous formats, if there is a need for other presentation ones, the language facilitates the development of non-standard formats.

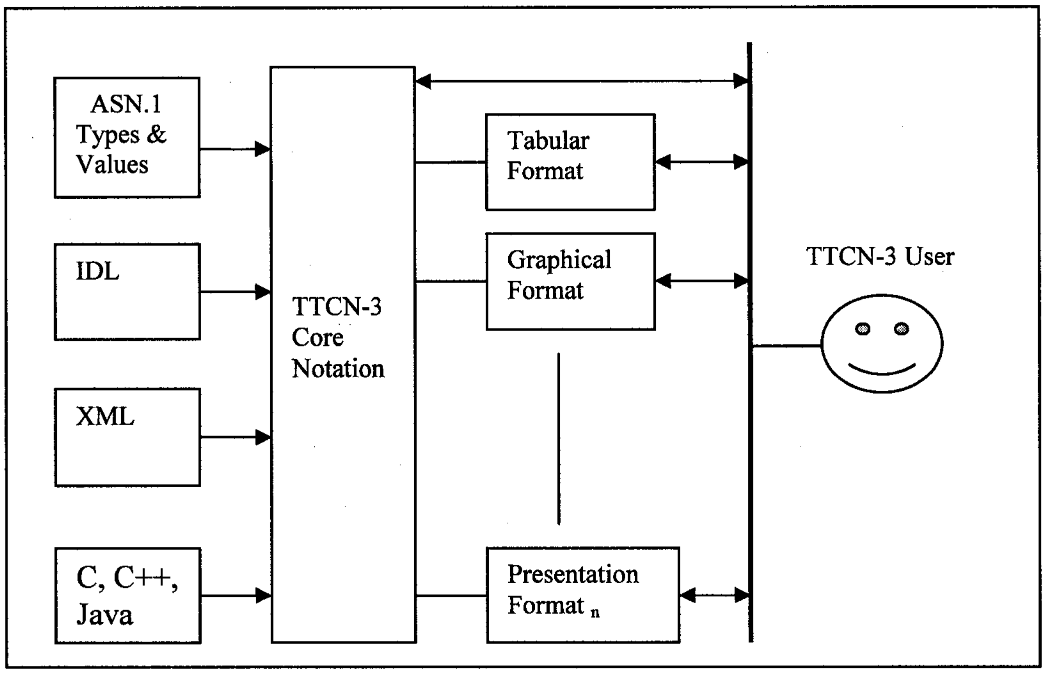


Figure 2-1: TTCN-3 formats and mappings

The core language may be used independently of the presentation formats. However, neither the tabular format nor the graphical format can be used without the core language.

2.3 TTCN-3 Core Language Structure

The top-level unit of all TTCN-3 test specifications is called a *Module*. It is the building block that defines an executable test suite [14]. A module cannot be structured into sub-modules, but it can import other modules or definitions from other modules. The language allows test suite parameterization through using module parameters option. A test suite is one or more modules that contain a completely defined set of test cases. Each module consists of a module definition part and an optional module control part that will be explained in the following sub sections.

2.3.1 Module Definition Part

As the name indicates, the definitions part of a module defines data types, constants, test data templates, test components, communication ports, functions, signatures for procedure calls at ports, test cases, etc. TTCN-3 language does not support global variable. However, the variables defined in a test component can be used by all test cases and functions running on that component. According to its different functionalities, the definition part could be divided into three main parts: Type and Templates definitions for test data, components and ports for communications, and test cases and functions for test behaviour

2.3.1.1 Module Definitions Part

Table 2-2 gives an overview of the types of TTCN-3 [14]. The type definitions are needed for test data structures.

Class of type	Keyword
Simple basic types	integer
	float
	boolean
	objid
	verdicttype
Basic string types	bitstring
	hexstring
	octetstring
	charstring
	universal charstring
Structured types	record
	record of
	set
	set of
	enumerated
	union
Special data types	anytype
Special configuration types	address
	port
	component
Special default types	default

Table 2-2: Overview of TTCN-3 Types

TTCN-3 has a number of pre-defined basic data types as well as structured types such as records, sets, unions, enumerated types and arrays. It also defines normal and external constants. These type definitions are global to the whole module.

2.3.1.2 Templates Definitions

A template is a special kind of data structure which is necessary for concrete test data. It provides parameterization and matching mechanisms for specifying test data that should be transmitted or received during the test via the test ports. Template can be used to transmit a set of distinct values and/or to check if a set of received values match the template specification.

2.3.1.3 Ports and Components Types Definitions

A module has two kinds of test components: the Main Test Component (MTC) and the Test Parallel Component (PTC). Within every test configuration there must be only one MTC while we could have many PTCs. The Test System Interface is also considered as a third component. Figure 2-2 shows a configuration to a test system consists of the MTC connected to two other test components [14]: PTC1 and PTC2 from one side and communicated with the SUT from the other side. Usually, the object being tested is called IUT, which could represent a direct interface for testing or a part of the system. Therefore, this object means either IUT or SUT and generally called SUT. The PTC1 and PTC2 should also communicate with the SUT. In addition, the figure illustrates the test system interface as the boundary of the test system to the SUT.

The single MTC is created automatically by the test system at the beginning of each test case execution. But, PTCs should be created dynamically during the execution of a test case using the *create* operation explicitly. The behaviour defined in the body of the test case will be executed on the MTC component.

Each test component may have its own local ports and timers. During a test case execution each test component has its own behaviour and hence several test behaviours may run concurrently in the test system. Therefore, a test case can be seen as a collection of test behaviours. Test case execution should end when the MTC terminates [14]. When the MTC terminates, the test system has to stop all PTCs not terminated by the moment when the test case execution is ended [14].

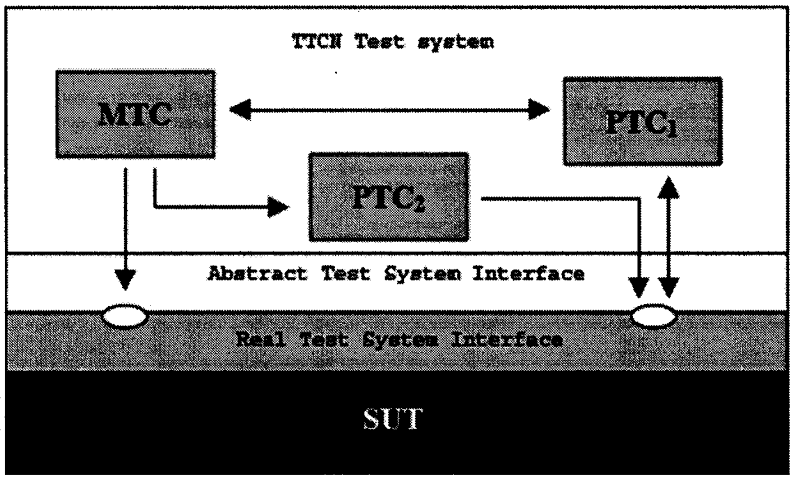


Figure 2-2: TTCN-3 Components

Test components can communicate with each others via abstract mechanisms called communication ports. The operations on these ports provide both message-based and procedure-based communication capabilities. The principle of procedure-based communication is to call procedures in remote entities. On the other hand, message-based communication is based on an asynchronous message exchange.

The configuration and connection among the components themselves through the communication ports are defined by using *create* and *connect* keywords and occur at test run. While the connection between the components and the test system interface is dynamic and can be modified during the test run using *map* and *unmap* keywords. Each port is modeled on the receiving side as FIFO queue to store the incoming messages or procedure calls until they are processed by the component of that port as illustrated in figure 2-3 [14].

Test component types and port types indicated by the keywords *component* and *port* should be defined in the module definitions part. However, the actual configuration of components and the connections between them (performed by *create* and *connect* operations) or the communication with the test system interface (by means of the *map* operation) are within the test case behaviour.

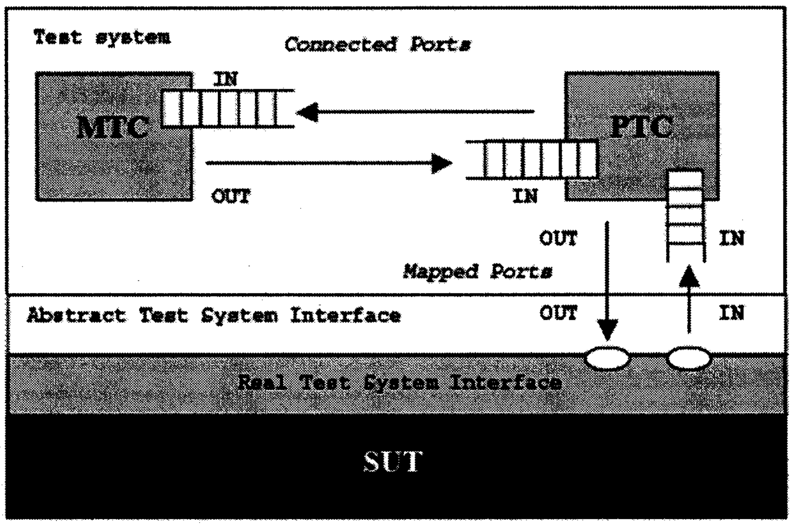


Figure 2-3: Communication ports

TTCN-3 allows the dynamic specification of test configurations. A configuration consists of a set of inter-connected test components with well-defined communication ports and an explicit test system interface, which defines the borders of the test system [13].

In real test environment, test cases need to communicate with SUT, so we need to define the number and type of all possible communication ports that connect a use case to SUT at the test run. Each test case should be accompanied with an abstract test system interface (TSI) to define the borders of the test system.

The test configuration along with its dynamically created test components and communication topologies to the SUT are considered one of the powerful features of TTCN-3.

2.3.1.3.1 Timer as a Related Configuration Tool

Timers are usually declared in component type definitions can be declared in the module control part, test cases, functions and altsteps, which are running on the given component type. A timer declaration may be assigned by default to a non-negative float value where the base unit of duration is measured by seconds.

TTCN-3 supports timer handling by using a number of timer operations shown in table 2-3 [14]. In each TTCN-3 scope unit timers declared there are two conceptual local lists: *running-timers list* and *timeout-list*. The timeout-lists are part of the snapshots that are taken when a test case is executed. The two lists belonging to a unit are updated when a timer is started, stopped, or timed out in the scope of that unit.

Timer operations	
Statement	**Associated keyword or symbol**
Start timer	**start**
Stop timer	**stop**
Read elapsed time	**read**
Check if timer running	**running**
Timeout event	**timeout**

Table 2-3: Timer operations

When a timer expires, it is conceptually added as an event to the timeout list where each timer has just one entry, and the timer is deactivated right away for that scope unit.

2.3.1.4 Test cases and Functions definitions

Functions may be used to do some calculation or to specify test behaviour using the communication operations (e.g. send, receive). A test case is a special kind of function to represent a collection of test behaviour describing the MTC behaviour. The header of the test case should define two parts: the interface and system. The interface part references the MTC on which the test case will run. The system part refers to the test system interface and it may be optional if the only component used in a test case is the MTC.

When a test case is invoked, the MTC is created, the ports of MTC and test system interface are instantiated and the behaviour specified in the test specification is started on MTC. All the actions are preformed implicitly without any operation (no *create* or *start* is needed). At the beginning of a test case, the test configuration destroys all previous operation on precedents test cases, so they become invisible to the new test case. A test case starts by the use of *execute* statement in the module control part. It finishes with the MTC termination which stops all running PTCs and removes them from the system. Then, the final test verdict will be calculated on the basis of the local ones of all test components and the result of a test case is a value of type *verdicttype*.

2.3.1.4.1 Test Verdict a Related Behaviour Tool

Local verdict is an object created for each test component at the time of its instantiation in the MTC and every PTC with the initialized verdict value *none*. The *setverdict* and *getverdict* are the only test verdict operations. The verdict can have five different values: *pass*, *fail*, *inconc* (inconclusive), *none* and *error*. When changing the value of the local verdict using the *setverdict* operation for example, the effect of this change must follow the predefined overwriting rules listed in the table below [14].

Current value of Verdict	New verdict assignment value			
	pass	inconc	fail	none
none	pass	inconc	fail	none
pass	pass	inconc	fail	pass
inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail

Table 2-4: Overwriting rules of test verdicts

These rules make sure that, e.g., a fail test verdict does not become a pass during test case execution. When the execution of all test components for a test case terminates, the global verdict of the test case will be updated according to those rules and returned. The effect of this implicit operation should also follow the overwriting rules shown in the same table.

Example :

```
:
setverdict(pass);        // The local verdict is set to pass
:
setverdict(fail);        // When this line is executed, the value of the local verdict is
:                        // overwritten to fail
:                        // When the PTC terminates, the test case verdict is set to fail
```

2.3.2 Module Control Part

Test cases are defined in the module definition part while calling them as well as managing and controlling their execution should be done within the module control part. Therefore, the control part is considered the dynamic behaviour of the TTCN-3 specifications.

The control of test execution describes the relations between test cases such as sequences, repetitions and dependencies on test outcomes. For this purpose, selection and /or iteration statements such as (if-else, do-while), and conditions for starting a test case may be used. The control part can declare its own dynamic elements (local variables and timers) in its declaration part.

2.4 General Structure of TTCN-3 Test System

Conceptually, a TTCN-3 test system can be seen as a set of entities that interact with each other. Each entity has a specific functionality in the test system to implement as follows [19]:

- Test Execution (TE): Execute or interpret compiled TTCN-3 code
- Test Management (TM): Manage test execution.
- Coder/Decoder (CD): Administer types, values.
- Test Logging (TL): Handle test components.
- Test Component Handler (CH): Handle test components.
- SUT Adapter (SA): Perform suitable communication with the SUT.
- Platform Adapter (PA): Implement external functions and handle timer operations.

Figure 2-4 [20] shows the general structure of a TTCN-3 test system implementation as a group of entities, which will be explained soon [22].

The TTCN-3 Executable (TE) entity is considered the heart of a test system. It implements a TTCN-3 module on an abstract level using the core language. The other interacting entities of a TTCN-3 test system make these abstract concepts solid [22].

Besides the TE entity, the TTCN-3 test system has two major interfaces, the TTCN-3 Control Interface (TCI) and the TTCN-3 Runtime Interface (TRI). The purpose of both interfaces is to deal with all other features that can not be concluded from only the information in an abstract test represented by a TTCN-3 module. Table 2-1 depicts an overview of the test system interrelation [11]. It also shows which entities belong to each of the two interfaces TCI and TRI.

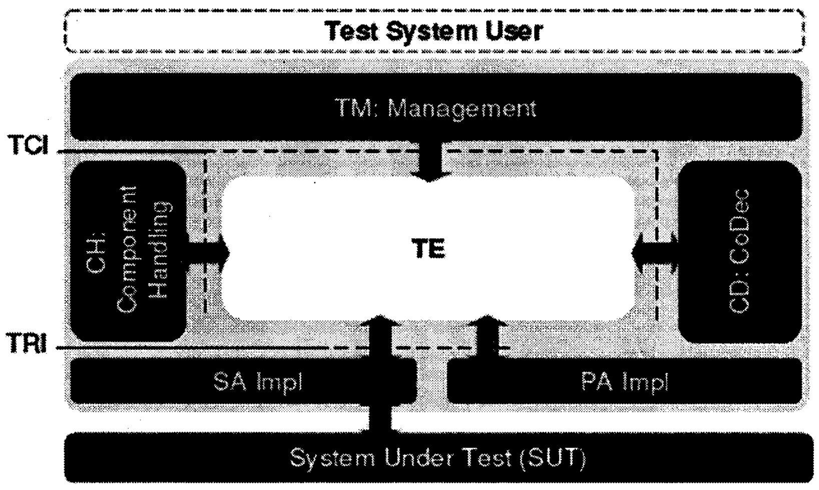


Figure 2-4: General structure of TTCN-3 test system

The rest of this chapter will describe these entities and give an idea about their roles in the test system. It will also explain the interaction among them and define their corresponding interfaces as it is illustrated in the previous figure and summarized in table 2-5 according to their TTCN-3 standardizations parts.

Test Component	Entity Identification	TTCN-3 part
TE	TTCN-3 Executable	ETSI ES 201 873-1 TTCN-3 Core Language (CL)
SA	System Adapter	ETSI ES 201 873-5 TTCN-3 Runtime Interface (TRI)
PA	Platform Adapter	
CD	Codec	ETSI ES 201 873-6 TTCN-3 Control Interfaces (TCI)
TM	Test Management	
CH	Component Handling	
TL	Test Logging	

Table 2-5: Overview of test system correlation

2.4.1 TTCN-3 Executable (TE)

The TTCN-3 Executable (TE) illustrated in figure 2-4 is the part of the test system in charge of the interpretation and execution of TTCN-3 modules. Figure 2-5, which is a refinement to figure 2-4, illustrates the different TE structural elements [12][13]. As explained earlier in this chapter, these elements are control, behaviour, components, types, values, ports, timers, and queues. The structural elements within the TE represent the functionalities that are defined either in a TTCN-3 module or in the TTCN-3 standard (ES 201 873-1).

For example, the structural element "Control" represents the control part in a TTCN-3 module and is specified within the module. On the other hand, "Queue" is another structural element not defined in the module [13]. It is a requirement of a TTCN-3 Executable from the TTCN-3 specification that each port of a test component has its own port queue and that is why its functionality is defined in the specification instead of the module. As a result, the TE entity in a test system implementation corresponds to the executable code produced either by a TTCN-3 compiler or a TTCN-3 interpreter.

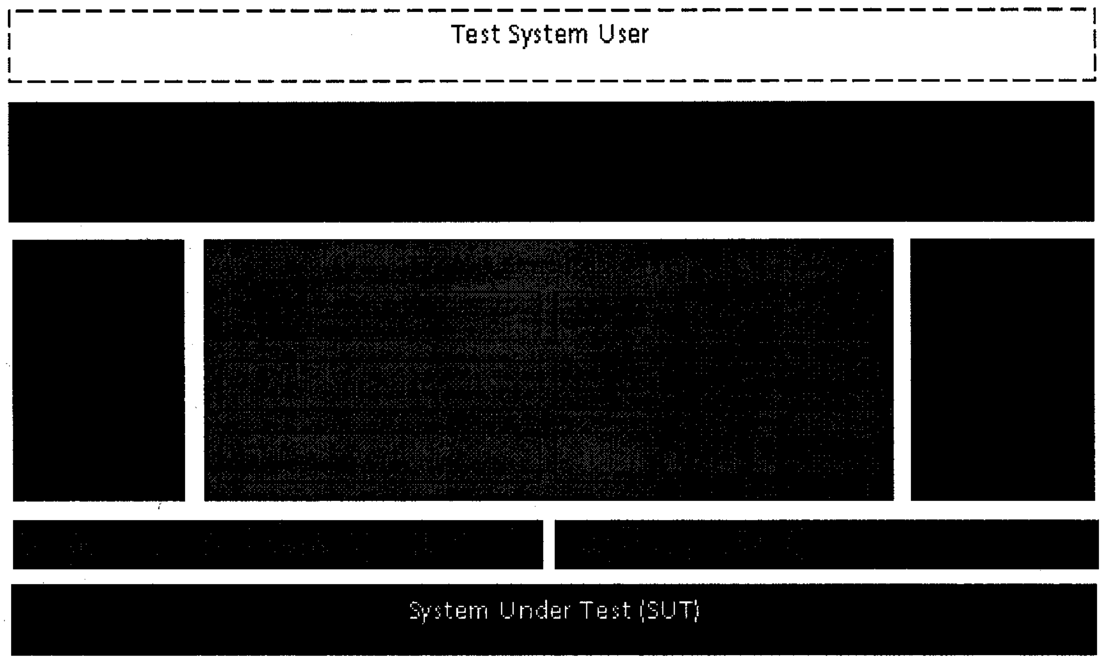


Figure 2-5: Refinement of TTCN-3 test system structure

Conceptually, the TE can be decomposed into six conceptual interacting units: a Control, Behaviour, Component, Type, Value, and Queue entity. These units deal with the execution or interpretation of test cases, the sequencing and matching of test events, as defined in the corresponding TTCN-3 modules. It interacts with the runtime entities to send/receive/log test events during test case execution, to create/remove TTCN-3 test components, and to handle external function calls/action operations/ timers [20].

2.4.2 TTCN-3 Control Interface (TCI)

The TTCN-3 Control Interface (TCI) is the sixth part of TTCN-3 standardization (ETSI ES 201 873-6 V3.1.1 (2005-06) [19]). Its objective is to provide adaptation for test management, distribute execution of test components among different test devices, encode and decode test data, and log information about test execution [20]. This interface is defined as a collection of operations independent of any target using the Interface Definition Language (IDL). Then, concrete language mappings to Java or ANSI- C, for example, are defined in the TCI standard specification (ETSI ES 201 873-6).

There are two kinds of TCI operations "required" and "provided", which are defined from user's point of view [19]. The operations that a TTCN-3 Executable should called required operations. It means the user "requires" from a TTCN-3 Executable a specific functionality to provide to the TTCN-3 test system and participate in its construction. To satisfy such requirements, the TE must tell the user about events where the user should "provide" an option. Thus, the functionality that should be provided by the test system to the TTCN-3 Executable is named provided operation.

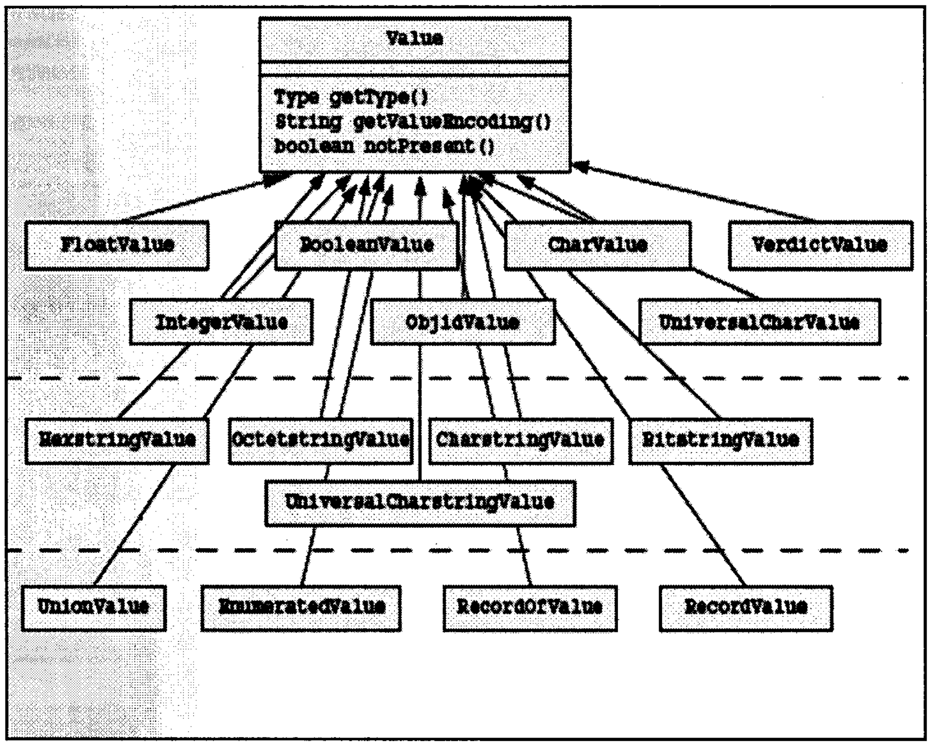


Figure 2-6: Abstract values hierarchy

Besides operations, test data play an important role during test execution. The TCI specification defines a set of abstract data types (ADTs) used to signify data communicated between the TE and the other entities using the TCI interface. With these ADTs, the TCI can describe at a high level the kind of data that should be passed from a calling to a called entity. For these abstract data types, a set of operations is defined to process the data by the coder/decoder. Each operation returns either a TTCN-3 value of these ADTs or a TTCN-3 basic type like integer or string. Then, the concrete representation of these ADTs and the definition of basic data types like string are defined in the chosen language mappings (e.g. Java). Figure 2-6 shows the hierarchy of the abstract data values. Within a TTCN-3 test system, the TCI defines the interaction between TE entity from one side and the CH, the TM, CD, and TL entities from another side. These TCI entities will be explained in the next subsections.

2.4.2.1 Test Management (TM)

TM is the entity in charge of the overall management of a test system and the implementation of a test system user interface including error handling.

It interacts with the TE as follows. After the test system has been initialized, test execution begins with the TM entity. The TM calls TTCN-3 modules, transmits module parameters and external constants to the TE, keeps track of the test case execution, and supplies test event logging. The differentiation between functionality related to test execution control and those related to test event logging can be done in the TM entity.

On the other hand, the TE provides the entry points to the test cases, the start/stop of a test case and a control part. The Test Management Interface has two sub-interfaces: *TciTM Required* and *TciTM provided*, as it is depicted in the below [22].

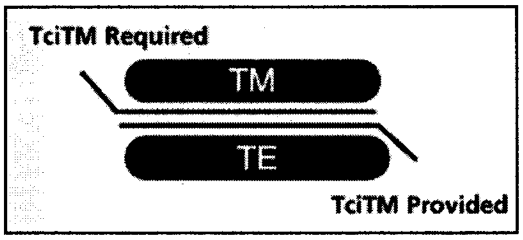


Figure 2-7: TciTM sub-interfaces

The TCI-TM Required Interface: TCI-TM defines the interface of the test management to the TTCN-3 runtime behaviour. The test management needs to call TTCN-3 module, start execution of the control part and test cases, as well as to interrupt test execution. The TE from its part offers the entry point for test case execution and some elementary database functionality.

The TCI-TM Provided Interface: TCI-TM provided defines the user interface of the TTCN-3 runtime behaviour to the test management. Objects that implement this interface have to provide the TTCN-3 runtime behaviour with the module parameters and will be called by the runtime behaviour upon the termination of test execution for a single TTCN-3 test case or a TTCN-3 module.

2.4.2.2 Coding and Decoding (CD)

This entity is in charge of encoding TTCN-3 values according to the encoding attribute into bitstrings to be sent to and understood by the System Under Test [19]. Additionally, it is responsible for decoding bitstrings according to decoding hypothesis into TTCN-3 values to be sent to the TE. The TE decides which codecs should be used. It passes the TTCN-3 data to the suitable encoder to be encoded. Also, the CD entity decodes the received data using the appropriate decoder to get TTCN-3 values. The TCI Codec Interface describes its "required" and "provided" operations through two sub-interfaces: *TCI-CD required* interface and *TCI-CD provided* interface shown in figure 2-8 [22].

The TCI-CD Required Interface: It specifies the operations the CD requires from the TE. All these operations are also required at the TCI-TM and TCI-CH interfaces. The TciValue interface defines operations on type server objects of the TTCN-3 Runtime Interface. These objects represent either types that are defined in a TTCN-3 module or basic TTCN-3 types. Each object returns a Type upon a request. To be able to achieve decoding, the CD requires particular functionality from the TE indicated by operations.

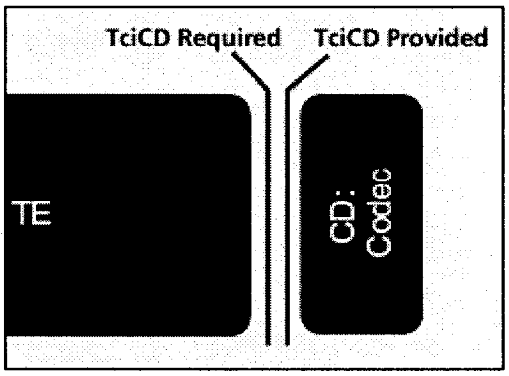


Figure 2-8: TciCD sub-interfaces

The TCI-CD Provided Interface: When the TTCN-3 runtime behaviour makes an encoding/decoding request, this interface should respond to the request. To do that, it is implemented as objects that provide the TTCN-3 runtime behaviour with access to particular encoders/decoders. Normally this will be the test adapter object.

The codec will perform two actions encoding and decoding. In the first action, the codec takes the communication data that are described in a high-level in the TE entity and passes it to the encoder. By using a set of functions and accessing this data through the Value interface, the encoder converts the internal TTCN-3 data representation to bitstring to make it real world data. This encoding process is independent from the SUT. While in the second action, the decoding is done through decoding hypothesis. Specifically, the decoder translates bitstring accessed by Type to TTCN-3 data representation. TE may query multiple times for decoding the same bitstring. Accordingly, the decoder tries to decode the provided bitstring using the suitable decoding rules into a value of given type to allow matching with TTCN-3 template definitions. In case the decoding successes, it returns the value; otherwise, a null value is returned.

2.4.2.3 Components Handling (CH)

This interface consists of operations needed to implement the management of the communication between TTCN-3 test components in a centralized or distributed test system. It includes operations to create, start/stop test components, set up connection between TTCN-3 components, manage test components and their verdicts, and handle message/procedure based communication between TTCN-3 components.

The TE can be distributed among several test devices across one or many physical systems. It also synchronizes test system entities that could be distributed onto several test devices as depicted in figure 2-9 [19]. In this case, there will be multiple TEs. One of them is special because it is responsible for starting a test case and calculating its final verdict.

Each node within a test system includes the TE, SA, PA, CD and TL entities.

Nevertheless, the test system has only one TM and one CH entity to provide test management and test component handling respectively between the TEs on each node. All the operations specified in CH interface are distributed into two sub-interfaces: *TciCH Required Interface* and *TciCH Provided Interface* as depicted in figure 2-10 [22].

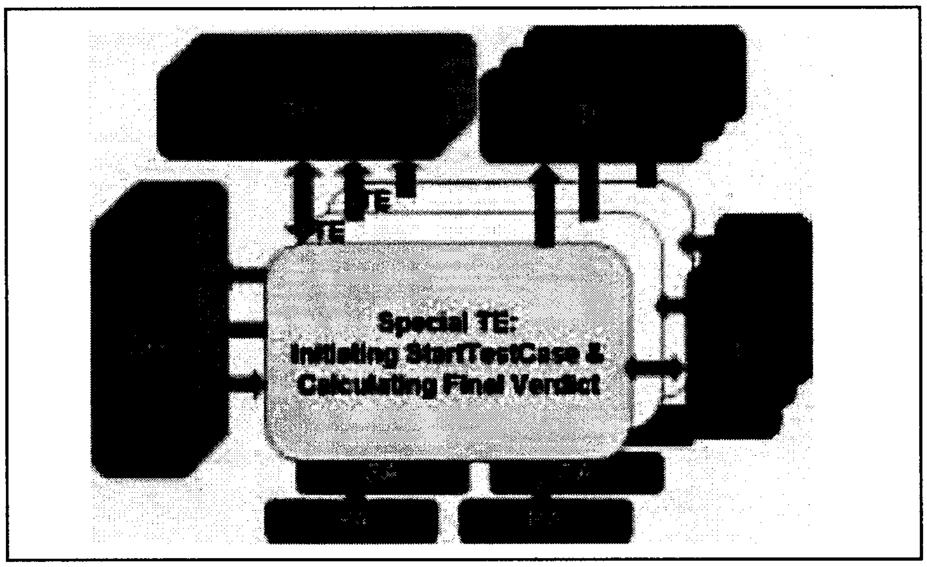


Figure 2-9: General structure of distributed test system

TciCH Required Interface: This interface specifies the operations the CH requires from the TE. Besides, it requires all operation of the TCI-CD interface whose kind is *required.* Furthermore, the CH provides operation to start parallel test components and handle procedure based communication between test components.

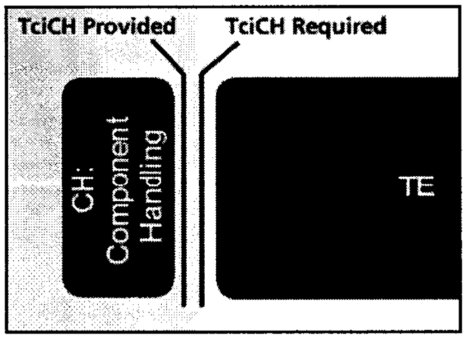


Figure 2-10: TciCH sub-interfaces

TciCH Provided Interface: In this interface, test component handling occurs. The interface specifies the operations should be provided to the TE by the CH entity. With these operations, it implements the communication of the TTCN-3 components among each other. It includes operations to establish connections between test components, send messages to a test component, and enqueue messages received from a test component.

Communication between TTCN-3 components is achieved through message based, procedure based, or mixture of both. Thus, it is the job of the CH to adapt message and procedure based communication of TTCN-3 components to the specific execution platform of the test system. CH is aware of connections between TTCN-3 test component communication ports. It transmits send request operations from one TTCN-3 component to another. The receiving component may reside in a different instance of the same TE located on a different node. Then, it places the received test events in the port queues of the TE in order to inform the latter of their presence.

The CH is also aware of the Procedure based communication operations between TTCN-3 components. It is able to distinguish between the different kinds of procedure-based communication operations (e.g. call, reply, and exception), and has to spread them to the target component located in TE using a suitable mode [19]. Nevertheless, the CH is not responsible of the implementation of the behaviour for the different components implemented in the TE; instead, it implements their communication.

2.4.2.4 Test Logging (TL)

The TL entity (interface) represents the test system to the user, performs test event logging, and provides information about the test execution as shown in Figure 2-11 [19].

For all TTCN-3 level operations, the logging provides an operation to log the respective event being performed by the TE, SA, PA, CH, or CD to the user. For instance, it logs the components that have been created, started and terminated; data has been sent to the SUT, received from the SUT and matched/mismatched to TTCN-3 templates; timers have been started, stopped, or timed out [19]. In addition, the TL entity controls the level of detail of this information.

The TCI Test Logging Interface (TCI-TL) describes the operations a TE is required to implement and the operations should be provided by test logging implementation to the TE.

The TL entity has a unidirectional interface where any entity belongs to the TE may send a logging request to the TL entity [20].

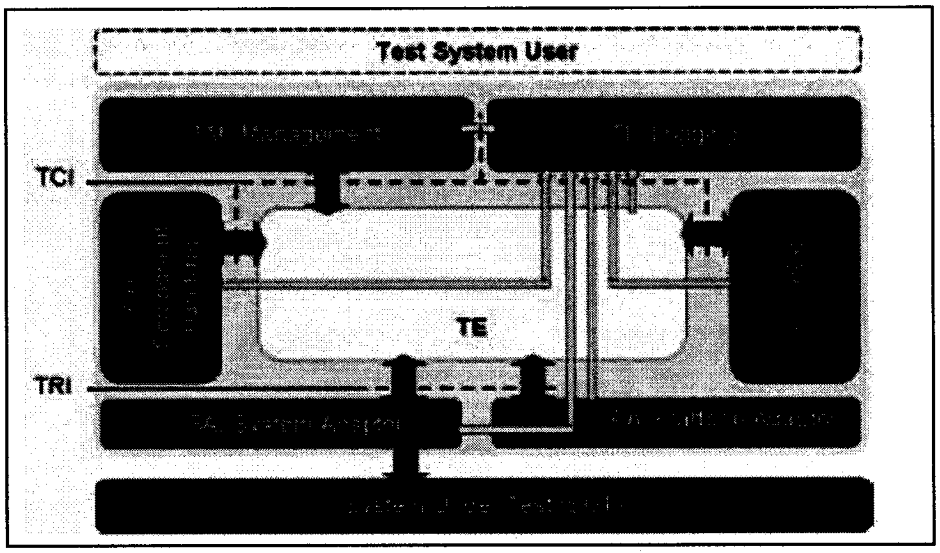


Figure 2-11: Detailed View of Test Logging

TCI-TL Provided: It specifies the operations the TL should provide to the TE. A hundred operations are implemented.

2.4.3 TTCN-3 Runtime Interface (TRI)

The TTCN-3 Runtime Interface (TRI) is the 5th part of TTCN-3 standardization that defines two sorts of adaptation. One is the adaptation for communication of a test system to a particular processing system under test. The other is the adaptation for timing and external functions to a particular processing platform. It provides techniques for the TE to send test data to the SUT or manipulate timers, and similarly to notify the TE of received test data as well as timeouts.

Similar to what we have seen in TCI interface, the TRI interface is defined as a collection of operations independent of any target language. Vendors can support language mappings for the TRI abstract specification to possible target languages such as ANSI C. These operations are implemented as part of one entity and called by other entities of the test system. For each operation, the interface specification defines the related data structures, anticipated effect on the test system, and any constraints for using the operation.

In the TRI operations, only encoded test data should be passed. Instead of defining an explicit data interface for TTCN-3, the TRI standard defines a group of abstract data types (ADTs) to indicate which information is to be passed from the calling to the called entity, and vice versa. The ADTS are used in the definition of TRI operations for connection (e.g. TriComponentIdType), timer (e.g. TriTimerIdType), and communication (e.g. TriMessageType) purposes. The concrete representation of these ADTs as well as the definition of basic data types are defined in the respective language mappings in Part 5 of TTCN-3 standardization.

The TRI Interface defines the interaction between the TTCN-3 Executable (TE), SUT Adapter (SA), and Platform Adapter (PA) entities contained by a TTCN-3 test system implementation. By its two entities SA and PA, the TRI can achieve the adaptation as we will see with more details in the following sub sections.

2.4.3.1 SUT Adaptor (SA)

The SA is the implementation of the System under Test Adaptor as defined in ES 201 873-5 [20]. It adapts the TTCN-3 message and procedure based communication operations with the SUT to the particular execution platform of the test system based on an abstract test system interface.

The SA implements a real test system interface with the TE called triCommunication. The mapping of the TTCN-3 test component communication ports to test system interface ports is visible to the SA. Thus, this interface is used to exchange encoded test data between the two SA and TE entities. The job of the triCommunication operations is to initialize the Test System Interface, set up connections with the SUT, deal with message and procedure based communication to the SUT, and to reset the SUT Adapter. These operations are divided into two categories between two sub interfaces: TriCommunicationSA Interface and TriCommunicationTE Interface as it is depicted in Figure 2-12 [22].

TriCommunicationSA Interface: Its operations are classified into categories. For example, it has a rest, communication handling, Message and procedure based communication operations.

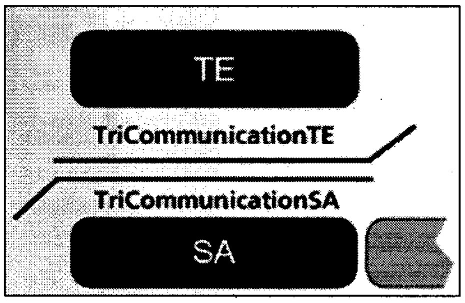


Figure 2-12: TriCommunictionSA interface

TriCommunicationTE Interface: It is also defined through a set of operations to add a message or a call to a queue.

2.4.3.2 Platform Adaptor (PA)

The PA is the implementation of the Platform Adaptor as defined in ES 201 873-5 [20]. In addition, it is an entity that adapts the TTCN-3 Executable to a particular execution.

The PA implements external functions and handle timer operations. The timer instances are created in the TE; however, the PA provides a TTCN-3 test system with a sole concept of time. A timer in the PA can only be recognized by its Timer IDentification (TID). This means that the PA treats both explicit and implicit timers in the same manner.

Figure 2-13 shows the PA interface with the TE, which is called TriPlatform [21].

It includes all operations and tools required to adapt the TTCN-3 Executable to a specific execution platform. The TriPlatform interface has operations to enables the call of external functions. It also has operations to implement and control timers using Timer IDs[20]. The timer operations enable starting, reading, stopping, and the inquiring the status of timers. When a timer times out, TriPlatform will add it to the timeout list and the PA informs the TE about it. The PA reports the status back and the TE indicates errors.

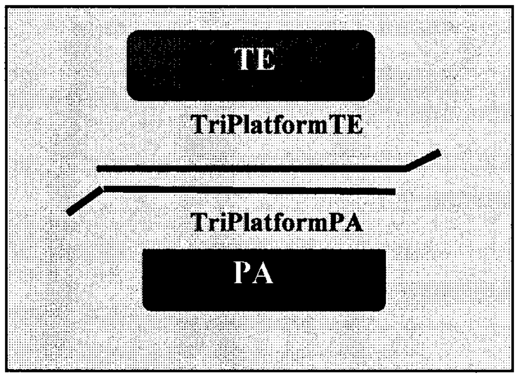


Figure 2-13: TriPlatform interface

TriPlatformPAInterface: It consists of a set of TriPlatformPA operations classified [21] under three categories: Platform reset, timer, and external functions operations.

TriPlatformTE Interface: The following is a TriPlatformTE operation [21]:

- void *triTimeout* (...);

2.5 Summary

This chapter introduced TTCN-3 and discussed the structure, and features of the core language. The new features provide timer handling, matching mechanisms, test verdicts, distributed test components, encoding information. With these features, the structure of the language allows building solid abstract test specifications. After that, the chapter explained the structure of a complete test system. It also showed the interactions among its entities though sub interfaces that specify what is provided by the corresponding entity and what is required from the interacted entity. These interactions are defined in terms of operations as a part of one entity and called by other test system entities. The operations definitions use a set of abstract data types and values of the types. In the next chapter, we will explain our methodology and start building our own test system.

CHAPTER 3
Our Methodology for Executing Timed Test Cases using TTCN-3

This chapter will talk about a methodology that we have developed to execute a previously given file containing a collection of timed test cases. The methodology is based on using the TTCN-3 language and all its related entities explained in the prior chapter to execute the test cases in that file. Section 3.1 will provide an overview of the methodology and the test system. After that, section 3.2 will give an idea about Timed Input Output Automata approach that was used to generate the test cases in that file based on the test purpose concept and the specification. The file generation is not part of our work, but we will show this approach only to give the reader an idea about the process that led to generate the test cases of that file without any implementation details. Then, section 3.3 is the starting point of our work in this thesis where we will explain by details the methodology that we have created. The methodology converts the test cases in that given file into an abstract test suite coded in the TTCN-3 testing language as a first step of the execution of the test. Finally, in section 3.4, we will explain how we configured our ATS, created a TTCN-3 project, and compile the ATS to get its executable.

3.1 Overview of the Methodology and Test System

As mentioned in the previous chapter, a TTCN-3 test system is conceptually composed of a set of interacting entities. Each entity corresponds to a particular aspect of functionality in a test system implementation as previously illustrated in figure 2-5.

In this section, we will give the reader an overview of our methodology through the development process of a TTCN-3 test system as it is illustrated in Figure 3-1 [34]. During the process, we will focus on our developed test system. In general, the process consists of six steps.

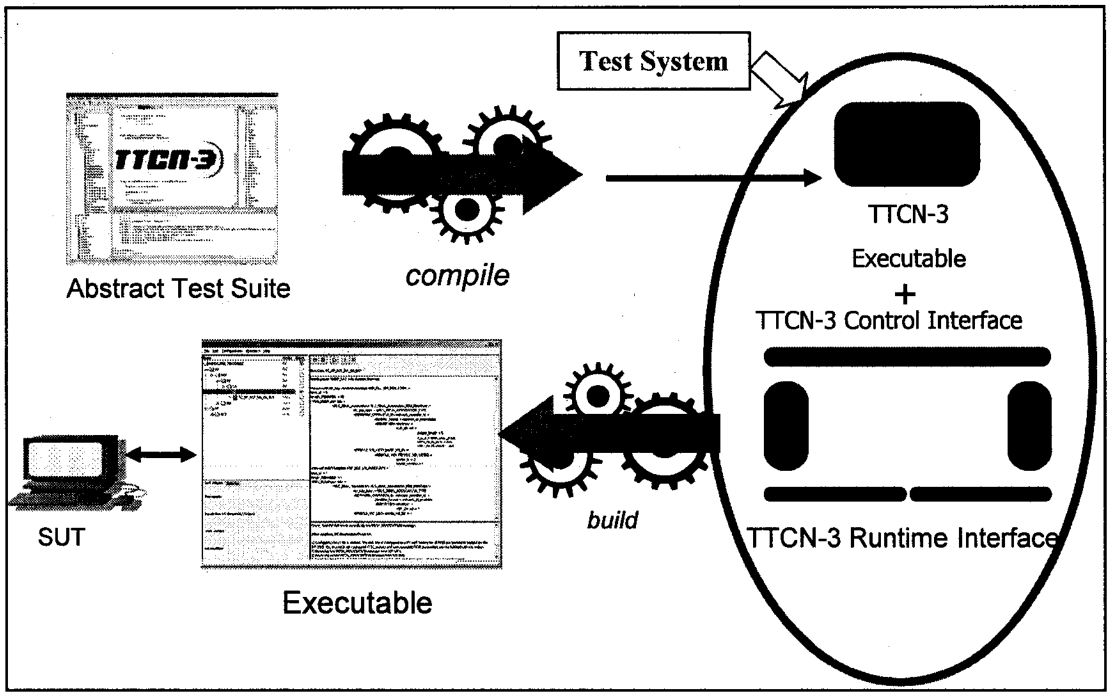


Figure 3-1: Overview of the Process of TTCN-3 Test System

Step one: It is about the development of a TTCN-3 Abstract Test Suite for the timed test cases. This can be done directly under a TTCN-3 environment provided by any TTCN-3 tools. However, in this case, test developers should have good knowledge of TTCN-3 testing language and be trained for it. To save time, effort, and training cost, we developed our methodology in this thesis through a program that automatically converted the timed test suite into TTCN-3 Abstract Test Suite (ATS). The methodology is also intended to use that program whenever we need to automatically generate TTCN-3 ATS for similar applications. Then, we compile the generated ATS to get the TTCN-3 Executable (TE) as a result of the compilation, which is depicted in the upper part of the Figure.

However, the TE cannot be run as it is because it is still in an abstraction level. To move from the abstraction to concrete level, we need to support the TE by implementing other parts to make from it a complete test system. The TE is the core entity of the test system and should interact with the other conceptual entities through the TTCN-3 Control Interface (TCI) and TTCN-3 Runtime Interface (TRI). As it is shown in the right hand-side of the figure, the TE with the TCI and TRI together form a complete test system that can easily be run.

Step two: This step is related to the TTCN-3 TCI interface. As we have explained in chapter 3, the TCI consists of four entities. The TM entity is responsible of the test management and the user test interface, which allows the interaction between the tester and the test system during run time. The CH entity is in charge of components handling while the TL entity supports logging the test events. The TM, CH, TL entities have ready-to-use implementation that comes with the compiler and tools supported by vendors. The fourth entity in the TCI interface is the CD. This entity needs to be implemented to adapt the data of the messages exchanged between the TE and SUT.

Step three: This step is related to the TTCN-3 TRI interface. The TRI has two entities: SA and PA. The PA is the platform adapter, which has also ready-to-use implementation. However, the SA is in charge of the system adaptation. It takes the encoded messages and communicates them to the SUT. It also takes the decoded responses and communicates them to the TE. This means the SA adapter depends on our configuration used in the abstract test, and that is why the SA should be implemented.

Step four: Once we have the TE, the codec (CD entity of the TCI interface), and the SUT adapter (SA entity of the TRI interface) ready, it means our test system is completed. This is indicated by the oval shown in the figure above. This test system should be applied on the SUT. However, because we are doing a black-box testing, there is no real time system or implementation under test. Thus, we have to stimulate a SUT in this step.

Step five: We build the programs mentioned above together using simple XML file and we get the executable test. Then, we apply the final executable on the SUT.

Step six: This step is about loading and running the test system to execute the test suite like it is shown in the lower part of the figure as an action. We need a module loader file to load the test campaign. Then, we run the test that will automatically launch the execution of the test cases.

In the following, we are going to explain and implement each of these steps, but before that we will talk about the TIOA approach for generating a file of timed test cases. This is just to show that we will start our methodology from a timed test case file generated by a solid approach. However, the implementation and details of the generation are out of the scope of this thesis. Our methodology is the continuing work that comes after the generation process with the purpose of executing the generated test cases and analysing the final results of the test.

3.2 Timed Test Suite Generation Using TIOA and MSC

As stated in chapter 1, the timed test suite for a real-time multimedia application has been generated based on the specification and test purpose. The approach deals with Timed Test Case Generation using Timed Input Output Automata (TIOA) and Message Sequence Charts (MSCs). The TIOA is used to describe the specification while MSC is used to describe the test purpose of the user. MSC is a graphical specification language standardised by the ITU-T as Recommendation Z.120 [35]. A test purpose is a precise representation of the functionality to be tested. The user is interested in testing only the most critical functions of the system or the most frequently executed parts of the system. The objective of using a test purpose is to help reduce the number of generated test cases since an exhaustive testing of a TIOA causes the well-known state explosion problem [36]. MSC-2000 provides a graphical representation that helps the user to clearly specify what to test. With this scenario language, communication behaviour between system entities and their environment can be specified [37]. It is used because it has tools that are efficient to express timing behaviour for real-time systems such as timers and the time constraints between any pair of events.

Figure 3-2 [38] depicts a test purpose in MSC for the multimedia system for which we will implement and run the TTCN-3 test system later in this chapter and the next one.

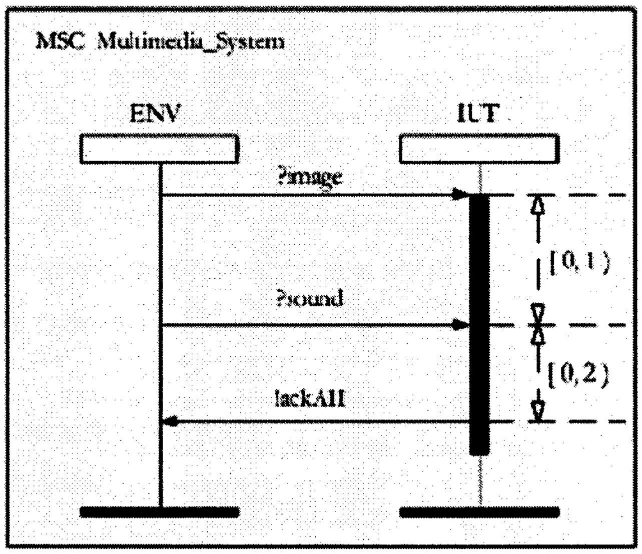


Figure 3-2: Test Purpose of Multimedia System

On the other hand, a TIOA is a tuple *$(I_A, O_A, L_A, l^0_A, C_A, T_A)$ [39], where:*

– *I_A is a finite set of input actions. Each input action begins with "?".*
– *O_A is a finite set of output actions. Each output action begins with "!".*
– *L_A is a finite set of location. The term "location" is chosen instead of the term "state" because the latter is used to define the operational semantics of the TIOA.*
– *l^0_A □ L_A is the initial location.*
– *C_A is a finite set of synchronous clocks set to zero in l^0_A assuming the time is dens, which means that the clocks values are real numbers.*
– *T_A is the set of transitions. Each transition consists of a source location, an input or an output action, a clock guard that should hold in order to execute the transition, a set of clocks to be reset when the transition is executed, and a destination location.*

Figure 3-3 [38] is an example of a TIOA describing the behaviour of a simple multimedia system. The system receives an image and its sound within two time-units, sends an acknowledgment in less than five time-units after the reception of the image, and then sends the message reset and starts waiting for another image. If the time constraints are not satisfied, the system issues the message error and goes back to its initial state. The TIOA that describes the system has four locations l0 (the initial location), l1, l2, and l3, six transitions and two clocks x and y. The transition from l0 to l1, denoted by $l0 \xrightarrow{?image, x<=2, x:=0, y:=0} l1$, is executed when the system receives the message *image* and the value of clock x is less than or equal to 2. When the transition is fired, the clocks x and y are set to 0 [38].

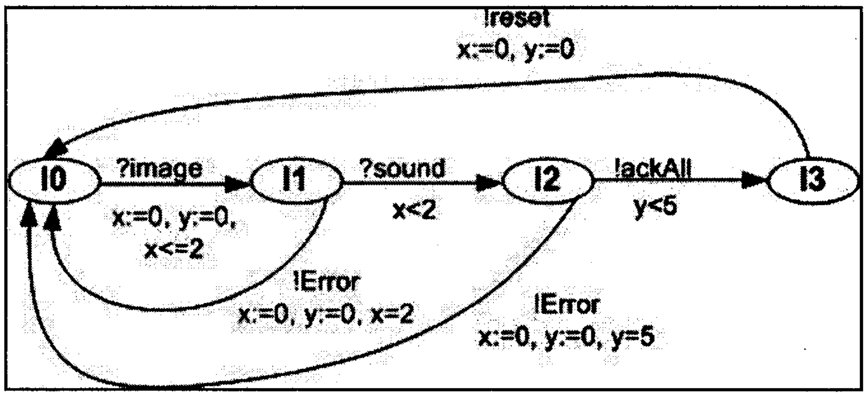


Figure 3-3: Specification of Multimedia System

This approach of efficient generation of test cases for real-time systems consists of four major phases.

Phase one: The MSCs of the test purposes should be converted to the TIOA module used for the specification as follows: Each message received by the IUT in MSC is translated to an input action in TIOA, and each message sent by the IUT is translated to an output action. The state between each pair of exchanged messages is indicated as a location in TIOA.

Phase two: a synchronous product is constructed from the TIOAs of the specification and that of the test purpose. This is done by creating the initial location of the TIOA through the concatenation of the initial location of the specification with that of the test purpose. Then, the transitions of the synchronous product are incrementally built and the remaining states are added to the set of states. Figure3-4 illustrates the synchronous product of the test purpose and specification TIOAs for the same multimedia system [38].

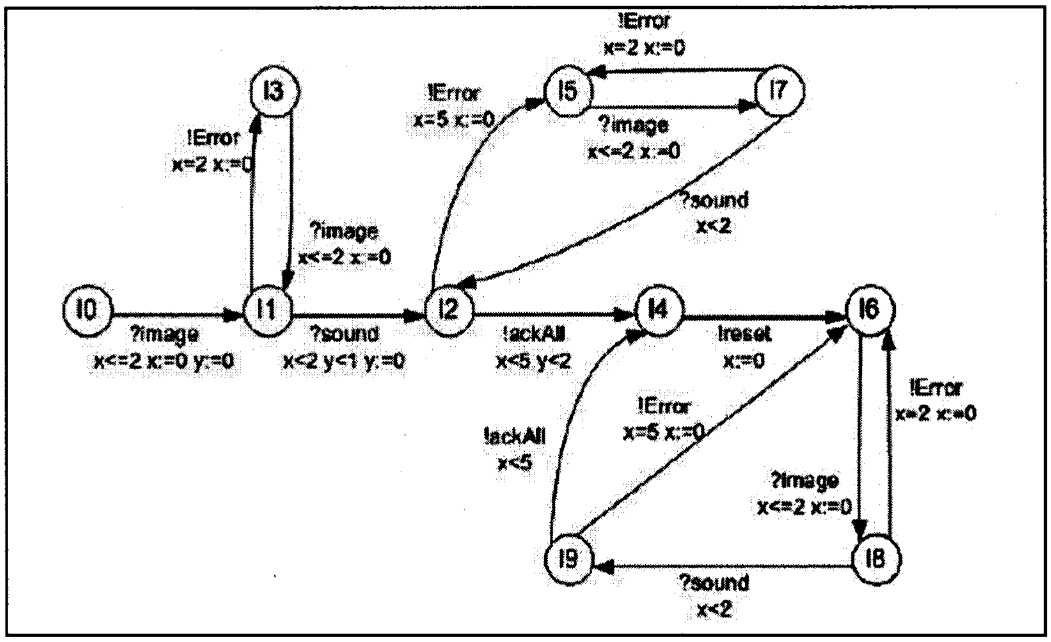
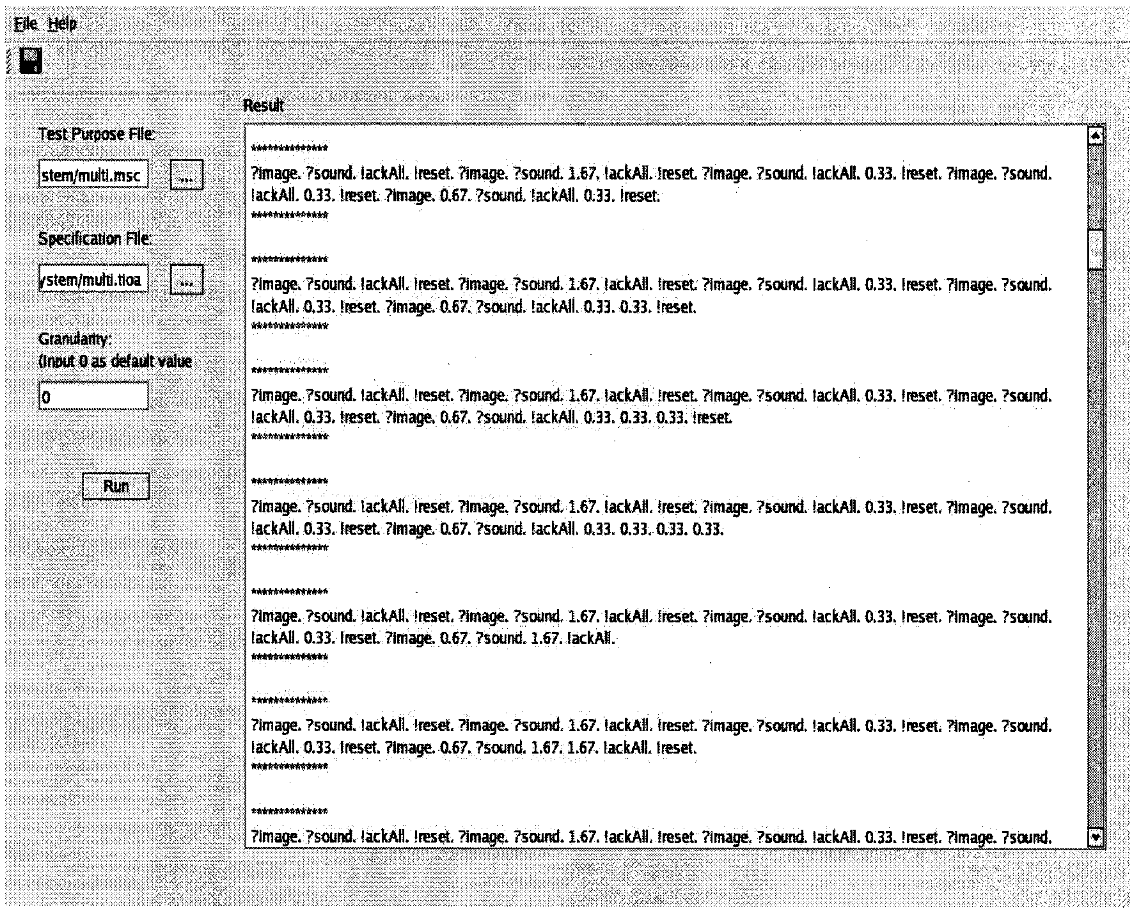


Figure 3-4: Grid Automaton of Specification and Test Purposes

Phase three: The infinity of delay transitions makes the number of states of the TIOA infinite. So, test cases should be generated from a subset of the TIOA called Grid Automaton (GA) [36] [38]. In this GA, the regions graph of the synchronous product is sampled by choosing a set of representatives for each state and accordingly instantiating the delay transition in the definition of region graph.

Phase four: The Grid Automata is traversed to extract the test cases for the system. So, each traversal gives rise to a test case that starts at the initial state of the grid automata and finishes when a leaf is reached.

At last, the result of the generation of timed test cases using the above approach is written into a text file. Each of the test cases consists of input actions, time delays, and output actions. Figure3-5 gives a sample of this file. The test cases in this file need to be executed and run using a proficient test method or language. That is why we have developed our methodology in this thesis using the TTCN-3 language. With the methodology, we will start from that text file to develop a program that automatically converts the test cases into a TTCN-3 suite, and then we completed the test system to automatically and efficiently execute the test suite.



Test Purpose File:
stem/multi.msc

Specification File:
ystem/multi.tioa

Granularity:
(Input 0 as default value
0

```
**************
?image. ?sound. !ackAll. !reset. ?image. ?sound. 1.67. !ackAll. !reset. ?image. ?sound. !ackAll. 0.33. !reset. ?image. ?sound.
!ackAll. 0.33. !reset. ?image. 0.67. ?sound. !ackAll. 0.33. !reset.
**************


**************
?image. ?sound. !ackAll. !reset. ?image. ?sound. 1.67. !ackAll. !reset. ?image. ?sound. !ackAll. 0.33. !reset. ?image. ?sound.
!ackAll. 0.33. !reset. ?image. 0.67. ?sound. !ackAll. 0.33. 0.33. !reset.
**************


**************
?image. ?sound. !ackAll. !reset. ?image. ?sound. 1.67. !ackAll. !reset. ?image. ?sound. !ackAll. 0.33. !reset. ?image. ?sound.
!ackAll. 0.33. !reset. ?image. 0.67. ?sound. !ackAll. 0.33. 0.33. 0.33. !reset.
**************


**************
?image. ?sound. !ackAll. !reset. ?image. ?sound. 1.67. !ackAll. !reset. ?image. ?sound. !ackAll. 0.33. !reset. ?image. ?sound.
!ackAll. 0.33. !reset. ?image. 0.67. ?sound. !ackAll. 0.33. 0.33. 0.33. 0.33.
**************


**************
?image. ?sound. !ackAll. !reset. ?image. ?sound. 1.67. !ackAll. !reset. ?image. ?sound. !ackAll. 0.33. !reset. ?image. ?sound.
!ackAll. 0.33. !reset. ?image. 0.67. ?sound. 1.67. !ackAll.
**************


**************
?image. ?sound. !ackAll. !reset. ?image. ?sound. 1.67. !ackAll. !reset. ?image. ?sound. !ackAll. 0.33. !reset. ?image. ?sound.
!ackAll. 0.33. !reset. ?image. 0.67. ?sound. 1.67. 1.67. !ackAll. !reset.
**************


**************
?image. ?sound. !ackAll. !reset. ?image. ?sound. 1.67. !ackAll. !reset. ?image. ?sound. !ackAll. 0.33. !reset. ?image. ?sound.
```

Figure 3-5: File of Timed Test Cases

3.3 Test Cases Conversion to TTCN-3 ATS

Our methodology consists of translating the test cases into a test suite using TTCN-3 testing language. Our methodology starts with the development of a C++ program called "ATS_Generator.cpp" to read the timed test cases from the input text file obtained from applying the TIOA approach explained in the previous section. Our program is called ATS_Generator because its purpose is to generate an Abstract Test Suite (ATS) written in TTCN-3 from textual test cases shown in the previous figure.

This program can be used by other applications similar to our multimedia one in order to automatically convert their textual test suite into an ATS without the need for having an expertise or any deep TTCN-3 knowledge. The tester of other applications can also use this program or easily modify it to their needs with just some knowledge of TTCN-3 language.

But, in order to make the program reusable by other similar applications, our solution should be as general as possible. Thus, we created another input text file where we intended to make the number and the type of actions (send/receive) along with their timers' values different from a test case to another. The following figure illustrates a small sample file reflecting more general test cases and will be used as our input to the program.

```
?image. 0.33. !ack.
?image. 0.67. !ack. 0.33. !reset.
?image. 0.33. ?sound. 0.67. !ackAll.
?image. 0.33. ?sound. 0.33. !ackAll.
?image. 1.33. ?sound. 0.33. !ackAll. 0.67. !reset.
?image. 0.33. ?sound. 0.67. !ackAll. 0.33. !reset.
?image. 0.67. ?sound. 0.67. ?data. 1.00. !ackAll. 0.33. !reset.
```

Figure 3-6: File of General Timed Test cases

Then, we converted the test cases of this file into a TTCN3 code written to an output file for the used "ATS_Generator.cpp" program that we have built as we will explain in this section. The code of this program is given in Appendix A, while some fragments concerning the functionality of the program will be illustrated in this section to support the explanation.

However, to write such program that produces a TTCN-3 ATS in an output file, the programmer should first know the functionality and structure of a TTCN-3 test suite using the core notation as it has been illustrated in chapter 2. The generated ATS stimulates the System Under Test (SUT) or Implementation Under Test (IUT) by sending particular test data and receiving other expected data as responses. Because TTCN-3 will be used for testing the specification, the system that has to be tested can be considered as a black box. From the black box only the defined interface is known where more details about the stimulations and the expected reactions are provided in the given input file (specification and test purposes). The objective of the ATS is to test if the system works conforming to that given information.

The sending and receiving of test data in the test suite are provided by the component Main Test Component (MTC). The MTC is in charge of the principal workflow of the test suite and the management of the test execution. Each test suite creates only one MTC during test suite execution. Parallel Test Components (PTCs) are usually used when a system should be tested by using multiple distributed components, which is not our case. Thus, in the module to be built there exists only one MTC besides the system component. If the system component test case was successful, the verdict will be set to *pass*; otherwise, it will be set to *fail* or to *inconc* if it is not defined.

The *ATS_Generator.cpp* program implements and calls the following set of functions:

void *declareTTCNmodule* (ofstream&);
void *readInputFile* (ifstream&, ofstream& , string [], int&, int&, int&, int&, int&);
void *TTCNtestCases* (ofstream& , string [], int& , int& , int& , int& , int&);
void *getOutput* (ofstream& , string [], int , int, int);
void *executeTestCases* (ofstream& , int);

The first function, *declareTTCNmodule*, defines the module name besides the types and data that will be used for the inputs and outputs. Then, it identifies the ports type followed by the MTC and System components types. It also writes the defined information to an output file which will later represent our TTCN-3 module.

The second function, *readInputFile*, will loop to read the timed test cases from the input file "input.txt" given in figure 3-6 one by one (line by line). For each iteration, the *readInputFile* calls the function, *TTCNtestCases*, to convert the entries of each test case that has been already read to a TTCN-3 test case. The *TTCNtestCases* itself calls the *getOutput* function to assist in the conversion process. The TTCN-3 code generated for each test case will be immediately written to the output file "output.txt" in order to continue building our module. When the end of "input.txt" file is finally reached, the *readInputFile* will call the *executeTestCases* function to define and write into the output file a module control part where the test cases will be called one by one for execution.

At this stage, the module or the ATS is ready and is passed to the text editor "CLEditor", which is used as a TTCN3 parser and analyser. The "CLEditor" provides comprehensive syntax checking according to the grammar of the TTCN-3 Core Language. After correcting all detected errors in the module if any, the module will be the guide in developing our program "ATS_Generator.cpp" and its expected output file as shown in appendix A.

The next section will demonstrate and explain the output file which is the TTCN-3 ATS (or module), then execute it using suitable TTCN-3 software.

3.4 TTCN3 Executable (TE) Generation

3.4.1 TTCN-3 Project Creation

With the "ttworkbench" software, we have created a TTCN3 Project under the "TTCN-3 development" environment. A TTCN-3 project has also a Java project nature, because this will be needed for developing other related functionalities. The project has two different sets of properties that we have set, one is for the TTCN-3 codes and the other is for Java codes.

After the project is created, the required TTCN3 module(s) must be created under the TTCN3 folder or added to it in case we have ready ones. We have created an empty module file called "MyModule.ttcn" and copied to it the contents of the output file "output.txt" generated using the "ATS_Generator.cpp" program. Appendix B shows the module file “MyModule.ttcn”.

3.4.2 Overview of the TTCN-3 Module

As it is mentioned in chapter 2, the highest unit in the TTCN-3 core notation is a module, which is similar to a class in JAVA. The module includes all source code fragments related to the whole test suite. Principally, all TTCN-3 test suites are composed of a definition part and a control part. The definition part includes all necessary definitions like data types, ports, test components, and test cases. The control part is usually optional and can be used to control the order of starting the test cases and define the conditions on starting one or more test cases. As described above, the test suite tests the sending and receiving of a several data packets.

3.4.2.1 Test Data

The test data will be exchanged in terms of input packets and output packets. To define a packet structure, the efficient data type record is constructed. An input packet consists of three fields. The first field represents the type of the packet (e.g. image, sound, etc…). The second field contains the length of the whole data packet and the last field carries the payload as follows:

```
type record inputPacket {
     octetstring pktType length(2),
     octetstring pktSize length(2),
     octetstring pktData
}
```

Similarly, the output packet has two fields in its structure, a length and data. Since the transmission of data packets is byte basis, each field of the record is of type *octetstring*. TTCN-3 allows specifying the maximum size of the strings holding the packets types and length.

After defining the types *InputPacket* and *OutputPacket,* variables of these two types should be defined and filled with values. For this purpose, templates of type *InputPacket* are defined to handle diverse inputs like image and sound. Then, each field of *packet* will be filled with octetstring values, representing the type, the length, and the payload. In addition, templates of type OutputPacket are defined and given values to handle diverse outputs (e.g. ack, reset). For example, below is the template for the inputPacket called *image:*

```
template inputPacket image := {
          pktType := '0003'O,
          pktSize:=  '0003'O,
          pktData := '01001010'O
     }
```

3.4.2.2 Communication Ports

For the communication between the components and the SUT, types of ports have to be defined. This is usually done using the keywords *type port*, followed by a user-chosen name (here *myPort*), and the type of communication. There are three types of communication: message, procedure, or mixed. For each type, we should configure the ports through which the data will be transmitted for incoming, outgoing, or incoming-outgoing communication. In this application, the ports have to provide a message based communication, which is indicated by the keyword *message.* Since the inputs packets are different from those related to the output packets, the port *myport* is configured to pass two separate types of messages as it is shown in the code below. These types are called *in* and *out* to indicate incoming or outgoing messages respectively.

```
     type port myPort message {
          out inputPacket;
          in outputPacket;
     }
```

3.4.2.3 Test Components

After we define the ports, the test components must be defined. The type definition decides which ports and timers a component possess. The first component to define as a type is the system component with the port *systemPort* declared in its body as follows:

The MTC is defined as a type and given the name *mtcType.* In the body of its definition, the type of ports which are used in the MTC can be declared as *mtcPort.*

Additionally, the MTC can get local timers to be able to act depending on time. The timers are initialized to the values given in the input file of test cases. However, the values of timers are different from a test case to another and we don't want to create a module for each test case. Thus, to handle this, there are two choices for defining and initializing the timers. Either by making a copy of the MTC for each test case with the related values of timers local to that MTC copy or by defining the timers related to each test case at the beginning of the test case. We have used the first manner in the creation of our ATS because it shows all the details about the timers and keeps their variable names as they are when the test runs. Here is the MTC copy for the first test case:

```
type component mtcType1 {
     port myPort mtcPort;
     timer timer1 := 0.33;
}
```

3.4.2.4 Test behaviour

The test behaviour should be specified through test cases definitions. Each test case has to be given a name (testCase1, testCase2...), and to specify the components it will be run on (MTC in our case), and the system (SUT) with which the MTC will interact. So far, we have two components with their defined ports, timers, and packets that should be sent. To determine where to send/receive messages, all the ports of the MTC have to be mapped to the ports of the system. All behavioural operations (e.g. sending, receiving) take place in the test case body which actually represents the MTC behaviour.

The communication with the system always occurs under the support of a test adapter [13]. The MTC sends a particular packet to the SUT using the *send* operation of the *mtcPort*. The message which will be sent is passed as parameter in the *send* operation. The related local timer begins to count down by using the *start* operation. After sending the packet, three possible events can occur. To handle several events or branching, TTCN-3 provides the behavioural statement *alt* that is similar to the compound (if-else) statement. Satisfying a given condition will decide which branch of code in the body of the *alt* statement must be executed. According to it, a local verdict for the test case will be set inside the *alt* statement. The following is part of the test behaviour for a simple test case:

```
mtcPort.send(image);
timer1.start;

alt{
     [] mtcPort.receive(ackAll)
     {
          timer1.stop;
          setverdict(pass);
     }

     [] mtcPort.receive
     {
          timer1.stop;
          setverdict(fail);
     }

     [] timer1.timeout
     {
          setverdict(fail);
     }
}
```

If the *mtcPort* receives the expected packet during a precise duration, the already started local timer will be stopped and the local verdict is set to *pass*. Or if the *mtcPort* receives something else such as a fragment of the packet, the local timer will also be stopped and the verdict will be set to *fail*. But, if a timeout occurred and nothing received, the local verdict is set to *fail*. A final verdict for a test case will be calculated during run-time test according to overriding rules. Based on it, the test case will be terminated with a verdict *pass* or *fail* to indicate it was successful or not.

The control part in this module has simply the functionality of starting several typical test cases and saving their resulted verdicts in different variables such as:

```
control {

     var verdicttype testcase1Result := execute(TestCase1());

     var verdicttype testcase2Result := execute(TestCase2());

    :
    :
```

3.4.3 Producing the TE

Then, the module should be compiled using the "TTthree" compiler included in the same used software. The compiler reads module definitions written in the TTCN-3 core language, which are highlighted in the example below and compiles them into Java sources [22].

```
module MyModule {

        // type declarations
        // configuration declarations
        // module parameters and
        // external function declarations
        // template declarations
        // function declarations
        // test case declarations
        // control part
```

In case errors have been found, the build operation will be aborted. Otherwise, a successful compilation will build byte code class files from the compiled Java sources and combined the classes into one JAR archive file [22]. The resulted Java archive file will be representing the TTCN-3 Executable (TE) and be placed by default in the same directory as the respective TTCN-3 module file. This TE is the essential part of our objective for creating a test system in order to execute the given timed test cases.

At this point, we finished the first step of our development process shown at the beginning of this chapter. However, the TE can not execute the test cases because it is still in the high level of abstraction as we have described in chapter 2. Thus, the TE needs to interact with other interfaces to adapt the TTCN-3 Executable to a real system and to achieve concrete representation to the abstraction suite. These interactions and adaptations will be the rest of the development of our methodology as we will explain in details in the following chapter.

3.5 Summary

In this chapter, we focused on our methodology of developing a test system to execute a given timed test suite using TTCN-3. We first gave an overview of this methodology and the steps of the development process. Then, we briefly showed the process of deriving timed test suite using MSC-2000 and TIOA, which in not part of our contributions; instead, its resulted file is considered our starting point. Then, we transformed the timed test cases to a TTCN-3 abstract module using a C++ program as well as our knowledge of the TTCN-3 core language. After, we compile the ATS to get the TTCN-3 executable file. The executable is the core part of a TTCN-3 test system and the first step of building it. In order to run the TE on a target test device against a real system under test, some supporting functionalities have to be implemented to complete the test system. In the next chapter, we will implement these functionalities, run the test system, and demonstrate the result.

CHAPTER 4

Test System Implementation and Execution

In this chapter, we will complement the first step of our development done in the preceding chapter in order to cover the development of the other steps in the process. We will use object oriented design and programming to develop the other parts of the test system that complement the job of the TE core entity generated in chapter 3. The programs will mostly be coded in Java under Eclipse environment and run as a part of the TTCN-3 project that we have already created in chapter 3.

In section 4.1, we will discuss the design of the packages and classes and draw their UML diagrams. In section 4.2, we will provide an overview of the implementation of our test system. After that, we will implement three programs: codec, SUT adapter, and SUT in section 4.3, section 4.4, and section 4.5 respectively. Section 4.6 explains how to get the executable archived files resulted from the compilation and interpretation of our created programs using a build file. Then, in section 4.7, we will prepare the test system for running. Thus, we will write a small Module Loader File to load the test campaign, and we finally run the implemented test system and show the results.

4.1 Class Diagrams

To show our implementation structure clearly, the classes of the implementation of the tools and interfaces of the test system are organized into three major packages: Codec (which belongs to TCI interface), Tools, and TRI as illustrated in Figure 1-2. The circles are provided by UML and used here to indicate interfaces. Each package or interface contains one or more classes. In the next sub sections, we will explain them along with their associated classes. Appendix C demonstrates, as a javadoc, the packages and class hierarchy for all java programs used in this project.

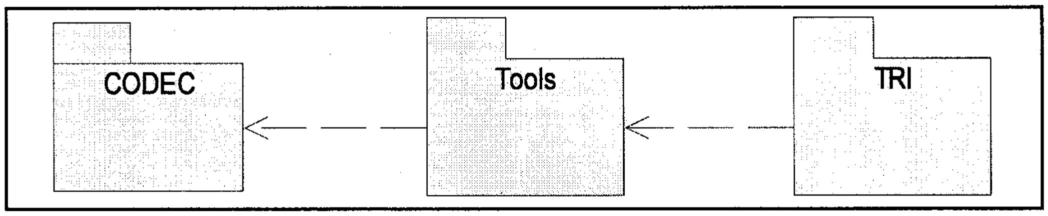


Figure 4-1: Package Diagram

As one can see from the class diagram in the following figure, we haven't focused on the operations and attributes of the base classes (e.g. AbstractBaseCodec and TestAdapter) and sub interfaces (e.g. TriCommunicationSA and TciCDProvider) for two reasons: First, because they have been already implemented by the vendors with the software used for our test system implementation. Second, they contain big sets of operations and attributes that can not fit in a diagram. However, we have included them for an overall comprehension since our designed classes depend on them.

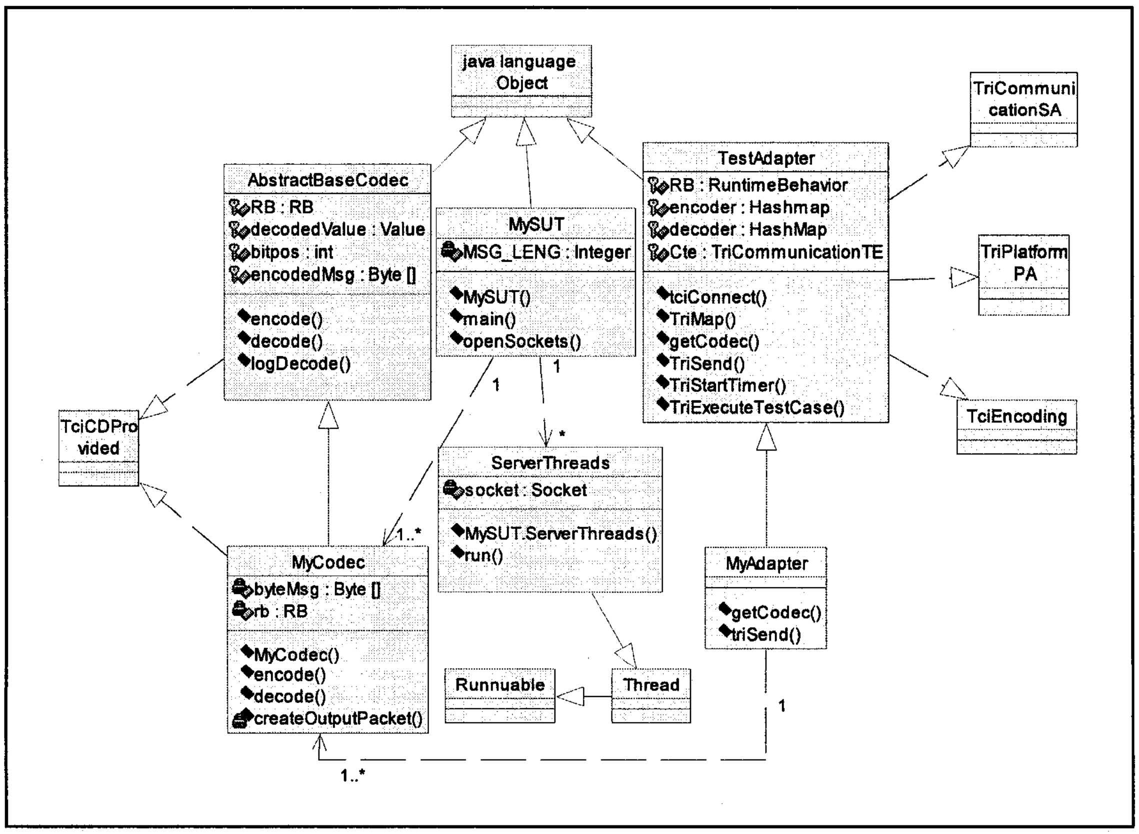


Figure 4-2: Overall Class Diagram

4.1.1 Codec Package

This is a sub-package of the TCI package (interface). Codec class diagram is shown in figure 4-3. It contains two major classes: AbstractBaseCodec and MyCodec.

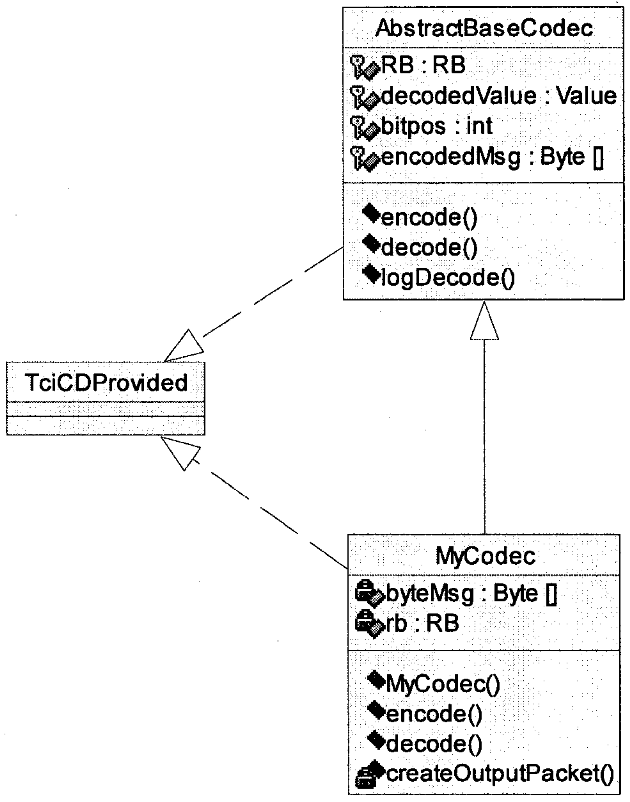


Figure 4-3: Class Diagram of Codec

AbstractBaseCodec Class: This is a basic class whose implementation is provided by vendors. It contains a big set of operation. From this set we have chosen encode and decode operations to show on the class diagram. The AbstractBaseCodec Class uses these two operations when implementing the interface TciCDProvided.

MyCodec Class: This class is inherited from AbstractBAseCodec class to reimplement the interface TciCDProvided. It has the means to convert the outgoing messages defined in the ATS into bitsting in order to enable the SUT to understand them.

It also converts the bitsting sent from the SUT to the TE into TTCN-3 values. Consequently, the codec job is to implement both encode and decode operations within the interface TciCDProvided. It first creates a new object of MyCodec Type. Then, it defines the encode operation that encodes a sent message based on the basic encoding rules specified in the AbstractBaseCodec and returns it as a message of type triMessage.

In addition, it defines the decode method that decode a received message according to the decoding hypothesis and return it as a TTCN-3 value. This class defines the auxiliary method createOutputPacket that is called by the decode method to put the received bytes into an output packet form and return that packet.

4.1.2 TRI package

In this package, we have the super class TestAdapter, which is a ready-to-use class. Three following sub interfaces are depending on it: *TriCommunicationSA*, *TriPlatformPA*, and *TciEncoding.* We have also the sub class MyAdapter as illustrated in following figure.

Test Adapter class: The basic TestAdapter class implementation is provided by the vendor of the software used for our test. Any other test adapter created for a particular application should be inherited from the TestAdapter class. This class provides the fundamental TRI functionality and the communication among test components. To achieve its job, it has three interfaces (classes) that extend the java Serializable interface and called TriCommunicationSA, TriPlatformPA, and TciEncoding.

TriCommunicationSA Interface: As its name indicates, implements the communication of the TTCN-3 TE to the SUT Adapter (meaning to the SA) through a set of operations. It contains operations to initialize the Test System Interface, set up connections to the SUT, handle message and procedure based communication with the SUT, and reset the SUT Adapter.

TriPlatformPA Interface: It includes all operations necessary to adapt the TTCN-3 Executable to a particular execution platform. The interface provides operation to start/ stop/read a timer, find its status, and to add timeout events to the expired timer list. Moreover, it offers operations to call TTCN-3 external functions and to reset the Platform Adapter.

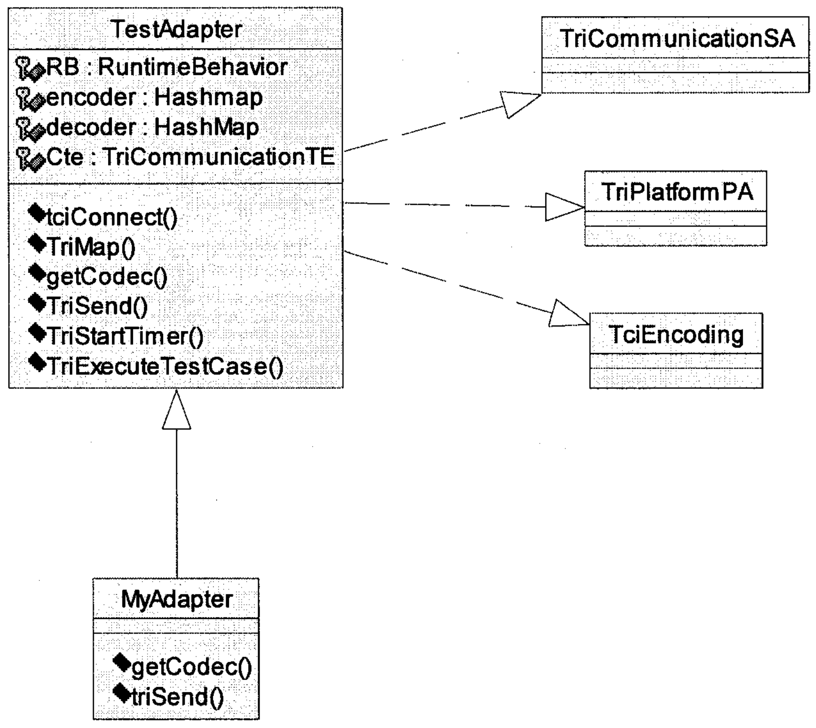


Figure 4-4: Class Diagram of TRI

TciEncoding Interface: It implements the objects for the TestAdapter to give the TTCN-3 Runtime behaviour access to particular encoders/decoders in order to react on encoding/decoding requests by the TTCN-3 Runtime Behavior.

MyAdapter Class: This adapter is inherited from the TestAdapter class. Its job is to create a new Adapter object to send messages via a given communication port to the SUT and to enqueue the messages received from the SUT. It implements the getCodec operation that verifies which codec is used in the Test System and returns the BaseCodec of type TciCDProvided. MyAdapter class also implements the triSend method, which is called by the TE whenever it executes a TTCN-3 send operation on a component port mapped to the TSI port. This result of the triSend operation is of TriStatus type whose value is either TRI_OK or TRI_ERROR depending on the success or fail of the operation.

4.1.3 Tools Package

This package contains what belongs to the SUT as follows.

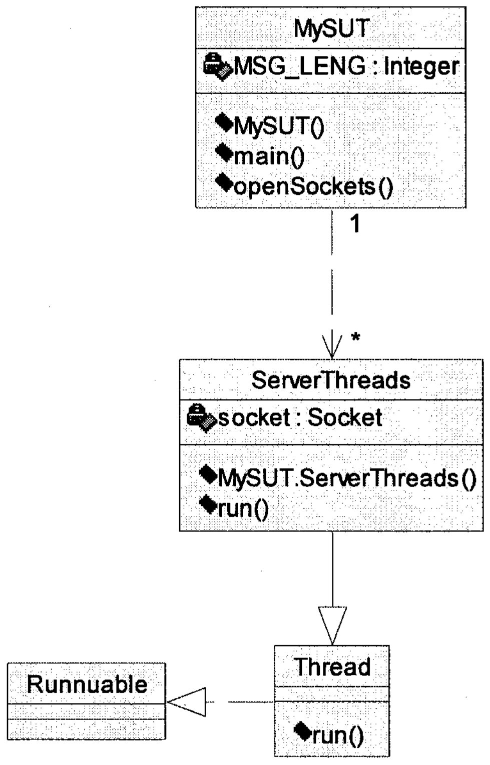


Figure 4-5: Class Diagram of Tools

MySUT Class: This class imitates the job of a system under test that must react to stimuli coming from the test system. Its objective is to react to stimuli coming from the test system. The TCP protocol will be used in this class as a communication protocol. The constructor method MySUT creates a new object. Then, the main method uses the new object to call the openSockets operation for communication purpose. The openSockets operation itself opens a sender and a receiver socket, and begins listening on the receiver socket. This class has also a nested one called serverThreads.

ServerThreads Nested Class: It is the child of the java language *Thread* class in order to overrides the implementation of the run method in the java language *Runnable* interface. Besides the run operation, the nested class has a constructor named *MySUT.ServerThreads.*

4.2 Overview of the Test Implementation

As mentioned in the previous chapters, a TTCN-3 test system is conceptually composed of a set of interacting entities. Each entity corresponds to a particular aspect of functionality in a test system implementation as illustrated previously in chapter 2. For example, the TE entity executes or interprets the compiled TTCN-3 code, TM manages the test execution, SA realizes proper communication with the SUT, and PA handles timing and implements external functions if any. The TM, CD, and CH entities belong to TTCN-3 Control Interface (TCI), while SA and PA belong to the TTCN-3 Runtime Interface (TRI).

With regard to these entities, Figure 4-6 [13] sketches out the test implementation process, the files involved, and their interrelations. It also resumes the process of the test development shown in Figure 3-1 of the last chapter. The TE corresponds to the executable code produced by the TTCN-3 compiler. The other part of the TTCN-3 test system (SA and CD) is represented by the TCI and the TRI interfaces. It includes the other entities to deal with other aspects that can not be concluded from only the information presented in the TTCN-3 module.

The TCI interface with its entities has ready-to-use implementation for test system user interface, test execution control and test event logging. What is left to implement in the TCI interface is just an appropriate Codec to meet our application needs. Similarly, the TRI interface with its entities has also ready-to-use implementation for its functionalities given and supported by vendors. Some of these implementations can be included as they are. For the others, either they have an empty implementation or should be overwritten by new implementation to match our own test system needs. For example, the PA entity implements external functions and operations to handle timers, so we don't need to implement a PA. However, we do need a SUT Adapter (SA) to communicate with our own simulated SUT.

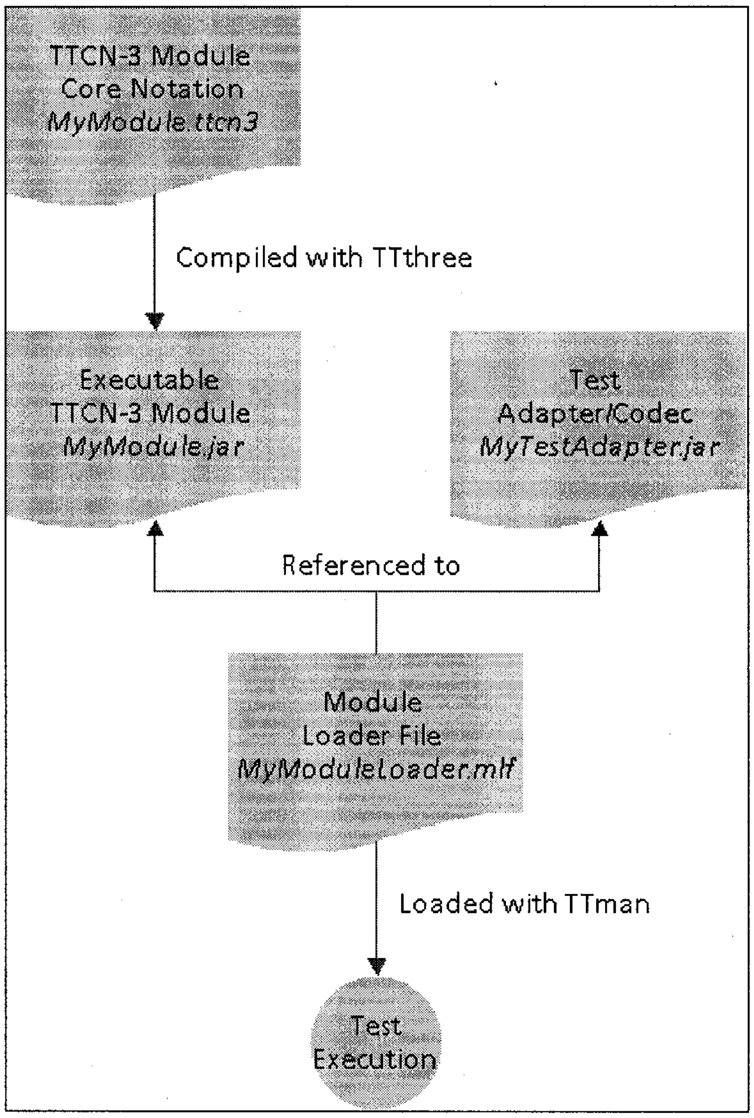


Figure 4-6: Test implementation process

In figure 4-6, the part of the test system which deals with interpretation and execution of TTCN-3 modules represents the TTCN-3 Executable (TE) is shown in the left-hand side of the diagram. It is produced as a result of the compilation to the TTCN-3 ATS using the compiler TTthree. For the part on the right-hand side of the figure, we will see in the next sections a Codec and a SUT Adapter programs, which are written in Java because the software used provides a TTCN-3 to Java mapping. The simulated SUT is not shown in this figure because it is not considered part of the test system, but it is needed to apply the test system on it. Building the developed program is not shown in this diagram. Then, a module loader file is created to load the test execution with the TTman loader tool provided with the used software.

4.3 Codecs

This section represents the second step of the development process of our TTCN-3 test system as shown in chapter 3. The codec is part of the TCI interface implementation. The test user interface, test management, logging event information can be taken by default from the interface implementation as they are. However, the codec has to be implemented.

The job of the codec is to ensure that the TTCN-3 data types of messages sent from the abstract test suite were understood in the system and vice versa [13]. The system understands the data in a bitstring format; hence, all sent data types from the test suite have to be mapped to a bitstring. In the other direction, the outgoing messages from the system should be translated from bitstring format to an appropriate TTCN-3 data type. The TTCN-3 data type that will be sent to the system is defined as a record of *InputPacket* type where its values are given in the template. The values of this type have to be encoded into a bitstring before passing it to the system. Regarding the other direction, the bitstring has to be decoded into an *OutputPacket* when a response message is sent to the TE.

The codec program, MyCodec, needs to include the basic codec class AbstractBaseCodec from the library, besides the classes that are needed to define the TCI data types mentioned in chapter 2. These classes are Type, Value, RecordValue, TciCDProvided, TciCDRequired, TciTypeClass. Additionally, it needs to include the file defining the TriMessage from the TRI interface.

The program myCodec is inherited from the AbstractBaseCodec to implement the interface TciCDProvided. This interface ensures having the two methods encode and decode. The first method "*encode*" gets a variable template of type Value, which is a TTCN-3 value. Then, the variable template will be encoded to a byte array and stored in a variable of type TriMessage. The encoding is performed by the super class AbstractBaseCodec.

On the other hand, the method "*decode*" takes two parameters, the received message of type TriMessage and the decoding Hypothesis of type Type. The received message which has to be decoded is stored in a variable. If this variable is empty, the method will return null because nothing found to be decoded. But, if is not empty, we have to check which kind of TciTypeClass the decodingHypothesis is. In case the TciTypeClass of the decodingHypothesis is a record, it should be verified if the decodingHypothesis supports messages from the type of the send message. The system has to send packets of type OutputPacket. If this type is found, a new variable OutputPacket from type RecordValue is built from type OutputPacket, by creating a new instance from the decodingHypothesis. The variable OutputPacket and the message to be received are then used as input parameters for the method createOutputPacket. The method is responsible for the actual translation from the byte array to the TTCN-3 type OutputPacket. If the decoding Hypothesis does not support messages from type OutputPacket, the order of decoding is given to the super class.

The method createOutputPacket gets an Output Packet from type RecordValue and the message which has to be decoded from a byte array to the Output Packet format. The fields of the Output Packet have to be filled according to the number of bytes specified in the module. To do this, another method is defined to convert bits to octetstrings.

The javadoc generated for this program is given in Appendix D.

4.4 SUT Adapter (SA)

This section represents the third step of the development process of our TTCN-3 test system as illustrated in the prior chapter. Normally, the test adapter task is to connect the abstract TTCN-3 test suite to the system. Because we are performing a black-box test, no real system has been used, but in the test adapter the system is simulated to react as required.

In our TTCN-3 project, under java sources folder, we create a system adapter called *MyAdapter* with an input and output interface. The system adapter receives particular data packets via the input interface and then sends other specific data packets via the output interface. Thus, the operations that adapt the TTCN-3 Executable to the particular execution platform should be implemented.

This adapter program should first include all required TRI types definitions mentioned in chapter 2 such as TriAddress, TriComponentId, TriMessage, TriPortId, and TriStatus. Besides, it needs the coded data from the TCI interface, so it will include BaseCodec and the TciCDProvided type from the TCI interface. The program is implemented as sub class inhereted from the generic class TestAdapter provided with the TRI package. The generic adapter class doesn't have a send operation because usually this operation depends on the requirement of the application that has to be tested. So, we should define our own triSend method to be called whenever there is a message to send. The method takes the required parameters such as the component id of the sender which is also the receiver, the id of the involved ports, the address of the SUT (optional), and the message that has to be sent. The method returns the status of the send operation which is TRI_OK in case the message has been sent successfully. Otherwise, TRI_ERROR will be returned. The return value TRI_OK does not mean that the SUT has received the sent message.

Next, this adapter should define the codec that will be used for a test suite. Since we have created our codec program, a method is needed to get the codec. In this method, we have to check first if the encoding name is specified in the TTCN-3 core because this is a possibility. If it is found, the method gets it from there and returns the Base Codec implementation. Otherwise, it sets the encoding Name to "*myCodec*", which is the program created in the previous section. The method will then create a new object of the class myCodec after making sure that the codec doesn't already exist as an object. It will also add the new object to a hash map to avoid objects redundancy. In case the encoding Name is not known, it will be logged with an appropriate error message. The javadoc for this program can be found in Appendix E.

4.5 System Under Test

At this point of development, our test system is completed and needs to be applied on the real Implementation Under Test (IUT) or System Under Test (SUT). Since we do not have a real one and we are performing a black-box testing, we will simulate one through writing a program for it in this section which covers step four of the development process. The program will play the role of a system under test. Its job is to react on the test system stimulation. It is written as a Java class using TCP protocol.

It is possible to use UDP protocol in place of TCP. The class opens server sockets using a method to open a sender and a receiver sockets. Then, it starts listening on the receiver socket. Upon receiving the events, it should send the output messages. In Appendix F, a javadoc for this program is provided.

4.6 Producing executable files for the java programs

This section represents the build action shown in the lower part of Figure 3-1 in the previous chapter and the fifth step of that process of test development. When we first created a TTCN-3 project, the system automatically made the necessary directories and some other related files such as build.xml. This file is like the make file in C/C++ languages. It contains an initialization part to set the name of the compilers to be used. It also sets the folders with their related paths such as TTthree, src, lib, ttcn3, and JVM command. Then, for the java sources, it sets the source and destination directories for the classes resulted from compilation. In addition, the default file usually leaves the rest to be added by the programmer/tester based on each individual test system implementation.

So, for our system, we added to the build file the statements to compile the test adapter and codec, and then to put their interpreted executable archived file (TA.jar) in the lib directory. Similarly, we added statements to the MySUT program to put its resulted executable archived file (SUT.jar) under the same directory. Finally, we define the clean operations to remove the created classes and jar files in order to start clean whenever we need to modify the java sources and re-run the build file.

The part of the code that is added to this file is given in Appendix G. In the following, we provide a segment code of it related to the codec and the test adapter compilation.

```
<echo>Compile MyProject Test Adapter and Codec</echo>
  <javac srcdir="${src}" destdir="${classes}">
 <classpath>
   <pathelement location="${TTthree}/lib/TTthreeRuntime.jar"/>
     <pathelement location="${TTthree}/lib/TTorg.jar"/>
 </classpath>
</javac>
```

4.7 Loading and Running the test System

In this section, we prepare the test system for running. To enable loading the test system to the project execution environment to be run, a module loader file (MLF) should be created.

4.7.1 Module Loader File (MLF)

It is possible to generate a draft for the MLF file from a template if the name of the main module has been specified in the project properties. The generated MLF template is coded using the XML language and located in the same place where the compiler has been started. The template will include the module name and its executable jar file with its path. It also passes recursively through all import statements and parameters in the compiled test suite if any [13]. Then, it describes the control part of the module (control part name, its module name, and set the verdict and status to their default values).

In the body of the control part description, it adds all available test cases as they are presented in the control part of the module. With each test case description, it mentions the name of the module to which a test case belongs, test case name, and set the selection, status and verdict parameter to default values. However, in order to use the generated MLF template for module loading with the TTman test management, we have to change it from a template to a Module Loader File (MLF) type and modify its contents to address our own test adapter and codec executable program (TA.jar).

4.7.2 Running the Test System

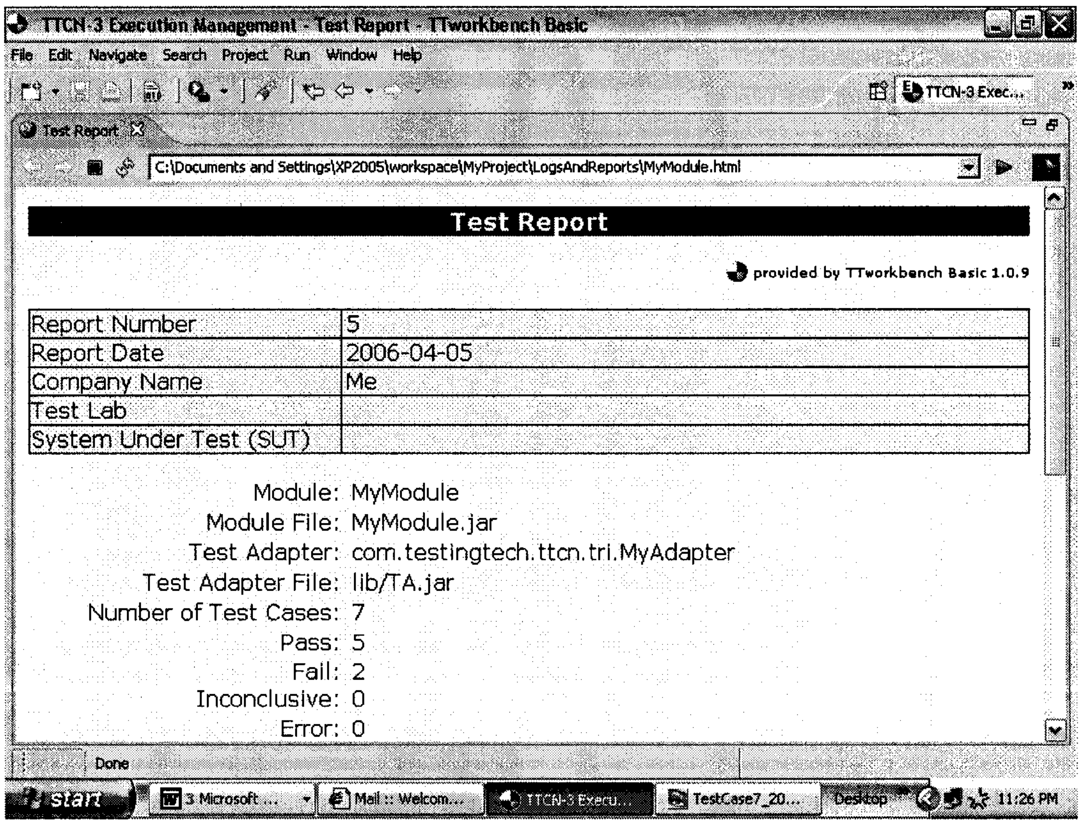
Running the Module Loader File will automatically take us from the TTCN-3 Development environment to the TTCN-3 Executable environment where the "Test Campaign Creator" window will pop up. The window lists the names of all available test cases. So, the user can determine the names and order of loading and running the selected test cases from the list according to his criterion or the importance of specific test case(s).

Then, the test campaign for our ATS "myModule" will be loaded with the selected test cases to the management view along with their attributes set to default values in the property view. By clicking on the module name, the system will automatically run the test cases one after the other while displaying all running details terminated by the result of each test case according to its final verdict.

If the ATS had a control module, then the tester could have chosen from the test campaign creator the "Use control Part" option to load the test campaign for all test cases. Then, only the name of the module is displayed on the management view. A summary about the default parameters of the module and the test cases is shown in the property view.

Pressing the run button will automatically run all test cases one by one (based on the conditions on them and the order specified in that control part of the module) while displaying the running details. The tester has the advantage of creating a report to save the test result for later reference or for more analysis. The following is the report created from running our ATS with seven test cases divided into two figures for visual clarity.

The running test system observes the reactions of IUT based on their expected values and on the time they have been received. Finally, the test results are automatically analyzed by the test system and a verdict is concluded. For any test case, if the output of each event matches the expected one while respecting the corresponding time duration, the local verdict is set to pass. A final verdict will automatically be concluded for each test case based on its local verdicts according to the overriding verdict rules. If all test cases terminated with *pass* verdicts, then the implementation is conforming to the specification and test purpose. Otherwise, there are faults in the implementation that must be sited and fixed.


TTCN-3 Execution Management - Test Report - TTworkbench Basic
File Edit Navigate Search Project Run Window Help
Test Report
C:\Documents and Settings\XP2005\workspace\MyProject\LogsAndReports\MyModule.html
Done
start

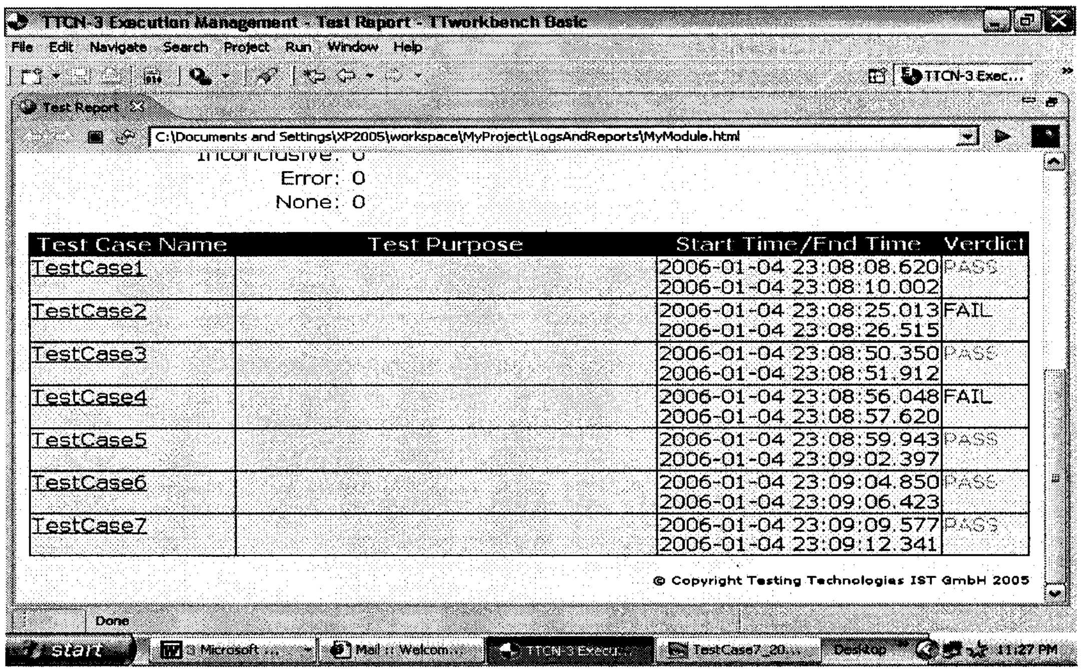
Test Report

provided by TTworkbench Basic 1.0.9

Report Number	5
Report Date	2006-04-05
Company Name	Me
Test Lab	
System Under Test (SUT)	

```
                Module: MyModule
           Module File: MyModule.jar
          Test Adapter: com.testingtech.ttcn.tri.MyAdapter
     Test Adapter File: lib/TA.jar
  Number of Test Cases: 7
                  Pass: 5
                  Fail: 2
          Inconclusive: 0
                 Error: 0
```

Figure 4-7: Test Report



C:\Documents and Settings\XP2005\workspace\MyProject\LogsAndReports\MyModule.html

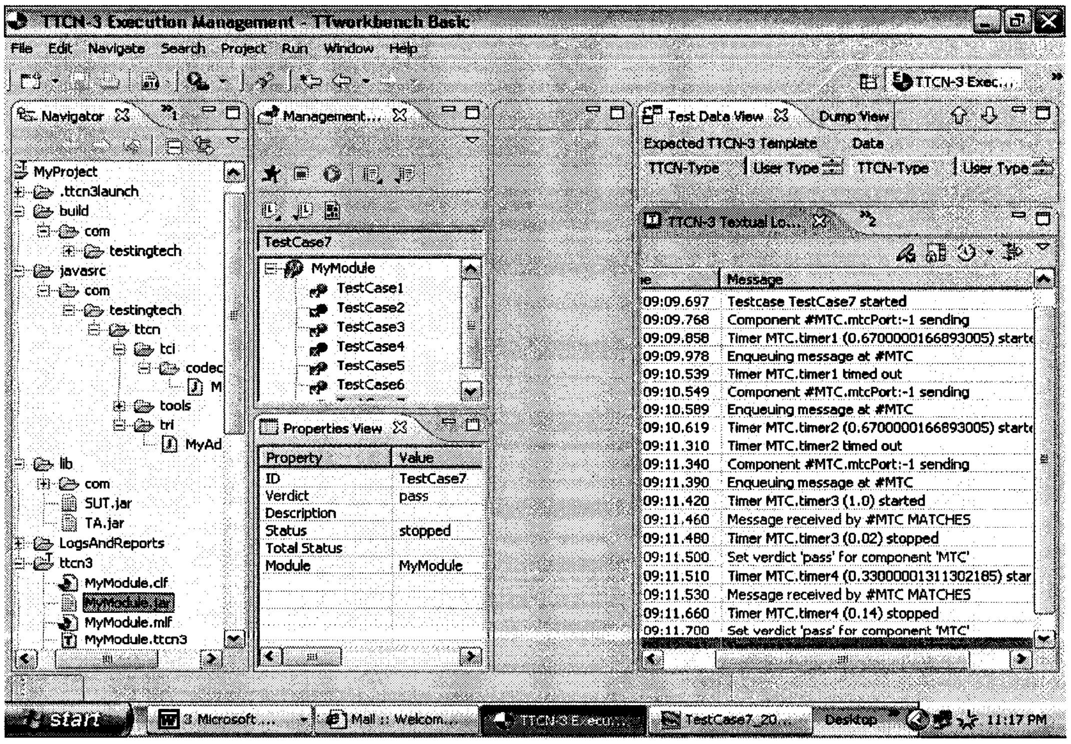
Error: 0
None: 0

Test Case Name	Test Purpose	Start Time/End Time	Verdict
TestCase1		2006-01-04 23:08:08.620 2006-01-04 23:08:10.002	PASS
TestCase2		2006-01-04 23:08:25.013 2006-01-04 23:08:26.515	FAIL
TestCase3		2006-01-04 23:08:50.350 2006-01-04 23:08:51.912	PASS
TestCase4		2006-01-04 23:08:56.048 2006-01-04 23:08:57.620	FAIL
TestCase5		2006-01-04 23:08:59.943 2006-01-04 23:09:02.397	PASS
TestCase6		2006-01-04 23:09:04.850 2006-01-04 23:09:06.423	PASS
TestCase7		2006-01-04 23:09:09.577 2006-01-04 23:09:12.341	PASS

© Copyright Testing Technologies IST GmbH 2005

Figure 4-8: Test Report (continue)

In figure 4-9, we depict the logging details resulted from running TestCase7 successfully.

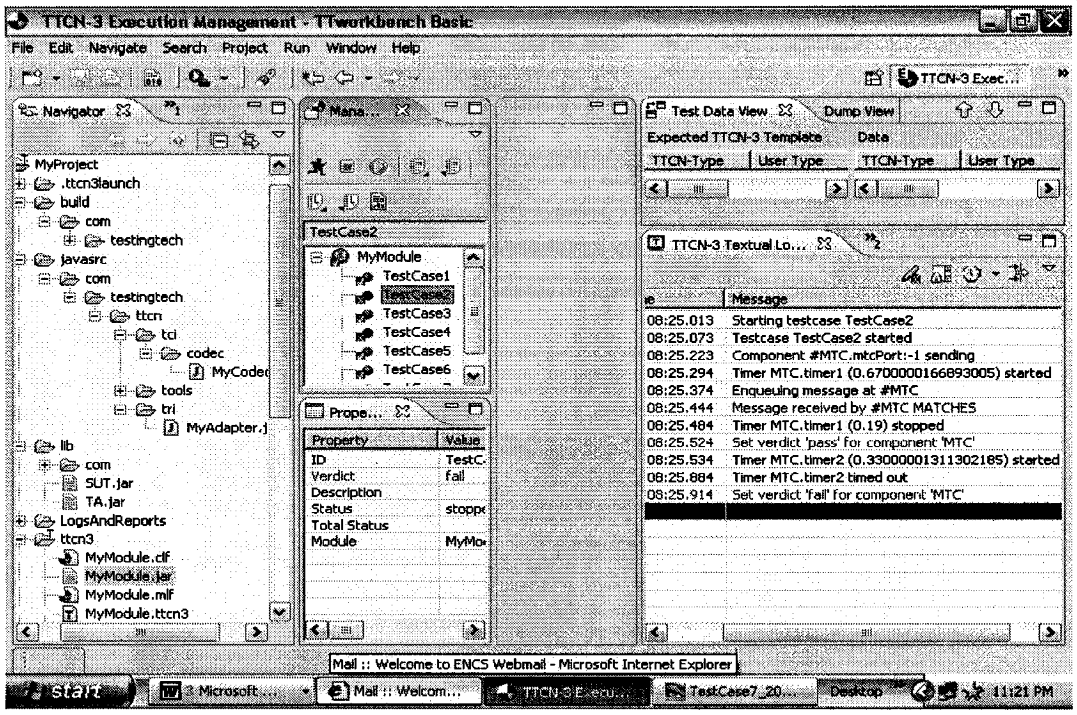


Property	Value
ID	TestCase7
Verdict	pass
Description	
Status	stopped
Total Status	
Module	MyModule

```
09:09.697  Testcase TestCase7 started
09:09.768  Component #MTC.mtcPort:-1 sending
09:09.858  Timer MTC.timer1 (0.6700000166893005) starte
09:09.978  Enqueuing message at #MTC
09:10.539  Timer MTC.timer1 timed out
09:10.549  Component #MTC.mtcPort:-1 sending
09:10.589  Enqueuing message at #MTC
09:10.619  Timer MTC.timer2 (0.6700000166893005) starte
09:11.310  Timer MTC.timer2 timed out
09:11.340  Component #MTC.mtcPort:-1 sending
09:11.390  Enqueuing message at #MTC
09:11.420  Timer MTC.timer3 (1.0) started
09:11.460  Message received by #MTC MATCHES
09:11.480  Timer MTC.timer3 (0.02) stopped
09:11.500  Set verdict 'pass' for component 'MTC'
09:11.510  Timer MTC.timer4 (0.33000001311302185) star
09:11.530  Message received by #MTC MATCHES
09:11.660  Timer MTC.timer4 (0.14) stopped
09:11.700  Set verdict 'pass' for component 'MTC'
```

Figure 4-9: Passed Test Case

In the following figure, we show the result of running a test case that is failed.



Property	Value
ID	TestC.
Verdict	fail
Description	
Status	stoppe
Total Status	
Module	MyMo

```
e             Message
08:25.013     Starting testcase TestCase2
08:25.073     Testcase TestCase2 started
08:25.223     Component #MTC.mtcPort:-1 sending
08:25.294     Timer MTC.timer1 (0.6700000166893005) started
08:25.374     Enqueuing message at #MTC
08:25.444     Message received by #MTC MATCHES
08:25.484     Timer MTC.timer1 (0.19) stopped
08:25.524     Set verdict 'pass' for component 'MTC'
08:25.534     Timer MTC.timer2 (0.3300000131130218 5) started
08:25.884     Timer MTC.timer2 timed out
08:25.914     Set verdict 'fail' for component 'MTC'
```

Figure 4-10: Failed Test Case

A graphical presentation is given in the next figure for a passed test case.

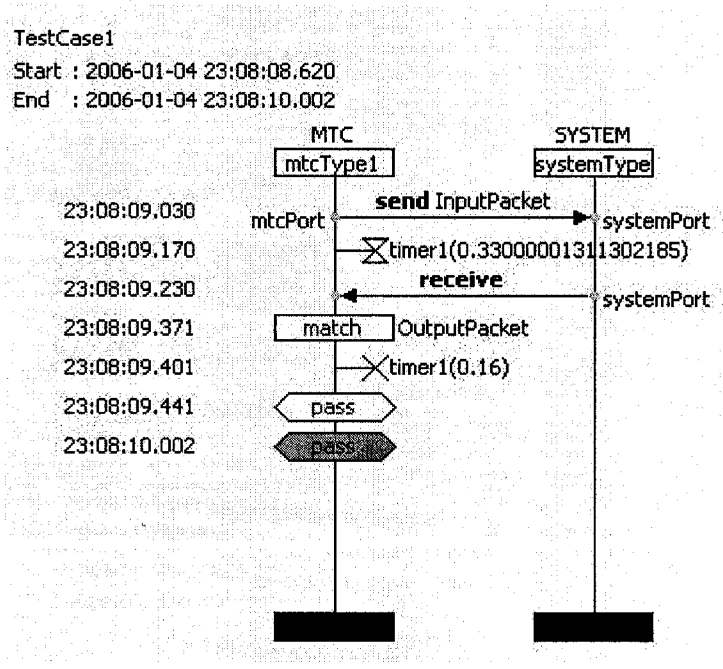


Figure 4-11: Graphical Result of a Passed Test Case

Another graphical presentation is shown in figure 4-12, but for a failed test case.

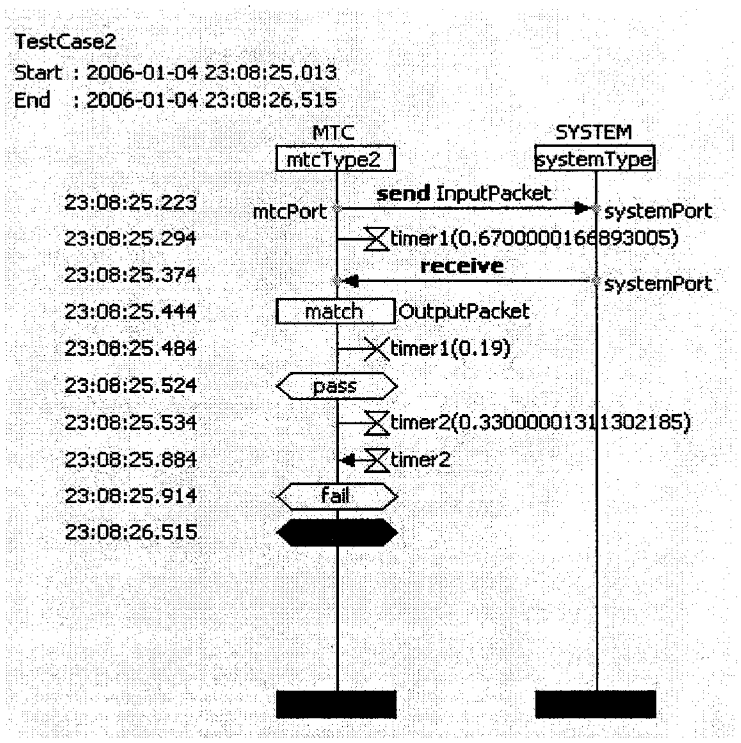


Figure 4-12: Graphical Result of a Failed Test case

4.8 Summary

In this chapter, we completed the process of the implementation of a test system for a given file containing test cases of a real-time application. To go from abstract to concrete level and complete the TE, we have shown the object oriented design for the tools to be used and the two interfaces: TCI and TRI Interfaces. Then, we wrote a codec and a test adapter programs to implement the required entities from the TCI and TRI interfaces. Moreover, a simulation to a system under test is also implemented. These complementary programs have been built to get the executable. After, to load the module for running, we created a Module Loader File (MLF). Running this file loads the test campaign allowing the tester to select the test cases to be run. Accordingly, the tester could see the test cases running one by one with a complete description of the run time details. In the next chapter, we will conclude our thesis and discuss the extensions and possible improvements regarding our approach.

CHAPTER 5
Conclusion and Future Work

Testing plays a key role in software life cycle. This is because in recent years, the quantity of software has developed and the dependency of society on this software is still increasing quickly. Nowadays, software systems are considered the centre functionality of electronics products. These systems have entered our daily lives and their rapidly growth is expected in the coming years. Moreover, the size and complexity of software systems have been increased demanding higher quality and shorter development time than ever. To undertake these rising requirements, testing has become a more significant stage in the development of a software system.

Most of cited systems are Real-Time Systems (RTS). The behaviour of RTS is time sensitive and governed by time constraints, which makes testing more important than ever. Real-time software is used to control safety critical systems. So, their applications are harder to test because they require not only the correct outcomes, but also to get these outcomes at the right time. This means to ensure their correctness and reliability with respect to both functional and real-time behaviour. The misbehaviour of RTS is generally due to the non-respect of the timing aspects and causes catastrophic consequences. For these reasons, testing such systems is one of the most challenging research areas and the development of new testing approaches to efficiently test RTS is an urgent need.

TTCN-3 is an international standardised test language for specification and implementation of complex test systems and solutions. It allows test automation which increases the efficiency of testing. It has been applied for different kinds of testing such as functional and conformance testing, regression testing, interoperability, integration, load or stress testing. The language is recommended in the third part of Conformance Testing Methodology and Framework and has also adopted for testing complex reactive and distributed systems. Hence, we have selected and used TTCN-3 to test our real-time multimedia application.

5.1 Contributions

Our work in this thesis was divided into two major phases. In the first phase, we developed a methodology to execute a given timed test suite for a real-time multimedia application. In the methodology, we created a program that automatically converts the given test cases into an abstract test suite represented by the TTCN-3 testing language. Then, we have created a TTCN-3 project and added the generated ATS to it. Then, we compiled the ATS and get its executable. The latter is considered the core part of TTCN-3 test system and is in a high level of abstraction. It needed to interact with the other parts of the system and to adapt to the SUT and platform. Thus, in order to run it, we needed some complementary work. We also highlighted the reusability of our program.

In the second phase of our development, we developed a codec to translate the abstract data to real world data and to decode the data in the inverse direction. With that codec, the executable (TE) was able to make a concrete communication with the SUT and data adaptation. We also created an adapter to adapt the TE to the SUT, so that the encoded/decoded data can be exchanged between the TE and SUT as sending stimuli and receiving responses during specified durations.

Additionally, we described and demonstrated the process of creating, implementing, and running the test system for our real-time application step by step. We also created a program to imitate a SUT in order to achieve a black-box testing. At the end of our development, we ran the test system and showed the automatic execution for the test cases of the application. In addition, we have noticed that the implemented test system can be run easily and automatically while displaying impressive details. Consequently, the tester can watch during the test run the involved test components along with their ports communications and their timers' values, operations, and durations. This is besides the ability to log the test for later review and to produce a HTML test report resuming the test results, as we have illustrated through the screenshots.

Before starting our development in the thesis, we introduced the language, gave the reasons to use it, and explained its structure that enabled us to generate the ATS through the methodology. Further, we explained the structure of the test system. The indicated explanations were important because the language is relatively new for many readers. So, we wanted them to get familiar with it in order to understand what we were talking about during the development.

Through this research, we experienced TTCN-3 as a testing language. Based on what we have learned about it (e.g. features, efficiency, flexibility, extendibility, portability, unambiguous operational semantics, C++/Java like syntax), we found it easy and simple to learn. Moreover, during our design and implementation in this research, we have applied the stated features and facilities of the language. We built our development in the intent to make our methodology helpful to the others for their testing purposes and save their time and cost. The generated ATS can be used as it is, modified to match other needs, or imported as a module to be included into other ATS to extend them. We pointed out also that the developed adaptors are flexible, so they developed one time, and then can be reusable to reduce investment cost in the future.

5.2 Extensions and Future Work

As stated before, testing distributed real-time systems is one of the most challenging research areas in testing. TTCN-3 has proven its efficiency for testing distributed systems. Although the timer mechanism for handling time is one of its powerful features, the development of the language concentrated on features for functional testing. Therefore, testing soft real-time requirements can easily be realized in TTCN-3. However, our work could be extended to address some concepts needed to test hard real-time requirements more efficiently. We have two possible extensions: TIMEDTTCN-3 for real-time requirements and TIMEDGFT for their graphical representation.

5.2.1 TIMEDTTCN-3: A Real-Time Extension for TTCN-3

*TIMED*TTCN-3 is a real-time extension for TTCN-3. It introduces new concepts to support the test and measurement of real-time requirements. This extension adds a new test verdict called *conf* (abbreviation for conforming) to signify a functional *pass* with an associated non-functional *fail*. It also introduces absolute time and timestamps. In addition, it allows the definition of synchronization requirements for test components and provides logging mechanisms for an offline evaluation to real-time requirements.

5.2.1.1 Time Extension

TTCN-3 supplies a timer mechanism for time handling. This timer mechanism is designed for supervising the functional behaviour of an IUT to avoid the blocking of a test case or to provoke exceptional behaviour. Additionally, the measurement of durations is affected by the TTCN-3 snapshot semantics and by the order of checking up the port queues and the timeout list [41].

Moreover, TTCN-3 doesn't have the concept of absolute time (e.g., a test component cannot read and use its local system time). In real-time testing, the absolute time is important to verify the relationships between observed test events and to coordinate test activities, and that is why this concept is added to *TIMED*TTCN-3. In case of distributed test environment, the system time may be exchanged among test components with synchronized clocks to check real-time requirements that cannot be measured locally or to timely coordinate among test activities.

5.2.1.2 Real-Time Properties Evaluation

While functional behaviour is basically tested by using sequences of send and receive operations, real-time requirements can be tested by relating particular points in time to each other. These points are called timestamps and serve to evaluate whether the points in time of interesting events fulfill a certain real-time requirement or not. We have two possibilities of timestamps evaluation: an online or an offline evaluation. Online evaluation is needed if it is not possible to separate functional and non-functional requirements, such as when a non-functional aspect directly influences the functional behaviour of a test case. In such a case, the evaluation of non-functional observations and timestamps must be performed during the test run in order to react on the result of the evaluation.

On the other hand, offline evaluation may be used when the non-functional requirements have no influence on the functional reaction of a test case. *TIMED*TTCN-3 offers a means to record timestamps into a log file during a test run in order to evaluate them afterwards. Depending on the timestamps in the log file, the non-functional aspects can be evaluated when the *testrun* has finished. The final test verdict is composed of the functional test verdict and result of the subsequent offline evaluations.

5.2.2 TIMEDGFT: A Real-Time Graphical Presentation

For the visualization of *TIMED*TTCN-3 test cases, a real-time extension for GFT called *TIMED*GFT is introduced to allow a graphical presentation of *TIMED*TTCN-3 constructs.

TimedTTCN3		TimedGFT
Concept	Realization	Presentation
Timezones	new parameter of create statement	MyTC:=CType.create (Berlin)
	new parameter of execute statement	MyTestCase (Berlin)
	timezone operation	no special symbol
Absolute time	now operation	no special symbol
	resume statement	@[t+3.0]
Logging	extension of log statement	MyTemplate
Logfile handling	first, next, previous and retrieve operations	no special symbols
Testrun handling	getlog operation	no special symbol
	overwriting of verdicts in control part	myTestrun. setverdict(fail)
Non-functional verdict	conf verdict	conf

Table 5-1: Real-Time Constructs of TIMEDGFT

As stated in chapter 2, GFT is one of the standardized presentation formats of TTCN-3. It provides an exact way of displaying the TTCN-3 behaviour descriptions (e.g. test cases, altsteps, functions and module control) graphically. GFT is based on the MSC standard. It uses a subset of MSC and extends this subset with particular test symbols and keywords. All TTCN-3 statements have an appropriate GFT symbols. Therefore, just the concepts of the *TIMED*TTCN-3 extension should be considered in the definition of the graphical real-time extension *TIMED*GFT. Table 5-1 summarizes the *TIMED*GFT presentation of the new *TIMED*TTCN-3 concepts and statements [44].

BIBLIOGRAPHY

[1] G. Liu, Timed Test Suite Generation Based on Test Purpose Expressed in MSC, Thesis, p. 6-22, Montreal, Quebec, Canada, February 2004.

[2] ISO, Information Technology, Open Systems Interconnection, Conformance testing methodology and framework. International Standard IS-9646, ISO, 1991.

[3] ITU-T, TTCN-2: The Tree and Tabular Combined Notation (TTCN), Conformance Testing Methodology and Framework, Part 3, Recommendation X.292, 1997.

[4] Z. Xiang, Testing Embedded Real-Time Systems Based on Test purposes, Thesis, p. 5-16, Montreal, Quebec, Canada, December 2004.

[5] A. En-Nouaary and G. Liu, Timed Test Suite Generation Based on Test Purpose, 1st International Conference on Information & Communication Technologies: form Theory to Applications, Damascus, Syria, ICTTA'04, April 2004, PDF 1-14 and pp. 1-18.

[6] G. Réthy, Application of TTCN-3 for 2.5 and 3G Testing, The TTCN-3 User Conference 2004, Sophia, Antipolis, France, May 2004, PDF 1-24.

[7] International Organization for Standardization, "Information technology - Open systems interconnection – Conformance testing methodology and framework - Part 3: The Tree and Tabular combined Notation (TTCN)", ISO/IEC 9646-3, 2nd ed., Geneva, November 1998.

[8] C. Willcock, Introduction to TTCN-3, The TTCN-3 User Conference 2005, Sophia, Antipolis, France, June 2005, PDF 1- 48.

[9] Testing and Test Control Components For Tester Developers, Finland, http://www.openttcn.com.

[10] ETSI ES 201 873-X TTCN-3 standards at www.etsi.org/ptcc/ptccttcn3.htm, also published as ITU-T Recommendation Z.140, Z.141, and Z.142.

[11] TTCN-3 Testing and Test Control Notation at www.ttcn-3.org/ttcn-3About

[12] C. Desroches, Comparing TTCN-3 and TTCN-2, TTCN-3 User Conference

2004, Sophia, Antipolis, France, May 2004, PDF 1-22.

[13] TTworkbench for TTCN-3 and Eclipse, Information available at www.testingtech.de/products/ttwb_intro.

[14] ETSI European Standard (ES) 201 873-1 V3.1.1 (2005-06) Methods for Testing and Specification (MTS), The Testing and Test Control Notation version 3, Part 1: TTCN-3 Core Language, p. 3-208.

[15] Testing Technologies TTCN-3, Germany: http://www.testingtech.de/ttcn3.

[16] ETSI Technical Report (TR) 101 666 (1999-05): Information technology - Open Systems Interconnection Conformance testing methodology and framework, The Tree and Tabular Combined Notation (TTCN) (Ed. 2++). ETSI, 1999.

[17] ETSI European Standard (ES) 201 873-2 V3.1.1 (2005-06) Methods for Testing and Specification (MTS), The Testing and Test Control Notation version 3, Part 2: TTCN-3 Tabular Presentation Format (TFT).

[18] ETSI European Standard (ES) 201 873-3 V3.1.1 (2005-06) Methods for Testing and Specification (MTS), The Testing and Test Control Notation version 3, Part 3: TTCN-3 Graphical Presentation Format (TFT).

[19] ETSI European Standard (ES) 201 873-6 V3.1.1 (2005-06) Methods for Testing and Specification (MTS), The Testing and Test Control Notation version 3, Part 6: TTCN-3 Control Interface, p. 3-255.

[20] I. Schieferdecker and T. Vassiliou-Gioles, Tool Supported Test Frameworks in TTCN-3, FMICS workshop, June 2003, pdf 1-10.

[21] ETSI European Standard (ES) 201 873-5 V3.1.1 (2005-06) Methods for Testing and Specification (MTS), The Testing and Test Control Notation version 3, Part 5: TTCN-3 Runtime Interface, p. 4-54.

[22] T. Vassiliou-Gioles, The Execution of TTCN-3 tests, The TTCN-3 Runtime and Control Interfaces, TTCN-3 User Conference 2004, Sophia, Antipolis, France, May 2004, PDF 1-73.

[23] European Telecommunications Standard Institute, Protocol and Testing Competence Center at www.etsi.org/ptcc.

[24] J. Grabowski and A. Ulrich, An Introduction to TTCN-3, The TTCN-3 User

Conference 2004, Sophia, Antipolis, France, May, 2004.

[25] S. Burton, A. Barsel, I. Scheiferdecker, Automated Testing of Automotive Telematics systems using TTCN-3, 3rd Workshop on System Testing and Validation - SV04, Paris, December 2004.

[26] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe, On the design of the new testing language TTCN-3, Proceedings of 13th International Conference on Testing of Communicating Systems (TestCom), Ottawa, Canada, August 2000, PDF 1-36.

[27] Z. R. Dai, P.H. Deussen, M. Busch, L. P. Lacmene, T. Ngwangwen, J. Herrmann, M. Schmidt. Automatic Test Data Generation for TTCN-3 using CTE, 18th International Conference Software & Systems Engineering and their Applications - ICSSEA 2005, Paris, November- December 2005, PDF 1-9.

[28] Test Execution Logging and Visualization Techniques, 17th International Conference Software & Systems Engineering and their Applications - ICSSEA 2004, Paris, France, November - December 2004.

[29] J. Grabowski, D. Hogrefe, G. Rethy, I. Schieferdecker, A. Wiles, and C. Willcock, An Introduction into the Testing and Test Control Notation (TTCN-3), Testing Technologies at http://www.testingtech.de/ttcn3/whitepaper.

[30] Implementation of TTCN-3 Test Systems using the TRI, IFIP 14th International Conference on Testing of Communicating Systems, TestCom 2002, March 2002.

[31] I. Schieferdecker, T. Vassiliou-Gioles, Realizing Distributed TTCN-3 Test Systems with TCI, IFIP TC 6 / WG 6.1 15th International Conference on Testing of Communicating Systems, TestCom 2003, Sophia Antipolis, France, May 2003.

[32] I. Schieferdecker, S. Pietsch, T. Vassiliou-Gioles. Systematic Testing of Internet Protocols, First Experiences in Using TTCN-3 for SIP, 5th Africom Conference on Communication Systems, May 2001

[33] M. Schünemann, I. Schieferdecker, A. Rennoch. M. Li, C. Desroches, Improving Test Software Using TTCN-3, GMD Report 153, 2001.

[34] M. Ebner, An Introduction to TTCN-3 Version3, ITU Study Group 17, Geneva,

5-14 October 2005.

[35] ITU-T: Recommendation: Message Sequence Chart (MSC), International Standard Z.120 (11/99) with Corrigendum 1, ITU-T, International Telecommunication Union-Telecommunication Standardisation Sector SG 10 (2001)

[36] A. En-Nouaary, R. Dssouli, F. Khendek. Timed Wp-Method: testing Real-time Systems, IEEE Transactions on Software Engineering, Volume 28, Issue 11 p. 1023 – 1038, November 2002.

[37] M. Ebner, TTCN-3 Test Case Generation from Message Sequence Charts, Institute for Informatics, Software Engineering Group, University of Göttingen, Germany, January 2005, pp. 1-19.

[38] A. En-Nouaary and G. Liu, Timed Test Suite Generation Based on Test Purpose,1st International Conference on Information and Communication Technologies: form Theory to Applications, Damascus, Syria, ICTTA'04-April 2004.

[39] A. En-Nouaary and R. Dssouli, A Guided Method for testing Timed Input Output Automata, 15th IFIP International Conference, TestCom 2003, Sophia Antipolis, France, May 2003.

[40] D. Apostolidis, D. Tepelmann, A. Rennoch, A. Vouffo, Use of TTCN-3 for the Development of SIGTRAN Tests, 18th International Conference Software & Systems Engineering and their Applications - ICSSEA 2005, Paris, November, December 2005.

[41] Z. Dai, J. Grabowski, H. Neukirchen, TIMEDTTCN-3 – A Real-Time Extension for TTCN-3, Proceedings of the IFIP TC6/WG6.1 14th International Conference on Testing of Communicating Systems, (TestCom 2002), Berlin, Germany, March 2002, PDF 1-18.

[42] J. Grabowski, Real-Time TTCN-3: *TIMED*TTCN-3 A Real-time Extension for TTCN-3, TTCN-3 User Conference 2004, Sophia, Antipolis, France, May 2004, pp.1-8.

[43] I. Schieferdecker, J. Grabowski, The Graphical Format of TTCN-3 in the

context of MSC and UML, Proceedings of the 3th International Workshop on SDL and MSC (SAM 2002), Telecommunications and beyond: The Broader Applicability of SDL and MSC,Aberystwyth, UK, June 2002.

[44] Z. R. Dai, J. Grabowski, and H. Neukirchen, TimedTTCN-3 Based Graphical Real-Time Test Specification Extension for TTCN-3, Proceedings of the IFIP TC6/WG6.1 15th International Conference on Testing of Communicating Systems, TestCom 2003, Sophia, Antipolis, France, May 2003, PDF 1-18.

[45] A. En-Nouaary, *Testing Real-Time Systems using Test Purposes*, International Workshop on Communication Software Engineering (IWCSE'2002), Marrakech, Morocco, December 2002.

APPENDIX A
Abstract Test Suite Generator program

```
/***********************************************
File name: ATS_Generator.cpp
***********************************************/

#include <iostream>
#include <fstream>
#include <string>

using namespace std;


/****************** Functions' Prototypes ******************/

// function to create TTCN3 module declarations
void declareTTCNmodule(ofstream& );

// function to extract test cases data from a text file and feed it to a TTCN3 module
void readInputFile(ifstream&, ofstream& , string [], int&, int&, int&, int&, int& );

// This function will create all TTCN3 test cases inside a module
void TTCNtestCases(ofstream& , string [], int& , int& , int& , int& , int&);

// function to bring an output token from the test case already read from a text file into
an array to feed it to a specific TTCN3 test case
void getOutput ( ofstream& , string [], int , int, int );

// This function will generate the control part of a TTCN3 module for execution
void executeTestCases(ofstream& fout, int testcasesCount);
```

```
        /****************** The main Function ******************/
int main()
{
    const int MAX = 40;
    string tcAry[MAX];      // array to hold a test case tokens
    int arySz = 0;          // array size
    int inCount = 0,        // counter for inputs
      outCount= 0,          // counter for outputs
        tcCount = 0,        // count the number of lines (testCases) in the input file
        timersCount = 0; // count for timers

    ifstream fin ("input.txt");
    ofstream fout ("output.txt");

    if (fin.fail ())
    {
     cerr << "ERROR: Cannot open the input file for reading.\n";
     exit(1);
    }

    if (fout.fail())
    {
     cerr << " ERROR: Cannot open the output file for writing.\n";
     exit(1);
    }

    declareTTCNmodule(fout);
    readInputFile(fin, fout, tcAry, arySz, inCount, outCount, timersCount, tcCount);

return 0;
}

        /*************** Functions Implementations ***************/
```

```
void declareTTCNmodule(ofstream& fout){

    /******** Start declaration to create TTCN3 module for test cases ********/

    fout<< "module MyModule {" << endl << endl;

    fout<< "\t type record inputPacket {" <<endl;
    fout<< "\t\t octetstring pktType length(2)," <<endl;
    fout<< "\t\t octetstring pktSize length(2)," <<endl;
    fout<< "\t\t octetstring pktData" <<endl;
    fout<< "\t }" << endl;

    fout<< endl;

    fout<< "\t template inputPacket image := {" <<endl;
    fout<< "\t\t pktType := '0003'O," <<endl;
    fout<< "\t\t pktSize:= '0004'O," <<endl;
    fout<< "\t\t pktData := '01001010'O" <<endl;
    fout<< "\t }" << endl;

    fout<< endl;

    fout<< "\t template inputPacket sound := {" <<endl;
    fout<< "\t\t pktType := '0001'O," <<endl;
    fout<< "\t\t pktSize:= '0004'O," <<endl;
    fout<< "\t\t pktData := '10100100'O" <<endl;
    fout<< "\t }" << endl;

    fout<< endl;
    fout<< "\t template inputPacket data := {" <<endl;
    fout<< "\t\t pktType := '0002'O," <<endl;
    fout<< "\t\t pktSize:= '0004'O," <<endl;
    fout<< "\t\t pktData := '10010010'O" <<endl;
fout<< "\t }" << endl;

fout<< "\t type record outputPacket {" <<endl;
fout<< "\t\t octetstring pktSize length(2)," <<endl;
fout<< "\t\t octetstring pktData" <<endl;
fout<< "\t    }" << endl;

fout<< endl;

fout<< "\t template outputPacket ackAll := {" <<endl;
fout<< "\t\t pktSize:= '0002'O," <<endl;
fout<< "\t\t pktData := '10001010'O" <<endl;
fout<< "\t }" << endl;

fout<< endl;

fout<< "\t template outputPacket reset := {" <<endl;
fout<< "\t\t pktSize:= '0002'O," <<endl;
fout<< "\t\t pktData := '10100100'O" <<endl;
fout<< "\t }" << endl;

fout<< endl;
fout<< "\t type port myPort message {" <<endl;
fout<< "\t\t out inputPacket;" <<endl;
fout<< "\t\t in outputPacket;" <<endl;
fout<< "\t }" << endl;

fout<< endl;

fout<< "\t type component systemType {" <<endl;
fout<< "\t\t port myPort systemPort;" <<endl;
fout<< "\t }" << endl;

fout<<  endl;
```

```
}

void readInputFile(ifstream& fin, ofstream& fout, string testCaseAry[], int& arySize,
int& inputCount, int& outputCount, int& timersCount, int& testcasesCount)
{

        char firstChar ;
        int i = 0, j = 0 , k = 0;
        string inputLine;  // each line represents a test case

        cout<< "The test cases extracted from the input file are the following:"<<endl;

        while(getline(fin,inputLine,'\n'))
        {
                cout<< endl <<"An inputLine is "<<endl<<inputLine<<endl;
                int leng = inputLine.size();
                testcasesCount++;

                i = inputLine.find(" ");
                string token = inputLine.substr(1,i-2);
                //cout<<"token is "<<token<<endl;
                //token = "input1";
                testCaseAry[arySize++] = token;
                inputCount++;
                int flag = 0;

                do {
                            j = i+1;
                            i = inputLine.find(" ",j); // find the position of the space
                                                          // between tokens

                            string token = inputLine.substr(j,i-j); // extract a token
                            //cout<<"token is "<<token<<endl;
                            firstChar = token.at(0);
```

```
if ( firstChar == '?' )
        inputCount++ ;
else if ( firstChar == '!' )
        outputCount++ ;
else
        ;

if( firstChar =='?' || firstChar =='!' ) // when the token is not a
                                          // timer value
{

        if(flag == 0)
        {
                testCaseAry[arySize++] = "0.0";
                timersCount++;
        }
        if (i < 0 ) // last token in the line (that's why no more
                    // spaces found)
                token =  inputLine.substr(j+1,leng-j-2);
        else
                token = inputLine.substr(j+1,i-j-2); // remove the
                                          // input or output prefix sign

        testCaseAry[arySize++] = token ;
        flag=0;
}

else  // when the token is a timer, insert it in the array
{

        token = inputLine.substr(j,i-j-1);
        testCaseAry[arySize++] = token;
        timersCount++;
        flag = 1;
}
```

```
        } while (i > 0); // while there are still tokens in the processed line


        cout<<"array size is "<<arySize <<" and timers number is "<<
              timersCount<<endl;


        for (k=0 ; k<arySize ; k++)
              cout << testCaseAry[k]<< " ";
        cout << endl;
        inputLine = ""; // reset the input line for the same reason
        cout << "You have "<< inputCount << " inputs in the current test case" <<
              endl;
        cout << "You have "<< outputCount << " outputs in the current test case"
              << endl;


        // call TTCNtestCases function to convert test case entries to a TTCN3 test
        // case
        TTCNtestCases(fout,testCaseAry, arySize, inputCount, outputCount,
                          timersCount, testcasesCount);

  }//end of while not eof
  cout<< endl << "You have " << testcasesCount << " testcases in the input file\n";

  // call the function excutetTestCases to create and add the control part to the
  // module file
  executeTestCases(fout, testcasesCount);
}


void TTCNtestCases(ofstream& fout, string testCaseAry[], int& arySize, int&
inputCount, int& outputCount, int& timersCount, int& testcasesCount)
{
  int i=0,
        j=0,
        k = 0 ;      //to count timers
```

```
fout<< endl;
fout<< "\t type component mtcType" << testcasesCount << " {" <<endl;
fout<< "\t\t port myPort mtcPort;" <<endl;
//Get timers values for a test case
for (j = 1 ; j < arySize ; j=j+2 ) // loop to declare and intialize the timers
{
        k++;  //timer count
        fout<< "\t\t timer timer" << k <<" := " << testCaseAry[j] << ";"<<endl;

}
fout<< "\t }" << endl;

fout<< endl;

//one test case starts:
fout<< "    testcase TestCase" << testcasesCount <<
          "() runs on mtcType" << testcasesCount << " system systemType" <<endl;
fout<< "    {" << endl;

fout<<endl;
fout<< "\t\t map(mtc:mtcPort, system:systemPort);" <<endl <<endl;
bool b = true;
i = 0;
k = 0;  // for timer
do{
        fout<< "\t\t mtcPort.send(" << testCaseAry[i++] << ");" <<endl;
        i++;  // take next token which is a timer value
        k++; // variable to hold the timer number


        if (k < inputCount) // if you still have inputs
        {
                fout<< "\t\t timer" << k << ".start;" <<endl;
                //wait until soundTimer expires
                fout<< "\t\t timer" << k << ".timeout;" <<endl;
                fout<< endl;
        }
        else
                b = false;

} while (k < inputCount); // as long as you have input token

// when you have more than one input, send the next input. Otherwise, go to the
// loop to get the outputs
if (b == true)
{//after last input token, get the timer value that is before the 1st output
        k++;
        fout<< "\t\t mtcPort.send(" << testCaseAry[i++] << ");" <<endl;
//      {
        //wait to receive ack


        i++;
}

j = 1; // j is used to count the number of the actual output
do{             //as long as we have output token
        getOutput ( fout,testCaseAry, i, k, j);
        i = i+2; // because each other token is an output while the other is a timer
                 // value
        j++;
        k++;
}while (j <= outputCount);

fout<< "\t\t unmap(mtc:mtcPort, system:systemPort);" <<endl;
fout<< "\t } " << endl<< endl; //end of testcase

//reset parameters:
i = 0;
j = 0;
```

```
        k = 0;
        inputCount = 0;
        outputCount = 0;
        timersCount = 0;
        arySize = 0; //reset the array for fresh start with a new testcase

}


// getOutput function definition:
void getOutput (ofstream& fout, string testCaseAry[], int i , int k, int j)
{
        if (j == 1) //to process first output token
        {
                fout<< "\t\t timer" << k << ".start;" << endl;
                fout<< endl;

                fout<< "\t\t alt{" <<endl;
                fout<< "\t\t\t [] mtcPort.receive(" << testCaseAry[i] << ")" <<endl;
                fout<< "\t\t\t {" <<endl;
                fout<< "\t\t\t\t timer" << k <<".stop;" <<endl;
                fout<< "\t\t\t\t setverdict(pass);" << endl;
                fout<< "\t\t\t }" << endl << endl;

                fout<< "\t\t\t [] mtcPort.receive" <<endl; //receiving other than ack
                fout<< "\t\t\t {" <<endl;
                fout<< "\t\t\t\t timer" << k <<".stop;" << endl;
                fout<< "\t\t\t\t setverdict(fail);" << endl;
                fout<< "\t\t\t }" << endl << endl;

                fout<< "\t\t\t [] timer" << k <<".timeout" <<endl;  // receiving (fragment)
                                                                    // other than ack
                fout<< "\t\t\t {" <<endl;
                fout<< "\t\t\t\t setverdict(fail);" << endl;
                fout<< "\t\t\t }" <<endl;
        fout<< "\t\t } " << endl; //end of alt statement
        fout<< endl;
}

else  //to process all next output tokens
{
        // whether verdict is "pass" or "fail", it follows overwriting rules

        fout<< "\t\t timer" << k << ".start;" << endl;
        fout<< endl;

        fout<< "\t\t alt {" <<endl;
        fout<< "\t\t\t [] mtcPort.receive(" << testCaseAry[i] << ")" <<endl;
        fout<< "\t\t\t {" <<endl;
        fout<< "\t\t\t\t timer" << k <<".stop;" <<endl;
        fout<< "\t\t\t\t setverdict(pass);" << endl;
        fout<< "\t\t\t }" << endl << endl;

        fout<< "\t\t\t [] mtcPort.receive" <<endl; //receiving other than ack
        fout<< "\t\t\t {" <<endl;
        fout<< "\t\t\t\t timer" << k <<".stop;" << endl;
        fout<< "\t\t\t\t setverdict(fail);" << endl;
        fout<< "\t\t\t }" << endl << endl;

        fout<< "\t\t\t [] timer" << k <<".timeout" <<endl;  // receiving fragment
                                                           // instead of ack
        fout<< "\t\t\t {" <<endl;
        fout<< "\t\t\t\t setverdict(fail);" << endl;
        fout<< "\t\t\t }" <<endl;

        fout<< "\t\t } " << endl ; //end of alt statement
        fout<< endl;

}
```

```
}

void executeTestCases(ofstream& fout, int testcasesCount)
{
        fout<< endl;
        fout<< "\t control {" << endl << endl; //start control part

        for (int i = 1; i <= testcasesCount ; i++)
        {
                // i is the counter to hold test cases number
        fout << "\t\t var verdicttype testcase" << i << "Result := execute(TestCase" << i
             << "());" ;
        fout << endl << endl;
        }

        fout << "\t }" << endl; //end of control part
        fout << "}" << endl; //end of module

}
```

APPENDIX B

TTCN-3 Abstract Test Suite (MyModule.ttcn)

```
module MyModule {

      type record inputPacket {
           octetstring pktType length(2),
           octetstring pktSize length(2),
           octetstring pktData
      }

      template inputPacket image := {
           pktType := '0003'O,
           pktSize:= '0004'O,
           pktData := '01001010'O
      }

      template inputPacket sound := {
           pktType := '0001'O,
           pktSize:= '0004'O,
           pktData := '10100100'O
      }

      template inputPacket data := {
           pktType := '0002'O,
           pktSize:= '0002'O,
           pktData := '10010010'O
      }
      type record outputPacket {
           octetstring pktSize length(2),
           octetstring pktData
          }

      template outputPacket ackAll := {
           pktSize:= '0002'O,
           pktData := '10001010'O
      }

      template outputPacket reset := {
           pktSize:= '0002'O,
           pktData := '10100100'O
      }

      type port myPort message {
           out inputPacket;
           in outputPacket;
      }

      type component systemType {
           port myPort systemPort;
      }
```

```
   type component mtcType1 {
        port myPort mtcPort;
        timer timer1 := 0.33;
   }

testcase TestCase1() runs on mtcType1 system systemType
{

        map(mtc:mtcPort, system:systemPort);

        mtcPort.send(image);
        timer1.start;

        alt{
             [] mtcPort.receive(ackAll)
             {
                  timer1.stop;
                  setverdict(pass);
             }

             [] mtcPort.receive
             {
                  timer1.stop;
                  setverdict(fail);
             }

             [] timer1.timeout
             {
                  setverdict(fail);
             }
        }

        unmap(mtc:mtcPort, system:systemPort);
   }


   type component mtcType2 {
        port myPort mtcPort;
        timer timer1 := 0.67;
        timer timer2 := 0.33;
   }

  testcase TestCase2() runs on mtcType2 system systemType
  {

        map(mtc:mtcPort, system:systemPort);

        mtcPort.send(image);
        timer1.start;

        alt{
             [] mtcPort.receive(ackAll)
             {
                  timer1.stop;
                  setverdict(pass);
```

```
            }

            [] mtcPort.receive
            {
                timer1.stop;
                setverdict(fail);
            }

            [] timer1.timeout
            {
                setverdict(fail);
            }
        }

        timer2.start;

        alt {
            [] mtcPort.receive(reset)
            {
                timer2.stop;
                setverdict(pass);
            }

            [] mtcPort.receive
            {
                timer2.stop;
                setverdict(fail);
            }

            [] timer2.timeout
            {
                setverdict(fail);
            }
        }

        unmap(mtc:mtcPort, system:systemPort);
    }


    type component mtcType3 {
        port myPort mtcPort;
        timer timer1 := 0.33;
        timer timer2 := 0.67;
    }

    testcase TestCase3() runs on mtcType3 system systemType
    {

        map(mtc:mtcPort, system:systemPort);

        mtcPort.send(image);
        timer1.start;
        timer1.timeout;

        mtcPort.send(sound);
        timer2.start;
```

```
      alt{
           [] mtcPort.receive(ackAll)
           {
                timer2.stop;
                setverdict(pass);
           }

           [] mtcPort.receive
           {
                timer2.stop;
                setverdict(fail);
           }

           [] timer2.timeout
           {
                setverdict(fail);
           }
      }

      unmap(mtc:mtcPort, system:systemPort);
 }


 type component mtcType4 {
      port myPort mtcPort;
      timer timer1 := 0.33;
      timer timer2 := 0.33;
 }

testcase TestCase4() runs on mtcType4 system systemType
{

      map(mtc:mtcPort, system:systemPort);

      mtcPort.send(image);
      timer1.start;
      timer1.timeout;

      mtcPort.send(sound);
      timer2.start;

      alt{
           [] mtcPort.receive(ackAll)
           {
                timer2.stop;
                setverdict(pass);
           }

           [] mtcPort.receive
           {
                timer2.stop;
                setverdict(fail);
           }

           [] timer2.timeout
           {
                setverdict(fail);
                }
            }

            unmap(mtc:mtcPort, system:systemPort);
        }


        type component mtcType5 {
            port myPort mtcPort;
            timer timer1 := 1.33;
            timer timer2 := 0.33;
            timer timer3 := 0.67;
        }

        testcase TestCase5() runs on mtcType5 system systemType
        {

            map(mtc:mtcPort, system:systemPort);

            mtcPort.send(image);
            timer1.start;
            timer1.timeout;

            mtcPort.send(sound);
            timer2.start;

            alt{
                [] mtcPort.receive(ackAll)
                {
                    timer2.stop;
                    setverdict(pass);
                }

                [] mtcPort.receive
                {
                    timer2.stop;
                    setverdict(fail);
                }

                [] timer2.timeout
                {
                    setverdict(fail);
                }
            }

            timer3.start;

            alt {
                [] mtcPort.receive(reset)
                {
                    timer3.stop;
                    setverdict(pass);
                }

                [] mtcPort.receive
                {
                    timer3.stop;
```

```
                  setverdict(fail);
              }

              [] timer3.timeout
              {
                  setverdict(fail);
              }
          }

          unmap(mtc:mtcPort, system:systemPort);
 }


 type component mtcType6 {
      port myPort mtcPort;
      timer timer1 := 0.33;
      timer timer2 := 0.67;
      timer timer3 := 0.33;
 }

testcase TestCase6() runs on mtcType6 system systemType
{

      map(mtc:mtcPort, system:systemPort);

      mtcPort.send(image);
      timer1.start;
      timer1.timeout;

      mtcPort.send(sound);
      timer2.start;

      alt{
           [] mtcPort.receive(ackAll)
           {
                timer2.stop;
                setverdict(pass);
           }

           [] mtcPort.receive
           {
                timer2.stop;
                setverdict(fail);
           }

           [] timer2.timeout
           {
                setverdict(fail);
           }
      }

      timer3.start;

      alt {
           [] mtcPort.receive(reset)
           {
                timer3.stop;
```

```
                    setverdict(pass);
                }

                [] mtcPort.receive
                {
                    timer3.stop;
                    setverdict(fail);
                }

                [] timer3.timeout
                {
                    setverdict(fail);
                }
            }

            unmap(mtc:mtcPort, system:systemPort);
        }


        type component mtcType7 {
            port myPort mtcPort;
            timer timer1 := 0.67;
            timer timer2 := 0.67;
            timer timer3 := 1.00;
            timer timer4 := 0.33;
        }

      testcase TestCase7() runs on mtcType7 system systemType
      {

            map(mtc:mtcPort, system:systemPort);

            mtcPort.send(image);
            timer1.start;
            timer1.timeout;

            mtcPort.send(sound);
            timer2.start;
            timer2.timeout;

            mtcPort.send(data);
            timer3.start;

            alt{
                [] mtcPort.receive(ackAll)
                {
                    timer3.stop;
                    setverdict(pass);
                }

                [] mtcPort.receive
                {
                    timer3.stop;
                    setverdict(fail);
                }

                [] timer3.timeout
            {
                setverdict(fail);
            }
        }

        timer4.start;

        alt {
            [] mtcPort.receive(reset)
            {
                timer4.stop;
                setverdict(pass);
            }

            [] mtcPort.receive
            {
                timer4.stop;
                setverdict(fail);
            }

            [] timer4.timeout
            {
                setverdict(fail);
            }
        }

        unmap(mtc:mtcPort, system:systemPort);
    }


    control {

        var verdicttype testcase1Result := execute(TestCase1());

        var verdicttype testcase2Result := execute(TestCase2());

        var verdicttype testcase3Result := execute(TestCase3());

        var verdicttype testcase4Result := execute(TestCase4());

        var verdicttype testcase5Result := execute(TestCase5());

        var verdicttype testcase6Result := execute(TestCase6());

        var verdicttype testcase7Result := execute(TestCase7());

    }
}
```

APPENDIX C

Packages and Class Hierarchy in Java Doc

Packages	
com.testingtech.ttcn.tci.codec	
com.testingtech.ttcn.tools	
com.testingtech.ttcn.tri	

Hierarchy For All Packages

Package Hierarchies:
com.testingtech.ttcn.tci.codec, com.testingtech.ttcn.tools, com.testingtech.ttcn.tri

---

Class Hierarchy

- java.lang.Object
  - com.testingtech.ttcn.tci.codec.base.AbstractBaseCodec (implements org.etsi.ttcn.tci.TciCDProvided)
    - com.testingtech.ttcn.tci.codec.**MyCodec** (implements org.etsi.ttcn.tci.TciCDProvided)
  - com.testingtech.ttcn.tools.**MySUT**
  - com.testingtech.ttcn.tri.TestAdapter (implements com.testingtech.ttcn.tci.TciEncoding, org.etsi.ttcn.tri.TriCommunicationSA, org.etsi.ttcn.tri.TriPlatformPA)
    - com.testingtech.ttcn.tri.**MyAdapter**
  - java.lang.Thread (implements java.lang.Runnable)
    - com.testingtech.ttcn.tools.**MySUT.ServerThreads**

APPENDIX D

MyCodec JavaDoc

Package com.testingtech.ttcn.tci.codec

Class Summary	
MyCodec	This codec program converts the messages sent as data types defined in the ATS into bitstring. It also converts the bitstring sent from the SUT to the TE into TTCN-3 values

---

com.testingtech.ttcn.tci.codec

Class MyCodec

```
java.lang.Object
  └ com.testingtech.ttcn.tci.codec.base.AbstractBaseCodec
      └ com.testingtech.ttcn.tci.codec.MyCodec
```

All Implemented Interfaces:

org.etsi.ttcn.tci.MetaDecoder, org.etsi.ttcn.tci.MetaEncoder, org.etsi.ttcn.tci.TciCDProvided

---

```
public class MyCodec
extends com.testingtech.ttcn.tci.codec.base.AbstractBaseCodec
implements org.etsi.ttcn.tci.TciCDProvided
```

This codec program converts the messages sent as data types defined in the ATS into bitstring (to ensure that the SUT can understand them). It also converts the bitstring (messages) sent from the SUT to the TE into TTCN-3 values. So, the codec job is to implement both *encode* and *decode* operations within the interface TciCDProvided.

Field Summary	
`private byte[]`	**byteMsg**

Fields inherited from class com.testingtech.ttcn.tci.codec.base.AbstractBaseCodec

```
bitpos, decodedValue, RB
```

Constructor Summary

MyCodec(de.tu_berlin.cs.uebb.muttcn.runtime.RB rb) Creates a new object of MyCodec type.

Method Summary

private org.etsi.ttcn.tci.RecordValue	**createOutputPacket**(org.etsi.ttcn.tci.RecordValue outPkt, byte[] byteMsg) This operation feeds the outputPacket with the received bytes.
org.etsi.ttcn.tci.Value	**decode**(org.etsi.ttcn.tri.TriMessage rcvMsg, org.etsi.ttcn.tci.Type decodeHypo) This method decodes a received message according to the decoding rules.
org.etsi.ttcn.tri.TriMessage	**encode**(org.etsi.ttcn.tci.Value val) This method encodes a sent message based on the basic encoding rules in the AbstractBaseCodec implementation.

Methods inherited from class com.testingtech.ttcn.tci.codec.base.AbstractBaseCodec

```
createBitstringValue, createBooleanValue, createBooleanValue,
createCharstringValue, createCharValue, createFloatValue,
createHexstringValue, createIntegerValue, createIntegerValue,
createOctetstringValue, createUniversalCharstringValue,
encodeBitstring, encodeBoolean, encodeCharstring, encodeHexstring,
encodeInteger, encodeNativeType, encodeNativeValue, encodeOctetstring,
encodeRecord, encodeRecordof, encodeUnion, encodeUniversalCharstring,
getBit, logDecode, logEncode, setField, setField, setField
```

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Field Detail

byteMsg

`private byte[] **byteMsg**`

Constructor Detail

MyCodec

```
public MyCodec(de.tu_berlin.cs.uebb.muttcn.runtime.RB rb)
```

Creates a new object of MyCodec type.

Parameters:

`rb` - has to be used to determine some particular logging information.

Method Detail

encode

```
public org.etsi.ttcn.tri.TriMessage encode(org.etsi.ttcn.tci.Value val)
```

This method encodes a sent message based on the basic encoding rules in the AbstractBaseCodec implementation.

Specified by:

`encode` in interface `org.etsi.ttcn.tci.TciCDProvided`

Overrides:

`encode` in class `com.testingtech.ttcn.tci.codec.base.AbstractBaseCodec`

Parameters:

`val` - is a variable of a TTCN-3 type value to be encoded.

Returns:

Tri-encoded message in a byte array format.

---

decode

```
public org.etsi.ttcn.tci.Value
decode(org.etsi.ttcn.tri.TriMessage rcvMsg,
                                      org.etsi.ttcn.tci.Type decodeHypo)
```

This method decodes a received message according to the decoding rules.

Specified by:

`decode` in interface `org.etsi.ttcn.tci.TciCDProvided`

Overrides:

`decode` in class `com.testingtech.ttcn.tci.codec.base.AbstractBaseCodec`

Parameters:

`rcvMsg` - The received message that should be decoded.

`decodeHypo` - is the decoding hypothesis against which the message will be decoded.

Returns:

a decoded message of type triMessage based on the decoding hypothesis or `null` if message was not of this hypothesis type.

createOutputPacket

```
private org.etsi.ttcn.tci.RecordValue

createOutputPacket(org.etsi.ttcn.tci.RecordValue outPkt,byte[] byteMsg)
```

This operation feeds the outputPacket with the received bytes

Parameters:

`outPkt` - The record that must be filled in.

`byteMsg` - The received message.

Returns:

the outPkt that has been filled with the received bytes.

APPENDIX E
MySUT JavaDoc

Class MySUT

Package com.testingtech.ttcn.tools

Class Summary	
MySUT	The class `MySUT` emulates the job of a system under test for this project.

Class Hierarchy

- java.lang.Object

3.1 com.testingtech.ttcn.tools.**MySUT**
3.2 java.lang.Thread (implements java.lang.Runnable)

  - com.testingtech.ttcn.tools.**MySUT.ServerThreads**

Class MySUT

```
java.lang.Object
  └─com.testingtech.ttcn.tools.MySUT
```

```
 class MySUT
extends java.lang.Object
```

The class `MySUT` emulates the job of a system under test for this project. Its objective is to react to stimuli coming from the test system

Nested Class Summary	
class	**MySUT.ServerThreads**

Field Summary

private int	**MSG_LENG**

Constructor Summary
MySUT()   Creates a new object of type MySUT.

Method Summary	
static void	**main**(java.lang.String[] args) The main method calls through its new object the openSockets operation for communication
void	**openSockets**() This method will open a sender and a receiver socket.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

MSG_LENG

```
private final int MSG_LENG
```

Constructor Detail

MySUT

```
public MySUT()
```

Creates a new object of type MySUT.

Method Detail

main

```
public static void main(java.lang.String[] args)
```

The main method calls through its new object the openSockets operation for communication

Parameters:

`args` -

---

openSockets

```
public void openSockets()
```

This method will open a sender and a receiver socket. Then, it begins listening on the receiver socket.

---

com.testingtech.ttcn.tools

Class MySUT.ServerThreads

```
java.lang.Object
  └─java.lang.Thread
      └─com.testingtech.ttcn.tools.MySUT.ServerThreads
```

All Implemented Interfaces:

java.lang.Runnable

Enclosing class:

MySUT

```
public class MySUT.ServerThreads
extends java.lang.Thread
```

Nested Class Summary

Nested classes/interfaces inherited from class java.lang.Thread
java.lang.Thread.State, java.lang.Thread.UncaughtExceptionHandler

Field Summary

private java.net.Socket	**socket**

Fields inherited from class java.lang.Thread

MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY

Constructor Summary

MySUT.ServerThreads(java.net.Socket socket)

Method Summary

void	**run**()

Methods inherited from class java.lang.Thread

activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getAllStackTraces, getContextClassLoader, getDefaultUncaughtExceptionHandler, getId, getName, getPriority, getStackTrace, getState, getThreadGroup, getUncaughtExceptionHandler, holdsLock, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setDefaultUncaughtExceptionHandler, setName, setPriority, setUncaughtExceptionHandler, sleep, sleep, start, stop, stop, suspend, toString, yield

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

socket

private java.net.Socket **socket**

Constructor Detail

MySUT.ServerThreads

`public MySUT.ServerThreads(java.net.Socket socket)`

Method Detail

run

`public void run()`

> **Specified by:**
> `run` in interface `java.lang.Runnable`
> **Overrides:**
> `run` in class `java.lang.Thread`

APPENDIX F

MyAdapter JavaDoc

Package com.testingtech.ttcn.tri

Class Summary	
MyAdapter	This test adapter will send messages via a given port to the SUT and enqueue messages received from the SUT.

com.testingtech.ttcn.tri

Class MyAdapter

```
java.lang.Object
  └com.testingtech.ttcn.tri.TestAdapter
      └com.testingtech.ttcn.tri.MyAdapter
```

All Implemented Interfaces:

com.testingtech.ttcn.tci.TciEncoding, java.io.Serializable, org.etsi.ttcn.tri.TriCommunicationSA, org.etsi.ttcn.tri.TriPlatformPA

```
public class MyAdapter
extends com.testingtech.ttcn.tri.TestAdapter
```

This test adapter is used for running the runtime test. It sends messages via a given port to the SUT and enqueues messages received from the SUT using the triSend method.

See Also:

Serialized Form

Field Summary

Fields inherited from class com.testingtech.ttcn.tri.TestAdapter
codecs, CsaDef, Cte, decoders, encoders, err, EXT_FUNC_CODEC, out, parameterCodec, PpaDef, Pte, Rb, RB, tciCHProvided, tciTMProvided, TestAdapter

Constructor Summary
MyAdapter()   Creates a new Adapter object.

Method Summary	
org.etsi.ttcn.tci.TciCDProvided	**getCodec**(java.lang.String encodingName) Returns the BaseCodec implementation for simple tests
org.etsi.ttcn.tri.TriStatus	**triSend**(org.etsi.ttcn.tri.TriComponentId componentId, org.etsi.ttcn.tri.TriPortId tsiPortId, org.etsi.ttcn.tri.TriAddress sutAddress, org.etsi.ttcn.tri.TriMessage message) This operation is called by the TE when it executes a TTCN-3 send operation on a component port, which has been mapped to a TSI port.

Methods inherited from class com.testingtech.ttcn.tri.TestAdapter
addExternalFunctionImpl, addSUTActionImpl, cancel, disposeExternalFunctionImpl, enterTriExternalFunction, getValueForParam, inExternalFunction, leaveTriExternalFunction, print, println, setComm, setRB, setTciTMProvided, setValue, tciConnect, tciMapReq, tearDown, triCall, triExecuteTestcase, triExternalFunction, triMap, triPAReset, triRaise, triReadTimer, triReply, triSAReset, triStartTimer, triStopTimer, triSutActionInformal, triSutActionTemplate, triTimerRunning, triUnmap, warn

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

MyAdapter

`public **MyAdapter**()`

Creates a new Adapter object.

Method Detail

triSend

```
public org.etsi.ttcn.tri.TriStatus
triSend(org.etsi.ttcn.tri.TriComponentId componentId,

org.etsi.ttcn.tri.TriPortId tsiPortId,

org.etsi.ttcn.tri.TriAddress sutAddress,

org.etsi.ttcn.tri.TriMessage message)
```

This operation is called by the TE when it executes a TTCN-3 send operation on a component port, which has been mapped to a TSI port. Encoding the sendMessage must be done in the TE before calling this TRI operation.

Specified by:

`triSend` in interface `org.etsi.ttcn.tri.TriCommunicationSA`

Overrides:

`triSend` in class `com.testingtech.ttcn.tri.TestAdapter`

Parameters:

`componentId` - the identifier of the sending test component

`tsiPortId` - the identifier of the test system interface port via which the message is sent to the SUT Adapter

`sutAddress` - the destination address within the SUT

`Message` - the encoded message to be sent to or received from the SUT

Returns:

tri status of the triSend operation. It returns (TRI_OK) to indicate the local success or (TRI_Error) to indicate failure of the send operation.

---

getCodec

```
public org.etsi.ttcn.tci.TciCDProvided
getCodec(java.lang.String encodingName)
```

Returns the BaseCodec implementation for simple tests

Specified by:

`getCodec` in interface `com.testingtech.ttcn.tci.TciEncoding`

Overrides:

`getCodec` in class `com.testingtech.ttcn.tri.TestAdapter`

Parameters:

`encodingName` - the encoding name described by the encoding attribute in the TTCN-3 ATS

Returns:

The codec corresponding to the given encoding rules or null if no appropriate coding has been found

APPENDIX G

Build File

```
/*
 * (C) Copyright Testing Technologies, 2001-2005.  All Rights Reserved.
 *
 * All copies of this program, whether in whole or in part, and whether
 * modified or not, must display this and all other embedded copyright
 * and ownership notices in full.
 *
 * See the file COPYRIGHT for details of redistribution and use.
 *
 * You should have received a copy of the COPYRIGHT file along with
 * this file; if not, write to the Testing Technologies,
 * Rosenthaler Str. 13, 10119 Berlin, Germany.
 *
 * TESTING TECHNOLOGIES DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
 * SOFTWARE. IN NO EVENT SHALL TESTING TECHNOLOGIES BE LIABLE FOR ANY
 * SPECIAL, DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN
 * AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION,
 * ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF
 * THIS SOFTWARE.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
 * EITHER EXPRESSED OR IMPLIED, INCLUDING ANY KIND OF IMPLIED OR
 *EXPRESSED WARRANTY OF NON-INFRINGEMENT OR THE IMPLIED WARRANTIES
 * OF MECHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
 *
<target name="all" depends="init, clean, classes, TA.jar, SUT.jar"/>

  <!-- Compile MyProject classes -->
```

```
<target name="classes" depends="init">
  <echo>Compile MyProject Test Adapter and Codec</echo>
     <javac srcdir="${src}" destdir="${classes}">
    <classpath>
      <pathelement location="${TTthree}/lib/TTthreeRuntime.jar"/>
        <pathelement location="${TTthree}/lib/TTorg.jar"/>
    </classpath>
  </javac>
</target>

<!-- Make TA.jar -->
<target name="TA.jar" depends="init">
  <echo>Create TA.jar</echo>
  <jar    jarfile="${lib}/TA.jar" basedir="${classes}"
includes="**/MyAdapter*.class,**/MyCodec*.class">
  </jar>
</target>

<!-- Make SUT.jar -->
<target name="SUT.jar" depends="init">
  <echo>Create SUT.jar</echo>
  <jar jarfile="${lib}/SUT.jar" basedir="${classes}" includes="**/MySUT*.class">
    <manifest>
      <attribute name="Main-Class" value="com.testingtech.ttcn.tools.MySUT"/>
    </manifest>
  </jar>
</target>

<!-- Remove created classes -->
<target name="clean" depends="init">
  <delete file="${lib}/TA.jar"/>
  <delete file="${lib}/SUT.jar"/>
  <delete>
            <fileset dir="${classes}" includes="**/*.class"/>
 </delete>
</target>
```