Department of Computer Science

Per Johansson, Henrik Wallinder

A Test Tool Framework for an Integrated Test Environment in the Telecom Domain

D-level Thesis (30 ECTS)

2005:04

A Test Tool Framework for an Integrated Test Environment in the Telecom Domain

Per Johansson, Henrik Wallinder

© 2005 Per Johansson, Henrik Wallinder and Karlstad University

This thesis is submitted in partial fulfillment of the requirements for the Master's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Per Johansson, Henrik Wallinder

(perjoha@gmail.com, henrik.wallinder@bredband.net)

Approved, June 7, 2005

Opponents: Malin Abrahamsson, Aleksandra Gadji

Advisor Karlstad University: Katarina Asplund

Advisor TietoEnator, Karlstad: Lars Lundegård

Examiner: Donald F. Ross

Abstract

This thesis is the result of a Master’s Project in Computer Science at Karlstad University performed by Per Johansson and Henrik Wallinder in 2005. The project was carried out at the Telecom R&D, Test Tools & Solutions department of TietoEnator in Karlstad, which develops support systems for different telecom platforms. The purpose of the project was to study different test tool frameworks that can be used for creating an integrated test environment. The goal of the project was to find a product that TietoEnator could use in future projects. The method used was to first specify some basic requirements for a test tool framework, then carry out a market analysis to find candidate products, and finally build a prototype as a proof of concept for the product that best matched the specified requirements. The requirements include infrastructure for remote test bed launch and execution as well as centralized functions for building new test tools. The result from the market analysis was that the product that best fulfilled the stated requirements was a product from the open source project Eclipse: the Test and Performance Tools Platform (TPTP). A functioning prototype was built using Eclipse TPTP. The prototype makes it possible for a tester to prepare, run and evaluate a test executed on a remote machine. A final conclusion from the project is that there remains some work with additional functionality and documentation before Eclipse TPTP is mature to use in real projects, but that Eclipse TPTP has good potential for being a quality test tool framework with a rich set of functions in the future.

Acknowledgements

We would like to thank the following persons for helping us with the project described in the thesis:

- Lars Lundegård, our advisor at TietoEnator, who made this project possible two days before Christmas and has been encouraging throughout the project.
- Mats Berglund at Ericsson, Linköping, who has given us valuable feedback on our work, and has shared his expertise and experience of testing in the telecom domain with us.
- Katarina Asplund, our advisor at Karlstad University, who has patiently reviewed our thesis during the project.
- Johan Andersson, our co-worker at TietoEnator, who has helped us come over the Eclipse threshold.
- Magnus Einarsson at TietoEnator, who came up with the well thought-out suggestion for a prototype to implement.
- Lars Ohlén at TietoEnator, who has shared his knowledge and experience of working with Eclipse with us.
- Joe Toomey at IBM Rational, who has answered several questions necessary for us to understand the Eclipse TPTP design model and to succeed with building the prototype.
- Vesa-Matti Puro at OpenTTCN Oy, who has reviewed our survey of OpenTTCN and given us feedback on our work.
- Patrick Krånglin and Per Blysa at Telelogic AB, who has reviewed our survey of Telelogic TAU/Tester.

- Ove Teigen, ThinTech AS, distributor of Scapa Technologies, who has reviewed our survey of Scapa Test and Performance Platform 3.1.

Contents

1 Introduction ... 1

1.1 Project Goal ... 1

1.2 Background ... 1

1.3 Requirements of the Test Tool Framework ... 3

1.4 Method ... 4

1.5 Result ... 4

1.6 Exam Thesis Disposition ... 5

2 Background ... 7

2.1 Introduction ... 7

2.2 Software Testing ... 7

2.2.1 Test Methods
2.2.2 Test Tools
2.2.3 Test Cases and Scripts
2.2.4 Test Environment
2.2.5 System Under Test (SUT)
2.2.6 Test Tool Framework versus Test Framework

2.3 Telecom Platforms ... 13

2.3.1 Ericsson's Telecom Platforms
2.3.2 AXE
2.3.3 The Telecom Server Platform (TSP)
2.3.4 The Connectivity Packet Platform (CPP)

2.4 Testing of Telecom Platform Software ... 23

2.4.1 Meeting with Mats Berglund, Testing Expert at Ericsson

2.5 Currently Used Test Beds and Test Tool Integrations ... 31

2.5.1 The Simulated Environment Architecture (SEA)
2.5.2 The Message Protocol Handler (MPH)
2.5.3 The CPP Emulator
2.5.4 Vega and MessageDriver

3 Test Tool Framework 39

3.1 Introduction.. 39

3.2 Background.. 39

3.3 An Integrated Test Environment 40

3.4 Test Tool Framework Requirements 42

3.4.1 Connection to the System Under Test (SUT)

3.4.2 Centralized Functions

4 Market Analysis 45

4.1 Introduction.. 45

4.2 Candidate Products 47

4.3 Comparison Points.. 48

4.4 Eclipse TPTP 50

4.4.1 Introduction

4.4.2 Functionality

4.4.3 Architecture

4.4.4 Eclipse Modeling Framework (EMF)

4.4.5 Standards

4.5 Software Testing Automation Framework (STAF) 55

4.5.1 Introduction

4.5.2 Functionality

4.5.3 Architecture

4.6 Eclipse TPTP versus STAF 58

4.7 Summary.. 59

5 Prototype.. 61

5.1 Introduction.. 61

5.1.1 Prototype Components

5.1.2 Deployment

5.2 Requirements 64

5.2.1 Use Case: Execute Expect Test Against the CPP Emulator

5.2.2 Use Case: Prepare Test

5.2.3 Use Case: Run Test

5.2.4 Use Case: Evaluate Test

5.3 Design.. 69

5.3.1 Introduction

5.3.2 Eclipse TPTP Design Overview

5.3.3 Scope of the Prototype

5.3.4 Prepare Test

5.3.5 Run Test

5.4 Improvements of the Prototype 82

5.4.1 Permissions of Remote Agents
5.4.2 Telnet Port Forwarding
5.4.3 Test Agent Implemented in C
5.4.4 Separate Launching Agent
5.4.5 Test Management Integration
5.4.6 Port to TPTP 4.x
5.4.7 SUT Configuration as a New Resource Type
5.4.8 Deployment of Test Scripts

6 Summary and Evaluation 87

6.1 Market Analysis 87

6.2 Prototype 87

6.3 Discussion 88
6.3.1 Pros and Cons of a Common Framework
6.3.2 Standardization
6.3.3 Open Source
6.3.4 Eclipse-Based Products

7 Conclusion 95

References 97

A Definitions 103

B Acronyms and Abbreviations 113

C Introduction to TTCN-3 119

D Market Analysis – Product Descriptions 123

D.1 Danet TTCN-3 Toolbox 123
D.1.1 Introduction
D.1.2 Functionality
D.1.3 Architecture

D.2 IBM Rational Testing Products 129
D.2.1 Introduction
D.2.2 IBM Rational TestManager

D.3 JUnit 131

D.4 OpenTTCN Tester for TTCN-3 132
D.4.1 Introduction
D.4.2 Functionality
D.4.3 Architecture

D.5 Scapa Test and Performance Platform 3.1 136

D.6 Telelogic TAU/Tester 137
D.6.1 Introduction
D.6.2 Functionality

D.6.3 Architecture

D.7 Testing Technologies TTworkbench 142
D.7.1 Introduction
D.7.2 Functionality
D.7.3 Architecture

E Market Analysis – Comparison Points 147

F Market Analysis – Product Evaluations 151

F.1 Ready-to-Use Products 151
F.1.1 Danet TTCN-3 Toolbox
F.1.2 Eclipse TPTP 3.2 (as a ready-to-use product)
F.1.3 IBM Rational Test Manager
F.1.4 JUnit
F.1.5 OpenTTCN Tester
F.1.6 Scapa Test and Performance Platform 3.1
F.1.7 Telelogic TAU/Tester
F.1.8 Testing Tech TTWorkbench

F.2 Frameworks 160
F.2.1 Eclipse TPTP 3.2 (as a framework)
F.2.2 STAF

G Prototype – User Manual 163

G.1 Introduction 163

G.2 Eclipse Vocabulary 164

G.3 Eclipse Pre-Defined Architecture of Resources 165

G.4 Prepare Test 166
G.4.1 Changing to the Test perspective
G.4.2 Creating the Project
G.4.3 Creating and Editing the TPTP Expect Test Suite Resource
G.4.4 Creating and Editing the Artifact Resource
G.4.5 Creating and Editing the Location Resource
G.4.6 Creating and Editing the Deployment Resource

G.5 Run Test 189

G.6 Evaluate Test 192
G.6.1 Test Execution Structure
G.6.2 Exporting the Test Execution Result

H Prototype – Installation Instruction 199

H.1 Introduction 199

H.2 Requirements 200

H.3 Installation of the Eclipse Plug-ins 201

H.4 Installation of the RAC Plug-in 203

List of Figures

Figure 1: Conceptual Model of Software Testing ... 8
Figure 2: Functional Testing ... 9
Figure 3: Test Methods and their Relations ... 11
Figure 4: Conceptual Model of a Test Tool Framework ... 13
Figure 5: The Three Layers of the Logical Network ... 14
Figure 6: Example of an AXE Based System ... 16
Figure 7: The TSP Architecture ... 18
Figure 8: Software Fault in TSP ... 19
Figure 9: CPP Fundamental Architecture ... 20
Figure 10: Examples of CPP Network Nodes in the WCDMA Application Area ... 22
Figure 11: Schematic View of a Test Case ... 25
Figure 12: A Use Case Example ... 26
Figure 13: Example System Architecture for an End-to-End Test Case ... 27
Figure 14: Launching Model with Pre-Defined States ... 29
Figure 15: SEA Architecture ... 32
Figure 16: MPH Communication Layer ... 33
Figure 17: Example Configurations Using MPH ... 34
Figure 18: MPH Protocol Packet Format ... 36
Figure 19: MPH Control Channel Data Format ... 36
Figure 20: The Real versus the Emulated CPP Environment ... 37
Figure 21: One-to-One Relationship between the Test Tool and the SUT. ... 39
Figure 22: Many-to-Many Relationship between the Test Tool and the SUT. ... 40
Figure 23: Users that Benefit from an Integrated Test Environment. ... 41

Figure 24: External Interfaces. 42
Figure 25: Centralized Functions. 44
Figure 26: TPTP Architecture Overview 53
Figure 27: STAF Architecture Overview 57
Figure 28: Remote Test Bed Launch 62
Figure 29: Prototype Components 62
Figure 30: Deployment of the Prototype Components 63
Figure 31: Target Environment Deployment 64
Figure 32: Prototype Use Case 65
Figure 33: TPTP Basic System Structure 69
Figure 34: Test Launch Interactions 71
Figure 35: Test Execution Components 72
Figure 36: Prototype plug-ins 73
Figure 37: Expect Test Suite New Wizard 74
Figure 38: The Tester Creates a New Test Suite 75
Figure 39: Test Suite Editor Classes 76
Figure 40: TPTP EMF Test Profile Model 77
Figure 41: Text Execution Components 78
Figure 42: The Tester Starts the Test 79
Figure 43: Test Agent 80
Figure 44: Test Bed Launch 81
Figure 45: One-to-Many Relationship between a Test Tool and SUTs. 87
Figure 46: Integration by Means of a Common Test Tool Framework. 88
Figure 47: Specialized Test Tool 90
Figure 48: The General Structure of a TTCN-3 Test System 120
Figure 49: Danet TTCN-3 Toolbox 124
Figure 50: An Architectural Overview over TTCN-3 Toolbox 127
Figure 51: JUnit Interfaces and Classes 132
Figure 52: The OpenTTCN Campaign Manager 133
Figure 53: Telelogic TAU/Tester showing a TTCN-3 Tutorial 138

Figure 54: The Architecture of TAU/Tester’s Executable Test Suite (ETS) 140
Figure 55: TTworkbench showing the Built-in Text Editor 144
Figure 56: The Prototypes’ Three Use Cases 163
Figure 57: The Main Window of Eclipse 164
Figure 58: The Eclipse TPTP Resource Architecture 166
Figure 59: Wizards in Eclipse 166
Figure 60: Select Perspective Dialog Window 167
Figure 61: Creating a Simple Project 168
Figure 62: Another Way to create a Project in Eclipse 169
Figure 63: Creating a New Test Artifact 170
Figure 64: Creating a New TPTP Expect Test Suite 171
Figure 65: The TPTP Expect Test Suite Editor 172
Figure 66: The CPP Emulator Configuration Tab 173
Figure 67: The Expect Test Cases Tab 174
Figure 68: Creating a New Artifact 175
Figure 69: The Artifact Editor 176
Figure 70: The Select Resource Dialog Window 177
Figure 71: The Test Assets Tab 178
Figure 72: Creating a New Location 179
Figure 73: The Location Editor 180
Figure 74: Creating a New Deployment 181
Figure 75: The Deployment Editor 182
Figure 76: The Pairs Tab 183
Figure 77: The Add Artifact Dialog Window 184
Figure 78: The Select Resource Dialog Window 184
Figure 79: The Add Location Dialog Window 185
Figure 80: The Select Resource Dialog Window 186
Figure 81: The Pairs Tab 187
Figure 82: The Overview Tab 188
Figure 83: The Test Navigator after Test Preparation 189

Figure 84: How to Open the Run Dialog Window 190
Figure 85: The Run Dialog Window.. 191
Figure 86: My Expect Test Suite Test Execution Resource 192
Figure 87: The My Expect Test Suite Test Execution Editor... 193
Figure 88: The Events Tab.. 194
Figure 89: The Events Tab, with a Collapsed View 195
Figure 90: WinZip Showing the “My Expect Test Suite.execution.zip” File................ 197
Figure 91: ConTEXT Showing the “ResourceContents” File 197
Figure 92: Plug-ins in the Eclipse TPTP Architecture... 199
Figure 93: The “About Eclipse Platform Plug-ins” Dialog Window............................. 201
Figure 94: The “TPTP Expect Test Suite” Wizard.. 202

List of Tables

Table 1: MPH Service Primitives 35
Table 2: Example STAF Services 56
Table 3: IBM Rational TestManager Test Script Types 130
Table 4: Comparison Points used in the Market Analysis 149

1 Introduction

1.1 Project Goal

The purpose of the project described in this thesis was to act as a first study to increase the knowledge about available test tool frameworks that may be used for creating an integrated test environment. The main objective was to give TietoEnator guidance in which test tool framework, product or technical solution, they should use in future projects. The goal was to find an existing product on the market, which was as complete as possible and ready to use with as small modifications as possible. The product should use standard, open techniques and preferably be open source.

Two main groups of users would benefit from a general test tool framework: testers and people developing and maintaining test tools. Testers would benefit from a simplified test preparation, execution and evaluation. Simplifying the test preparation, execution and evaluation is also important for the people developing the emulators and tools at TietoEnator, in order for them to be able to verify that the test tools work in the emulated environment.

1.2 Background

The Telecom R&D, Test Tools & Solutions department of TietoEnator in Karlstad develops support systems for different telecom platforms. The platforms are typically distributed systems with a network of cooperating nodes, which together make up the system functionality. The platforms are large and complex with a great amount of system software. Testing the system software is an essential part in the development process.

The support systems developed by TietoEnator include complete runtime environments consisting of emulators and tools for loading and executing the telecom platform software.

The runtime environments enable testing of the system software without access to the target hardware; testing can be performed on an ordinary PC instead. Since target hardware is often expensive and not widely available, the emulated environments are important. The emulated runtime environments increase the availability for testing and reduce the amount of necessary function testing in the target environment. Other benefits are increased determinism and better debugging possibilities.

The function tests are run from fully automated test suites. The test suites may be used for testing both in emulated and target environments. There are also manual, interactive tests that can be run, for example simulation of mobile phones.

An investigation of the need for test tools for one of the platforms, the Ericsson Telecom Server Platform (TSP) [4],[5], has been made by TietoEnator together with Ericsson. The investigation showed that different TSP subsystems have a similar need for test tools and have also developed their own test suites, with different solutions and techniques. For example, one of the subsystems has used JUnit [52], while another has based its tests on The Testing and Test Control Notation (TTCN) [48]. The result of the investigation showed the need for an integrated test environment and was one incentive for carrying out this project.

A common problem when performing function testing is the setup of the test environment. In End-to-End (E2E) testing, for example, all the nodes that take part in the function under test must be set up and connected. Each node implements one or more interfaces. In the test suites there are typically one separate tool for each interface. Thus, a test environment setup typically consists of a whole chain of different tools that must be connected to its respective node interface and run simultaneously during the test. Furthermore, there are often dependencies between nodes and components making it difficult to start-up or launch the test environment in an initial stable status. The setup of the test environment can be quite a complex task for the tester. The many different tools may also make it difficult to follow up the test execution.

The many dependencies between the test tools and the node interfaces may also make maintenance problematic. A change in the interface of a node may imply changes to a number of dependent test tools.

The main business goal with a test tool framework, from TietoEnator's point of view, is the possibility to use it as a basis for an integrated test environment for new platforms. It may also be possible to use it for integrating existing test tools, for example the different test suites for the TSP platform.

1.3 Requirements of the Test Tool Framework

The following are requirements that were specified for the test tool framework.

The test tool framework should be generic, allowing for interoperability with as many different test tools as possible. The test tool framework should also allow use of any test script language. The main target environment is telecom platforms, which means that testing of distributed systems must be supported. All different test methods used for testing telecom platform software must be supported; automatic functional regression testing must be supported as well as manual testing. Although telecom was the main focus, the test tool framework should be applicable in other technical areas as well.

The main technical requirement for a test tool framework is to find a general way of connecting the test tools to the nodes in the System Under Test (SUT). Ideally the whole platform, for example TSP, should be viewed as a single unit with one single access point for connecting test tools to the nodes to be tested.

The second technical requirement for a test tool framework is to centralize functions common to different test tools, as a base for building new test tools. The test tool framework could, for example, have support functions for logging and debugging. Another possibility is a common Graphical User Interface (GUI) that the test tools can use. The centralized functions would give a better integration of the different test tools, less duplicated functionality, a reduced number of tools, fewer dependencies, simplified maintenance, simplified use for the tester and a more uniform look and feel. The support functions should also use standardized techniques where possible.

1.4 Method

The scientific method used in this project was largely based on theoretical descriptions and comparisons of different available products. The project was divided into two phases: First, a market analysis of available products was carried out. The second phase was to build a prototype to further increase the knowledge about using a specific product for solving an actual problem.

During the project, the project group had the privilege to meet Mats Berglund, testing expert at Ericsson, to discuss the exam thesis and software testing in the telecom domain. Mats Berglund has contributed with valuable experience and has had the role as a reference person in the project.

1.5 Result

The market analysis resulted in a theoretical evaluation of different products for building an integrated test environment. The main conclusion from the market analysis was that the product that best fulfills the specified requirements of a test tool framework, see Section 1.3, is Eclipse. Eclipse is open source and is designed to be an open architecture that can be extended in many different ways. Eclipse provides quite many centralized functions for building different test tools, such as distributed test execution and a common GUI. Eclipse also uses standardized techniques such as Unified Modeling Language (UML) [36] and Extensible Markup Language (XML) [37].

A prototype was built as a proof of concept in order to show that Eclipse can be used as a basis for a general test tool framework. The prototype makes it possible for a tester to create a test configuration to be run on a remote machine. The test configuration includes a selection of test scripts, test bed configuration and SUT configuration. When the tester starts the test from the Eclipse client Workbench, the test bed is automatically launched on the remote machine, the test scripts are executed, and finally the test bed is torn down. The tester can view the test result in a test execution history in the Eclipse client Workbench.

The conclusion from the prototyping is that it should be possible to use Eclipse for creating an integrated test environment.

1.6 Exam Thesis Disposition

This thesis has the following disposition:

- Chapter 1 (this chapter) contains a summary of this thesis with purpose, background, method and result.
- Chapter 2 contains background information for the project. The concept of a test tool framework is described in the context of software testing in general and testing of telecom platforms in particular. Examples of current telecom platforms are given.
- Chapter 3 describes the concept of a test tool framework, as defined in this thesis.
- Chapter 4 contains the survey of the different products, with summary and conclusions.
- Chapter 5 describes the prototype that was implemented.
- Chapter 6 summarizes the results of the thesis.
- Chapter 7 contains a final conclusion for the project.
- Appendix A defines some concepts used in this thesis.
- Appendix B lists acronyms and abbreviations used in this thesis.
- Appendix C gives an introduction to TTCN-3.
- Appendix D contains descriptions of the different products studied in the market analysis.
- Appendix E contains descriptions of the points used to compare the different products in the market analysis.
- Appendix F contains evaluations for the different products studied in the market analysis.
- Appendix G is a User Manual for the prototype that was implemented.
- Appendix H is an Installation Instruction for the prototype that was implemented.

2 Background

2.1 Introduction

This chapter gives a background to the area for this project. Section 2.2 describes a test tool framework in the context of software testing in general and Section 2.3 gives an introduction to different telecom platforms. Section 2.4 describes some issues when testing telecom systems. Test beds and test tool integrations used at TietoEnator today are described in Section 2.5.

2.2 Software Testing

Software Testing is an essential part of Software Engineering. Traditionally, testing has been primarily seen as a quality improving activity. The main purpose of testing is to find defects and getting the defects corrected. If defects in the system are corrected in a systematic way the quality of the system will improve.

Historically, testing was mainly used as debugging to find out if software functioned as intended. The testing and debugging phase took place after the program was written. Testing was therefore considered as a follow on activity [13]. The terms testing and debugging were not clearly distinguished, but used as having the same meaning. Today software testing is considered as a separate process when developing software. The test phase is no longer seen as a follow on activity, but rather as an integrated process.

Testing methods have evolved rapidly in recent years and methods as Test Driven Development [18], [22] have become very popular. With Test Driven Development tests are created before the program is written. The tests are used as a reference for judging whether or not the system is correct; in the extreme case tests are even used as a replacement for software specifications.

Software systems are becoming more and more complex and extensive testing in all phases of the development is necessary to master the complexity.

Therefore, with increasing complexity in software systems and development methods based on tests, testing can no longer be seen only as a quality improving activity, but instead as a necessary activity in order to develop functioning systems.

Software testing involves many concepts. Figure 1 shows a brief conceptual model of software testing with the relationships between some of the concepts involved. Appendix A defines many important testing related concepts used in this thesis.

Figure 1: Conceptual Model of Software Testing

During the software testing phase in a software development project, the main objective is to define a set of tests that execute the System Under Test (SUT) in order to find defects. Software testing includes many test methods and test tools and is realized in a test environment. The test environment includes the test tools needed to conduct the tests and is also responsible for the correct configuration of the SUT.

A schematic model of functional testing is shown in Figure 2. A test generates stimuli that are sent to the interface of a SUT. In response to the stimuli, the test retrieves an output, interpreted as a test result generated by the SUT.

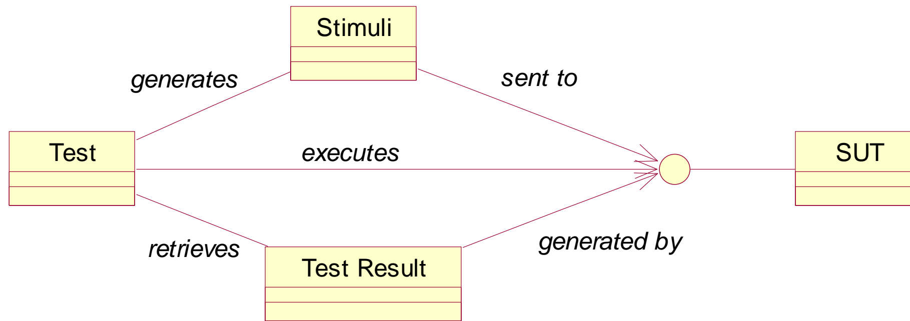


Figure 2: Functional Testing

2.2.1 Test Methods

The area of software testing includes a great number of methods. Testing can, for example, be fully or partly automated. There are a number of benefits with automated testing which give both lower cost and higher quality. Automatic tests can, for example, be run much more frequently than manual ones. One possibility is nightly smoke tests for each new build, in order to continuously regression test the system during development – something that would be impossible with manual testing. Automatic tests also enable much more test cases to be run compared to manual testing, giving a better coverage of the system being tested and thus more defects being found and corrected. Another advantage is that computers, unlike humans, do not miss a deviation from an expected result.

New processes for software development, for example lightweight processes such as Extreme Programming (XP) [16], are totally dependent on automatic testing. New test cases are created and run before new functions are added to the system in order to constantly keep the system on a sufficient quality level.

Even though automatic testing is preferable, there are often tests that cannot be automated. Test cases for starting up or shutting down a system or interaction with special hardware in a system must normally be manual.

Two major concepts in software testing are black box and white box testing. With black box testing, the test case selection is based on an analysis of the specification of the component without reference to its internal workings [24]. The SUT is tested by giving stimuli to and checking responses from the external interfaces of the SUT only. Testing based on an analysis of internal workings and structure of the components is called white box testing, glass box testing or structural testing [12]. White box testing includes techniques such as branch testing and path testing.

Figure 3 shows different test methods and their relations, as well as relations to functional and non-functional requirements. A test can be either black box or white box. Black box tests can be divided into functional tests and non-functional tests. Functional tests can be divided into system external interface test and unit test. Non-functional tests can be divided into load/performance and stress tests. Unit test, also called module test or basic test, tests a single unit or small cluster of coherent units. Unit test is sometimes seen as a white box test. There are a number of different types of system external interface tests, among others: protocol tests, Graphical User Interface (GUI) tests, Command Line Interface (CLI) tests, Operation and Maintenance (O&M) tests and terminal or End-to-End (E2E) tests.

Functional requirements can often be described by different use cases and are tested by means of system external interface tests. Non-functional requirements are tested with corresponding non-functional tests, such as load or stress tests.

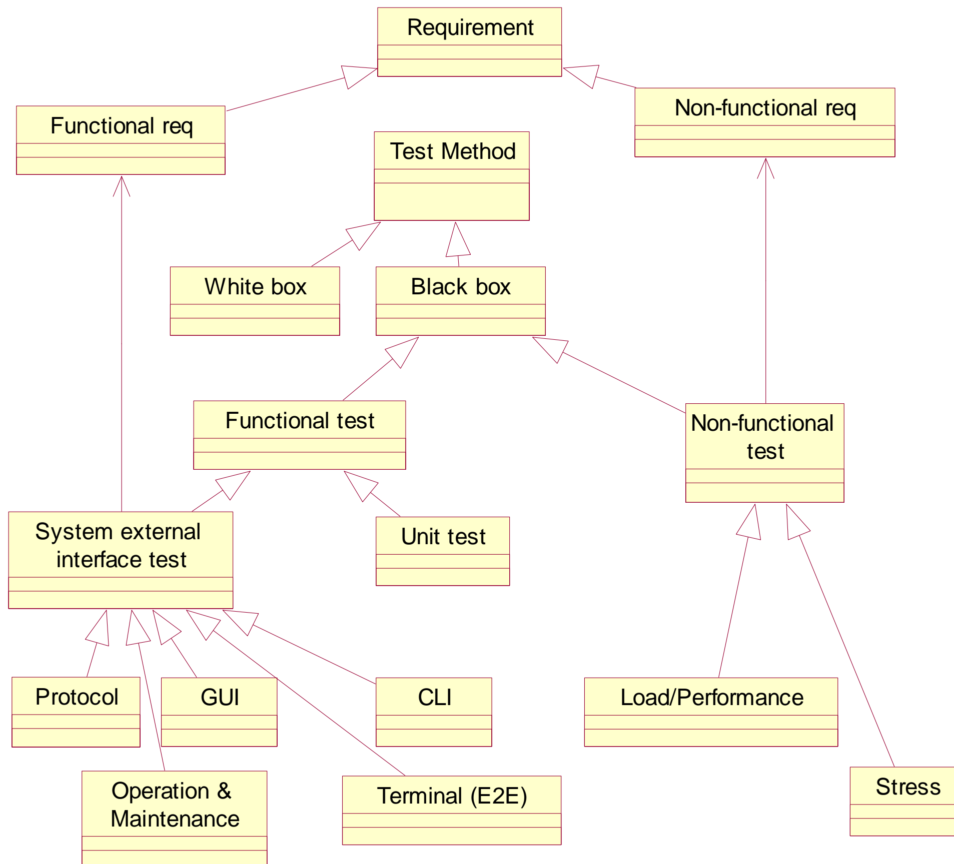


Figure 3: Test Methods and their Relations

2.2.2 Test Tools

The tools for software testing should support the methods used. There are tools for a number of purposes, for example test drivers, comparators and tools for creating stubs.

2.2.3 Test Cases and Scripts

Test cases are documented in test procedures. The test procedures are automated by means of test scripts and run by a test tool, typically a dedicated test driver. Test scripts may be written in an ordinary programming language, such as C++ or Java, or in a script language.

There are also special, standardized languages for creating test scripts as, for example, TTCN-3 [48]. An introduction to TTCN-3 is given in Appendix C.

2.2.4 Test Environment

To be able to run test cases, a test environment has to be created. An environment containing the hardware, the instrumentation, the simulators, the software tools, and other support elements needed to conduct a test is called a test bed [24].

2.2.5 System Under Test (SUT)

Before running a test, the test bed has to be configured for the system being tested – the System Under Test (SUT). The SUT is normally composed of a number of components. The Implementation Under Test (IUT) is the actual components within the SUT that are the target test objects for the current test.

2.2.6 Test Tool Framework versus Test Framework

The purpose with a framework in general is to make it easy for software developers to add new functionality. The purpose with a test tool framework is thus to make it easy to add new test tools to the test environment. The concept of adding new test tools has two different meanings in this project:

1. It should be easy to connect an existing test tool to the SUT, in order to let the test tool access and communicate with the target components.
2. It should be easy to create new test tools by building on functions implemented in the test tool framework. In this sense the framework has the role of a Software Development Kit (SDK).

A test framework should make it easy to add new test cases. A test framework may be built by means of a test tool framework and different tools. Figure 4 shows a conceptual model of a test tool framework.

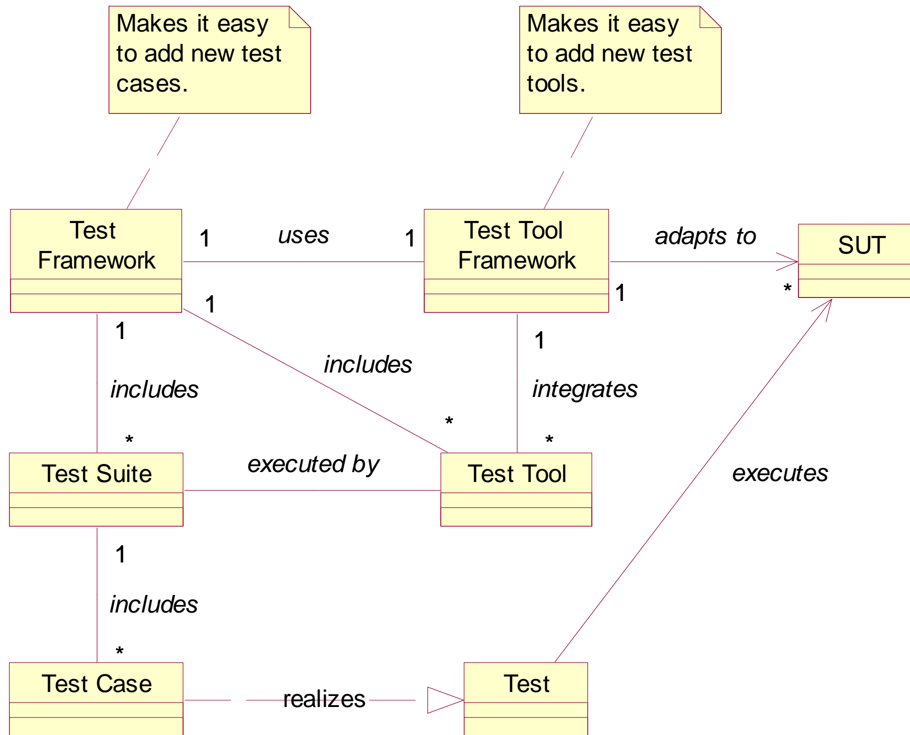


Figure 4: Conceptual Model of a Test Tool Framework

2.3 Telecom Platforms

This section gives examples of different Telecom Systems. Three platforms from Ericsson are presented: AXE, TSP and CPP. The purpose is to give an understanding of the basic functionality of these systems and also to describe the complexity of the SUTs in the telecom domain. When performing software testing, understanding of the SUT is essential.

2.3.1 Ericsson's Telecom Platforms

Today users require the possibility to always communicate, and their demand on telecom applications and services constantly increases. To be able to keep up, the Telecom Platforms have to evolve continuously. In the telecom world today, the trend is towards a convergence

of telephony, media and data communication. Ericsson's answer to this development [5] is to use a logical network divided into three horizontal layers, see Figure 5. Within the top layer *content and user applications layer*, server applications, databases and services are provided. The middle layer *communications control layer*, is responsible for control functionality. Finally, the bottom layer *connectivity layer* assures that the transport of all data is performed in an appropriate way.

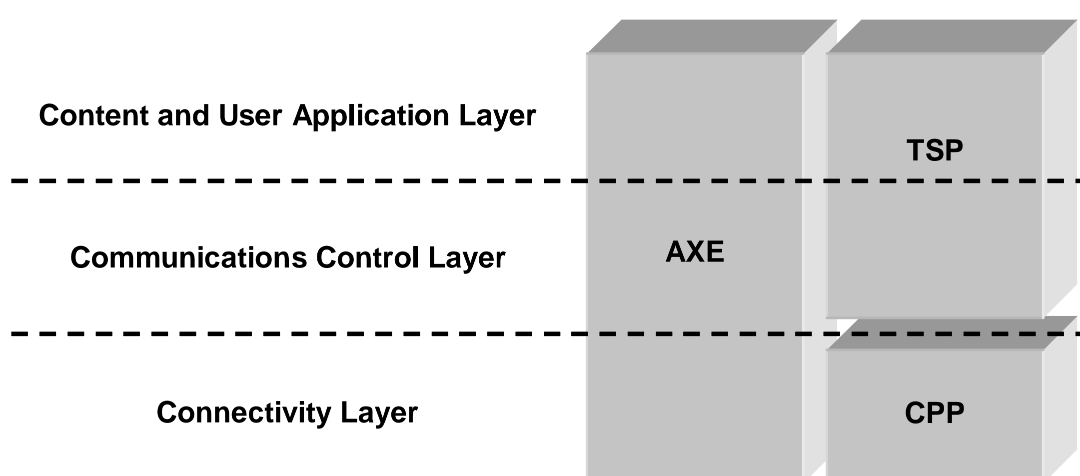


Figure 5: The Three Layers of the Logical Network

To be able to provide telephony, server and access applications, Ericsson's plan [5] is to extend the existing AXE telecom platform with two additional platforms: the Telecom Server Platform (TSP) for servers and the Connectivity Packet Platform (CPP) for gateways. Both platforms were created with the fundamental requirements of high availability. Users of telecom applications are generally used to higher availability, compared to users of data communication applications (it is more likely that your internet connection will be down than that your phone does not get a dial tone). The architecture has been created using a server-gateway split meaning that the applications and control functionality have been separated from the connectivity and transport. Another important approach when extending the architecture, according to Ericsson [5], is to use common system components and the same

building practice for AXE, TSP and CPP. Common key words for all platforms are high availability, real-time performance, scalable capacity and openness.

2.3.2 AXE

The AXE system [5] has existed for a long time and today AXE equipment can be found all over the world. As many new Internet based telecom services require packet data transports, there is a need for integration of packet data transports and legacy telecom circuit switching. AXE provides a base for merging circuit switching and packet data transport networks. AXE is not only the basis for legacy telephony applications such as Public Switched Telephone Networks (PSTN), but also used for mobile telephony. The AXE 810 version incorporates commercially standardized components for mobile telephony and it mainly serves as, see Figure 6:

- The Mobile Switching Centre (MSC) is a part of a Global System for Mobile Communication (GSM) network and its function can be compared to the exchange in a fixed network, plus everything extra needed to handle Mobile Stations (MS). MS collectively refers to all the devices communicating over the mobile phone network. The MSC controls the Mobile Stations and functions as authentication, location management, handovers, registration and the routing of the calls.
- Base Station Subsystem/System (BSS), which is a segment of the GSM system consisting of a Base Station Controller (BSC) and one or more Base Transceiver Stations (BTS) that are associated with it. The BSC manages the BTS (one or more), and the BSS itself is controlled by the MSC, which controls several BSS. The BSS is the interface between the MS and the MSC.

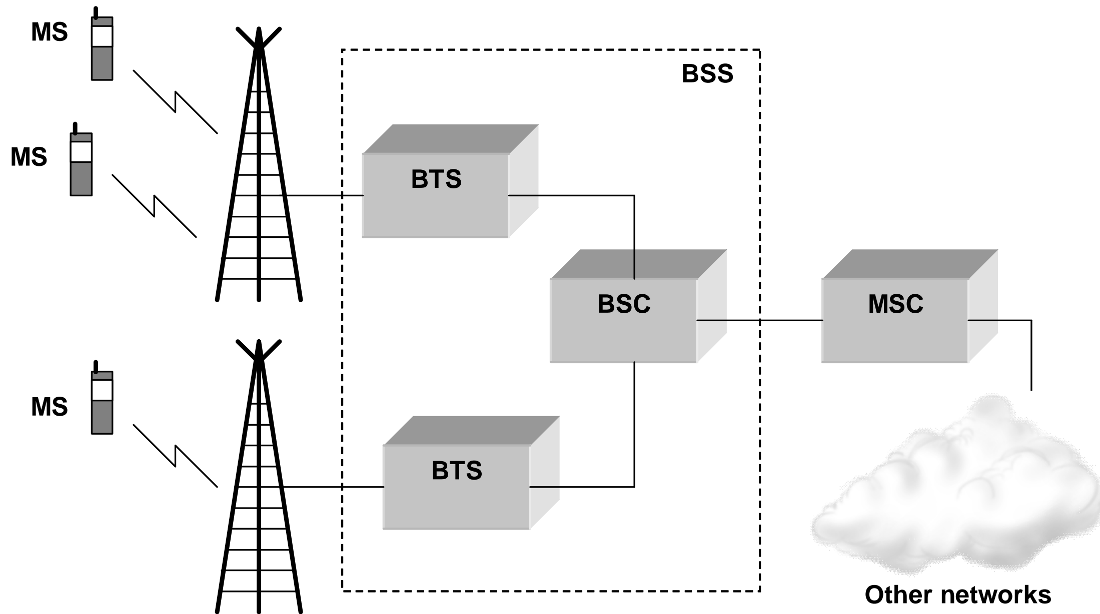


Figure 6: Example of an AXE Based System

To be able to get an open architecture of the AXE system, Ericsson uses commercially available hardware components, standard hardware building practices and commercially available software components and interfaces. Due to the open architecture approach, AXE hardware has been reduced drastically in size, making it much easier to work with. Even though the AXE is being constantly upgraded and developed it is still fully backward compatible.

2.3.3 The Telecom Server Platform (TSP)

The Telecom Server Platform (TSP) [4] is a robust and fault-tolerant platform based on open server technology. It is built to support new multimedia applications and control functionality. Considering the high availability requirements of the telecom users and that TSP is designed for server application purposes, TSP has key features such as high reliability, scalable capacity and real-time operation (minimal delay).

The architecture of TSP consists of both hardware and software, see Figure 7. The hardware used is several boards with off-the-shelf Intel CPUs connected to each other via an Ethernet network. The software running on top of the CPUs are two Operating Systems (OS), one Linux based and an OS called DICOS, developed by Ericsson and optimized for real-time processing. DICOS is based on queuing technologies and offers soft real-time response.

On top of the DICOS and Linux OS is a Clusterware called Telecommunications Object Request Broker (TelORB). TelORB handles network communication, database operations and effectively runs executing software on available nodes. TelORB distributes all processes redundantly (runs each process on at least two CPUs). By doing this, high availability and reliability is achieved. The TelORB makes all application processes transparent to the application. Since the application processes are transparent, the application does not know where its processes run.

On top of the TelORB relies Node Management and Signaling, see Figure 7. The Node Management is used to manage the TSP node. Node Management is based on Telecommunications Management Network (TMN) and incorporates standards such as Common Object Request Broker Architecture (CORBA) [14], Lightweight Directory Access Protocol (LDAP) [17], Hypertext Transfer Protocol (HTTP) and Simple Network Management Protocol (SNMP) [9]. The Node Management has support for several functions such as the following: Fault Management (FM), Configuration Management (CM), Performance Management (PM), provisioning support, logging and license management. The signaling part of the TSP architecture handles the Signaling System 7 (SS7) [20] and the Internet Protocol (IP) stacks. Currently SS7 is still the one most commonly used in telecom networks. The trend today, however, is that IP is becoming more and more popular due to new services and applications using IP.

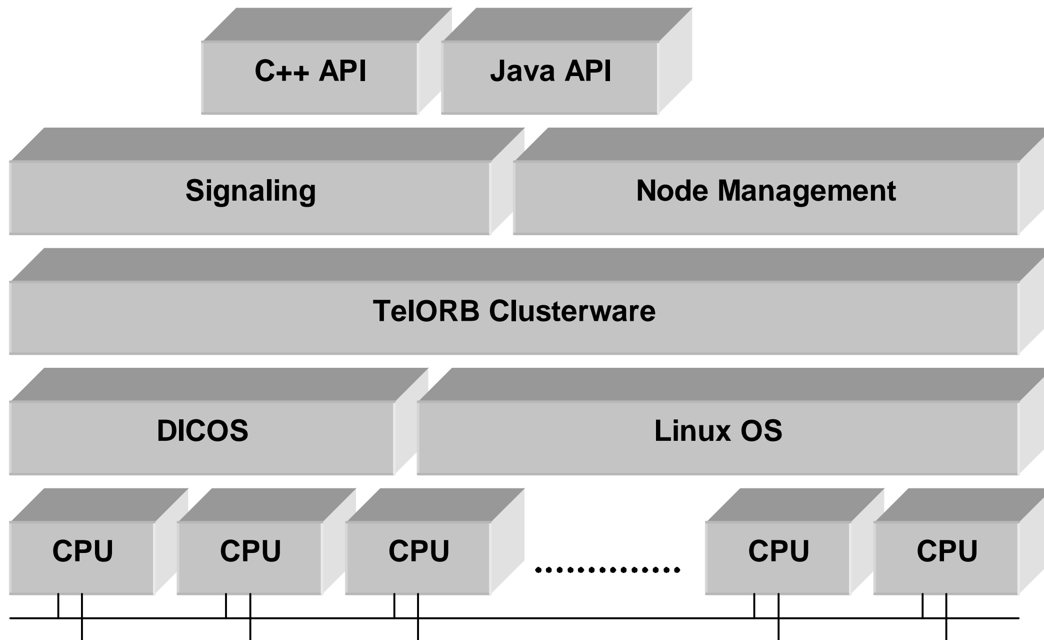


Figure 7: The TSP Architecture

High availability can be achieved with fault-tolerant hardware, but such hardware is expensive and will not give full control. Ericsson has therefore chosen to use the software solution TelORB to eliminate faults. Using software instead of hardware is a more cost-effective solution since standard off-the-shelf hardware can be used. To be able to guarantee high availability TelORB uses replication. By using replication, all the processes run on at least two nodes, meaning that all the nodes do not have to be available all the time. Figure 8 shows an example how this redundancy caused by the distribution works in case of a software failure. In Figure 8A, a working system is shown. In Figure 8B, a software fault occurs in a node, and the processes and the data are lost. Due to TelORB and its distribution, the processes and data are replicated (redundant) and can be found in the other nodes, see Figure 8C. When the node where the software fault occurred has restarted, the processes and data can be copied from the other nodes, see Figure 8D, and the system is back to a working system again.

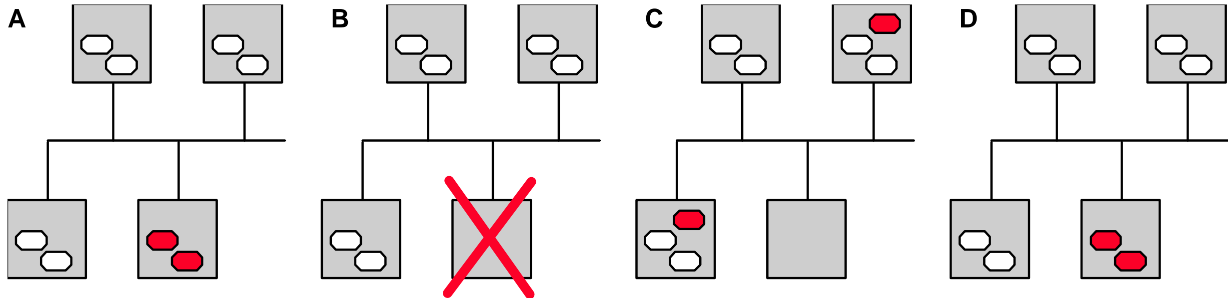


Figure 8: Software Fault in TSP

High availability is also provided in TSP by the opportunity to update while the system is still online. Since TSP is built on several standard off-the-shelf CPUs, upgrading can be done linearly by just adding more CPUs. By using the same concept as TSP does when software faults occurs, i.e. moving the processes and data to a working node in case of failure or downtime of a node, upgrading can easily be done without taking the system down. According to Ericsson [4] the system can even be moved to new hardware without downtime, by using the TelORB distribution concept.

The applications and services that TSP provides must be very robust and easily expandable. They share central databases containing essential user, traffic and charging data. Examples of TSP applications and services are:

- Home Location Register (HLR), which contains information (databases) about the subscriber for billing purposes and information about where the user is located.
- Media Gateway Control Function (MGCF), which provides signaling interoperability between IP and PSTN domains.
- Authentication, Authorization and Accounting (AAA) servers, which remotely control users' network access by requiring identification from them.
- General Packet Radio Service (GPRS) Support Node (GSN).

2.3.4 The Connectivity Packet Platform (CPP)

The Connectivity Packet Platform (CPP) [6] is used as Asynchronous Transfer Mode (ATM) and IP transport solutions for access networks. CPP is based on packet-switching technology and Ericsson introduced CPP for use with the third generation of radio access. The packet-switching technologies supported by CPP are TDM (Time Division Multiplexing), ATM and IP traffic. QoS (Quality of Service) can be achieved as well. The first CPP applications used ATM switching only, but IP has been introduced to enable access-networks products to switch between ATM and IP traffic.

A CPP node consists of two parts, an application part and a platform part, see Figure 9. The application is customized to handle the software and hardware specific for the application the CPP node is used for. The platform part can be divided into five subparts: CADE, Core, IP&C, SS7 and O&M.

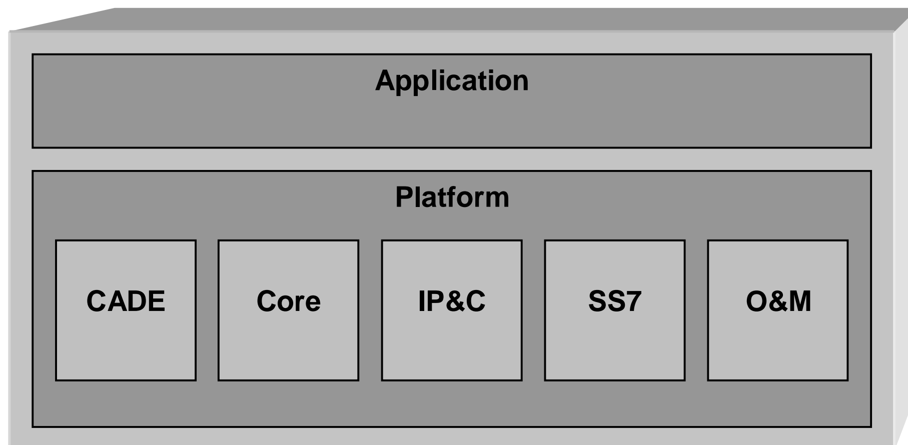


Figure 9: CPP Fundamental Architecture

The CPP Application Development Environment (CADE) is a software development environment for both application and CPP software. The CPP Core provides core functionality for the applications such as software execution via OSE (the operating system used), Java execution, system upgrades during operation, fundamental configuration and start/restart functions. The Internet Protocol and Connectivity (IP&C) provides the transport

service for both ATM and IP and network synchronization. The Signaling System number 7 (SS7) is used to send signaling messages between the CPP nodes in a network. Finally, the Operation and Maintenance (O&M) provides services to support management services and applications.

The hardware of CPP consists of several magazines (a kind of chassis) equipped with different types of circuit boards. Roughly, the CPP hardware consists of switch- and processor boards. The CPP switchboards handle user and control data in the node as well as between nodes. The CPP processor boards are used to perform a variety of tasks. Each CPP node has at least one processor board, the General Processor Board (GPB). The GPB functions as a central control and resource handler and also provides management services.

As in the case of TSP, robustness and fault-tolerance are achieved through processors working together in the CPP (distribution and replication of the executable software). When designing the architecture of CPP, the aspect that essential functions should survive hardware faults was in mind [6]. The switch, power supply, internal links and signaling links between the nodes are all redundant. Scalability was also important when designing CPP, since CPP was going to be used as a variety of applications. The result is that CPP can be used for a wide range of node setups, from a small node only handling one radio channel to a large node consisting of 30 or more magazines.

The applications based on CPP are most often found in Wideband Code Division Multiple Access (WCDMA) [7] and Code Division Multiple Access 2000 (CDMA2000) [7]. Another application area for CPP-based nodes is the Telephony Access Gateway (TAG), which creates access from an IP network to a circuit switched network.

WCDMA is a technology, partly developed by Ericsson, for wideband digital radio communications capacity-demanding applications, such as Internet services, streaming multimedia and videoconference. WCDMA has been selected for the third generation of mobile telephone systems in Europe, Japan and the United States. CPP-based applications in the WCDMA area are: the Media Gateway (MGW), the Radio Base Station (RBS), the Radio Network Controller (RNC) and the Radio access network aggregator / IP Router (RXI).

CDMA2000 is also a telecommunications standard adapted for the third generation of mobile telephone systems. CDMA2000 allows higher data transmission rates than the predecessor CDMA and is a competitor to WCDMA. CPP-based applications in the CDMA2000 area are: the Base Station Controller (BSC), the Home Agent (HA), the Packet Data Serving Node (PDSN), the Radio Base Station (RBS) and the WLAN Serving Node (WSN).

Figure 10 shows three CPP nodes with different CPP-based applications. The Radio Base Station (RBS) CPP node is responsible for transmission and reception in one or more cells to and from the User Equipment (UE). One example of a UE is a cellular phone. The Radio Network Controller (RNC) CPP node controls the use and integrity of radio resources. The CPP Media Gateway (MGW) connects the Mobile Core Network (MCN) with other networks such as GSM Radio Access Networks, Public Switched Telephone Network (PSTN) or other mobile networks.

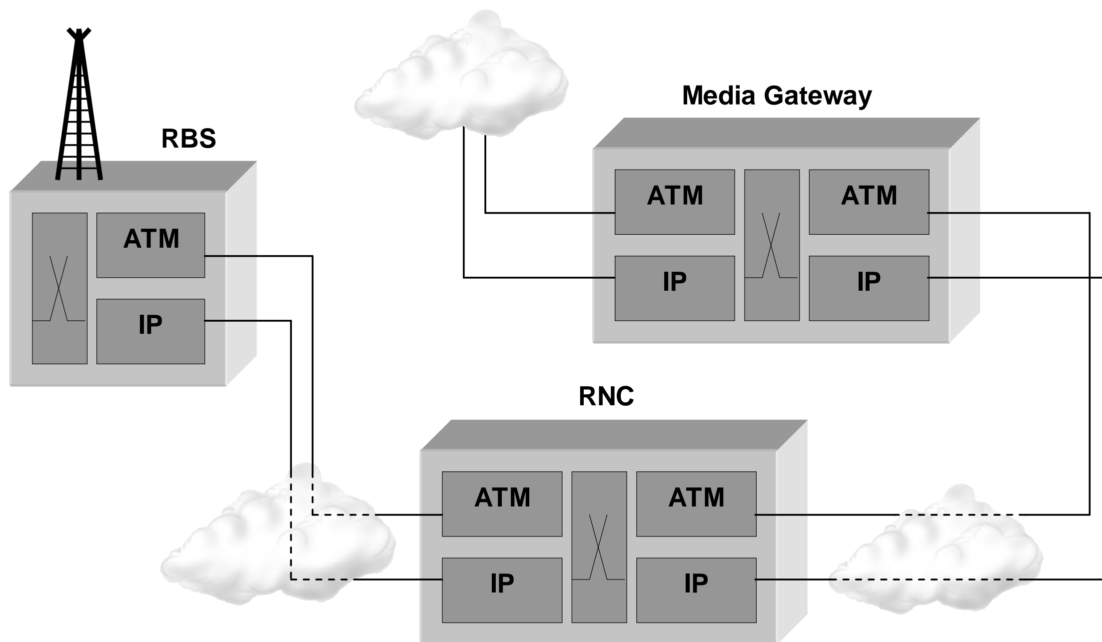


Figure 10: Examples of CPP Network Nodes in the WCDMA Application Area

2.4 Testing of Telecom Platform Software

Function testing is an essential activity in most software development projects. This is especially true when developing telecom platforms which typically are very large and complex systems. The quality requirements for telecom platforms are also very high, demanding extreme availability, among others. Furthermore, there are many functional requirements as well as real-time and other performance demands that must be met. The platforms are also essential parts of the communications infrastructure of the society and affect a large number of people. Therefore, testing these platforms is especially important.

At the same time, testing telecom platforms may be more difficult than testing other systems. Telecom platforms are typically distributed systems with a number of cooperating nodes, making testing a complex activity. Compared to a desktop system running on a single PC, for example, a distributed system is much more difficult to test. With a desktop system, the whole test bed with SUT is run locally, with the full control of the test execution on a single computer. With a distributed system, on the other hand, the test bed with possible use of different simulators and emulators, as well as the SUT, is run on a number of different nodes. The test tools must thus control remote computers and processes. There is typically also a great deal of internal communication between the nodes in a distributed system, which must run at the same time as different stimuli are given to the SUT when running different test cases. The communication between the nodes may, for example, include distributed functionality, synchronizing or keep-alive heart beats. The clustering and fail-over functionality needed to meet the high availability requirements also imposes additional complexity with a great amount of internal signaling, synchronizing and additional layering of the system software.

Function testing telecom platform software may be performed in different kinds of environments – in different test beds. Traditionally, function tests have primarily been run in target environments, but with the introduction of simulators and emulators function tests may also be run in simulator-based environments. Different strategies may be used. The same

function tests may be run both in a simulator based environment and on target, in which case many defects can be found and corrected before testing on target starts, thus saving both time and money. Another strategy is to run function tests in simulator based environments only, thus completely replacing the corresponding function testing in the target environment.

With open hardware architectures such as CPP and TSP, a test bed can also be assembled from standard computer components. With such a solution no simulators or emulators are used; instead function tests are run on alternative, low cost hardware. One example is the PCBox solution for TSP, in which standard barebone PCs are used for running DICOS, Linux and the different software systems.

New telecom platforms such as CPP and TSP are distributed systems consisting of a number of nodes and sub-systems. Thus a test bed may be configured by using a mix of real and simulator based environments.

2.4.1 Meeting with Mats Berglund, Testing Expert at Ericsson

At the start of this project the project group had a meeting [1] with Mats Berglund, testing expert at Ericsson. The purpose of the meeting was to discuss the exam thesis project as well as automated testing of telecom systems in general. Mats Berglund described many of the issues involved when testing telecom systems, which was very important for us to get a deeper understanding of the subject. A few, brief, extracts from the descriptions given by Mats Berglund are described in this section, in order to give a more complete picture of the complexity when testing telecom platform software.

Mats Berglund confirmed that testing solutions that enable automatization and re-use have been successful at Ericsson. Testing solutions that give both low cost and high availability are valuable; therefore, simulators, emulators and test automatization are interesting areas. Mats Berglund also described the software testing domain in general as very large and non-standardized. A common problem is that infrastructure is re-invented in many different testing products and solutions. According to Mats Berglund there is a need to unify and standardize testing concepts among different organizations and tool vendors. Another aspect is that it is

normally difficult to replace existing testing solutions altogether; instead it is important to build adaptable systems that enable interoperability and integration.

Mats Berglund also told us that there are quite a few testing products that are focused on desktop testing, that is, testing a program run on the local machine; less common are products which focuses on distributed systems. There are also many different scenarios or test use cases that must be supported. Mats Berglund told us that testing tools at Ericsson can be divided into four groups:

- Operation & Maintenance (O&M) Test Tools
- Protocol and Load Test Tools
- Terminal Test Tools
- GUI Test Tools

Besides automated testing, manual testing must be supported. According to Mats Berglund, manual testing should be seen as a special case of automated testing. The only difference between a manual test case and an automated one is that the manual test case requires human interaction at one or more stages; all other test management should be exactly the same.

Mats Berglund also described what the concept of a test case may mean when testing telecom systems. At an abstract level a test case may be described by the schematic view in Figure 11.

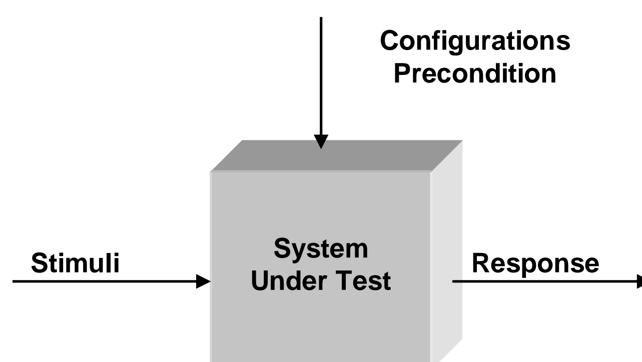


Figure 11: Schematic View of a Test Case

When function testing, the SUT is viewed as a black box. A test case sends stimuli to the SUT and checks if the response matches the expected result. Before the test case can be run, all configurations must be set up correctly and the SUT must be in the state required by the precondition of the test case. An experience shared by Mats Berglund is that it is important to distinguish between stimuli given to the SUT and internal communication; that is, communication between the nodes in the SUT. The configuration handling may be quite a complex task. A test case configuration includes data, nodes as well as connections between nodes. The test bed, possibly including a number of simulators and tools, must be correctly configured, and the SUT must, of course, be of the correct version. According to Mats Berglund, the number of combinations of all components to configure may be “as many as the stars in the Universe”.

It is important to emphasize that a test case in the telecommunications area can be very different from a test case used when testing desktop software. A test case when testing desktop software could, for example, be to test a function easily executed in an average PC environment. However, in telecommunications a test case often involves a complex network of hardware, often including simulators because of economical aspects. An example of a test case in telecommunications, which Mats Berglund described, could be the use case shown in Figure 12; a caller who initiates a phone call to a callee.

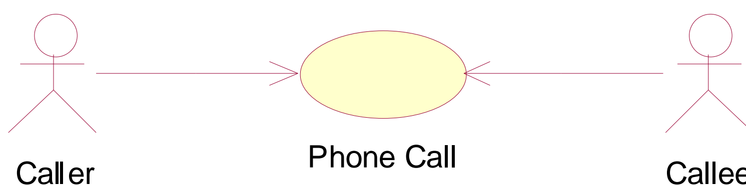


Figure 12: A Use Case Example

The use case may sound simple, but it is not so simple to test in a complex telecommunication architecture consisting of a number of different hardware components. Figure 13 shows an example system architecture needed to perform the test case in Figure 12.

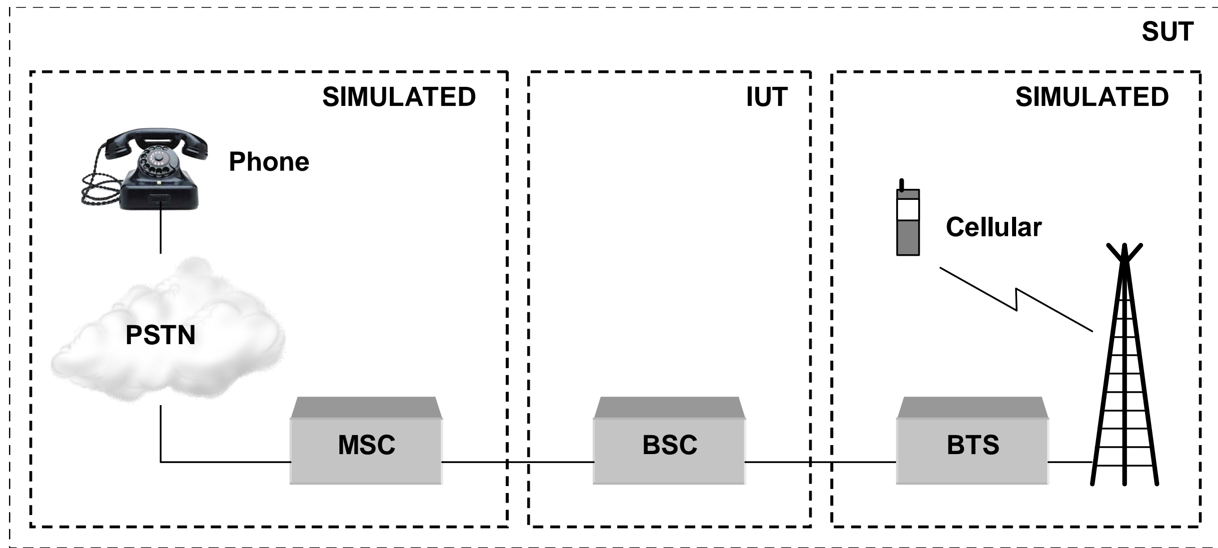


Figure 13: Example System Architecture for an End-to-End Test Case

A caller uses a phone connected to the Public Switched Telephone Network (PSTN) to dial a callee who has a cellular phone registered to a Base Transceiver Station (BTS). The SUT in this case is a PSTN phone, a Mobile Switching Centre (MSC), a Base Station Controller (BSC), a BTS and a Mobile Station (MS) (in this case a GSM cellular). The BSC is the actual Implementation Under Test (IUT); the specific part of the system (SUT) that is being tested.

We also discussed the problem of finding a general way of connecting different test tools to the SUT, since this is an important issue for this project. Mats Berglund explained that the problem of connecting different tools is part of a bigger concept called launching. The purpose of launching is to bring the SUT into a status that allows for testing to start, and also to tear down the test setup in a controlled way after testing has completed. Launching may therefore include all of the following steps:

1. Start up all nodes

2. Connect nodes
3. Assure that all nodes contain correct data
4. Test execution
5. Tear down

Start up and connection of nodes must be done in two steps because of dependencies between nodes. To be able to start up the system, a node A may be dependent on a node B to be up and running, and vice versa. A trick to solve this problem is to create a stub for the interface of node B, start the stub, and then start node A. Node A can then be connected to the real node B in step 2. According to Mats Berglund, step 2 – Connect nodes – is often confused with launching, but launching contains much more.

Step 3 is to check that all nodes contain correct data, a sanity check. Since the telecom systems under test often contain big volumes of data, the sanity check can be expensive. A sanity check may be optimized by only checking critical points. There is often a hierarchical organization of data and it is therefore sufficient to check certain points to gain confidence that all data are correct.

Mats Berglund also explained that a good approach would be to also let the launching mechanism maintain the SUT in predefined, controlled states, or levels, during the whole test execution. The different levels should be defined according to data definition levels. The idea is that each test case should be started from a predefined, stable, state and also return the system in a stable state after completion. There should be a few number of predefined states, in which the SUT would be allowed to be in between test cases, for example as in Figure 14.

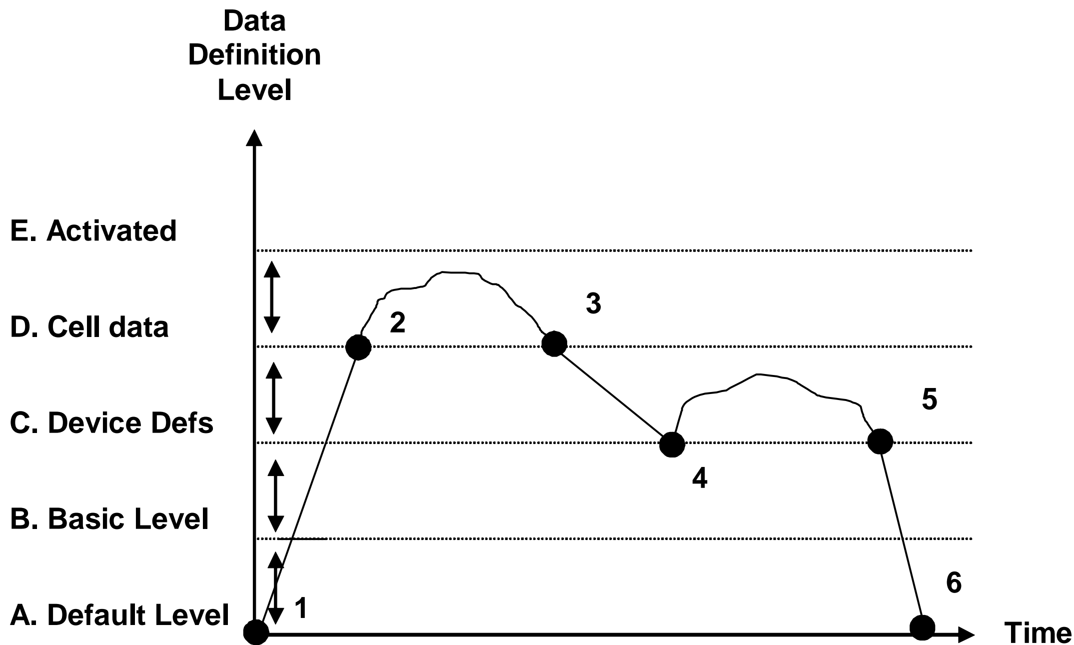


Figure 14: Launching Model with Pre-Defined States

In this conceptual view there are five different stable states that the SUT may be in between test cases. The launcher mechanism is responsible for setting up the SUT in an initial stable state. During test execution, each test case is responsible for setting the SUT in the state according to its precondition. If the SUT is not in the correct state, the test case orders the launching mechanism to change to the required state. The launcher supplies functions for changing levels. Before a test case finishes, the test case is also responsible for leaving the system in a stable state. If a test case stops before completion, the launching mechanism should set the system in a stable state instead. Figure 14 shows the following scenario, in which two test cases are run:

1. The launcher has run the sanity check and the SUT is in the default level. The first test case starts and orders the launcher to set the SUT in state D, according to its precondition.
2. The first test case runs and the data definition varies between level D and E.

3. The first test case ends and leaves the SUT in state D.
4. The second test case starts and brings the system to level C, by means of the launcher.
5. The second test case ends and leaves the SUT in state C.
6. After the last test case, the launcher takes the SUT to the default level.

The big advantage with the launching model described is that it enables test cases to be run completely independent of other test cases. A test suite can be set up with any collection of test cases, and the test cases can be run in any order. For optimization reasons, however, test cases on the same level may be grouped together. The normal situation when testing is that there is no launching mechanism as the one described, but test cases are instead dependent on other test cases and can only be run in a specific order. If a test case ends pre-maturely, the whole test suite ends. With the launching model described, the test suite will continue to run even if one or more test cases fail to complete.

In addition to launching issues, there are many other problems to solve in order to build a complete, automated test management system. Before launching the SUT, for example, there are other tasks that must be completed. An example that Mats Berglund gave was that the desktop from which the tester is controlling and running the test must be setup. This is called desktop launch, and may include the start of a workbench and different log viewers. Resources used by the test cases must also be booked and allocated before the test can start. This normally implies database support and additional control software.

Another important area is traceability between test cases, requirements and default reports, which normally imply a central database. Ericsson has developed a complete test management solution called Test Harness with connection to a central database for handling of all kind of resources and artifacts.

2.5 Currently Used Test Beds and Test Tool Integrations

This section describes a few test beds and test tool integrations used at TietoEnator and Ericsson today.

2.5.1 The Simulated Environment Architecture (SEA)

Simulated Environment Architecture (SEA) [19] is a system developed by Ericsson, originally for testing AXE system software. The SEA system includes a complete runtime environment consisting of emulators and tools for loading and executing the AXE system software. The SEA system enables testing of the system software without access to the target AXE hardware; testing can be performed on an ordinary PC instead. Since target hardware is a scarce resource, the emulated environment is important. The emulated runtime environment increases the availability for testing and reduces the amount of necessary function testing in the target environment. Other benefits are increased determinism and better debugging possibilities.

SEA is a scalable architecture for test environments and products that are emulator based. The architecture is based on the concept that by using well-defined interfaces and services, independent components can easily cooperate. To be able to cooperate, each component has a unique identity. By letting the users of SEA adding their own components to implement existing or new interfaces, SEA becomes a flexible environment for testing purposes. SEAs component-oriented architecture is based on Microsoft's Component Object Model (COM) [15], which will make SEA support future products implementing COM.

The kernel of SEA uses a layer-based model, see Figure 15. Each layer consists of several components working together. SEA can be divided into three core layers:

- osCore
- simCore
- appCore (AXE, CPP, TSP)

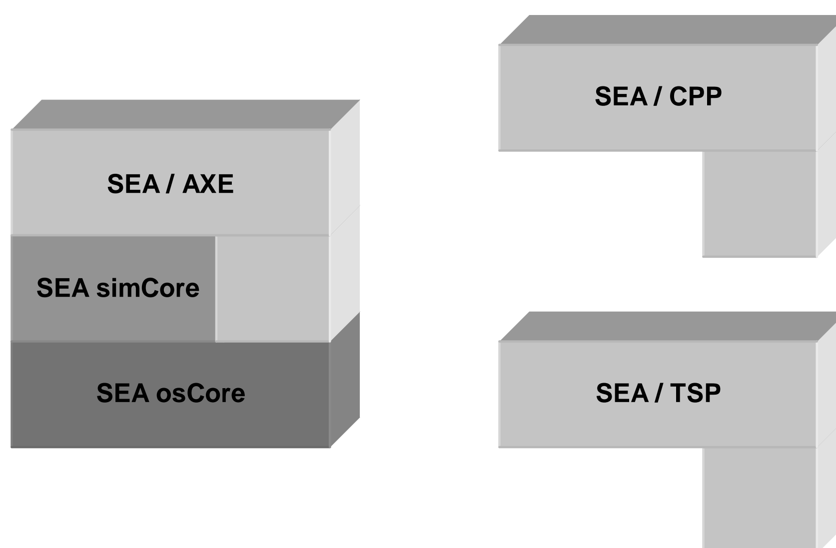


Figure 15: SEA Architecture

The osCore layer can be considered as the foundation or the "operating system" of SEA. It is the inner or bottom layer of SEA and acts as a virtual operating system. It hides and encapsulates properties and semantic differences between host operating systems. Another task of SEA is to bring up the kernel in a mode that makes it configurable to the components.

The simCore layer consists of generic and basic components which are needed by the simulation components. These generic components are:

- HTTP server component – to interact with the components in the kernel
- TCL interpreter component – to provide a script environment
- Scheduler component – to handle threads

Finally, the appCore can be seen as the simulated telecommunication platform (consisting of a number of basic components specific for the system that is going to be simulated). Using a layer approach makes it easy to change the appCore part, depending on which telecom platform is to be used when performing the tests.

2.5.2 The Message Protocol Handler (MPH)

The Message Protocol Handler (MPH) is a component of the SEA system which provides a service for establishing connections and communication channels between internal SEA entities and external entities. An example of an internal SEA entity is an AXE system component loaded into the SEA runtime environment. Examples of external entities are external test tools or other SEA processes.

The main purpose of the MPH component is to provide one single, central connection point, which external entities can connect and communicate through. By this connection point different test tools can establish communication channels with components in the SUT, for example AXE components loaded into the SEA environment.

2.5.2.1 MPH Design Solution

The MPH component in SEA is designed as a separate data communications layer, like the layers according to the standard OSI model [10].

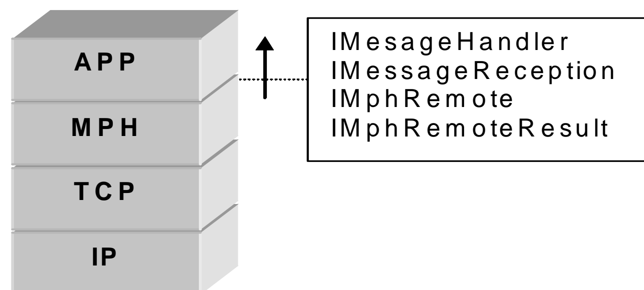


Figure 16: MPH Communication Layer

The MPH layer consists of:

- A service provided to the upper application layer (App in Figure 16), defined as a set of primitives in the four interfaces: IMessageHandler, IMessageReception, IMphRemote and IMphRemoteResult.

- An MPH protocol implementing the service.

The MPH service supports two different configurations, see Figure 17:

- Connection of an external tool, acting as a client, to a process with an MPH-server, for example a test tool connecting to a SEA process.
- Connection of two or more processes with MPH-servers, for example two SEA processes.

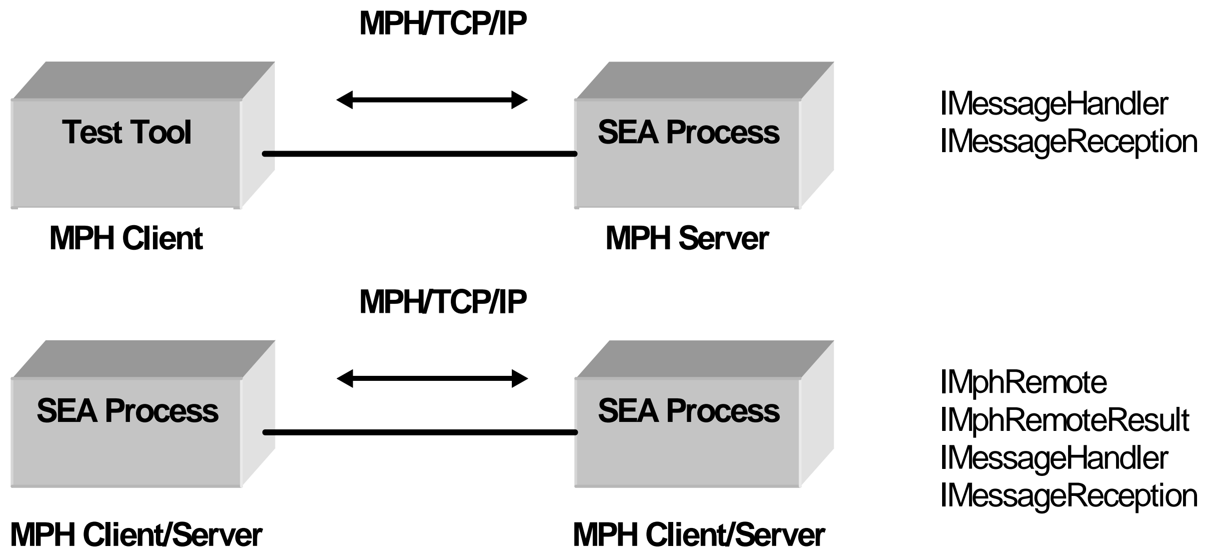


Figure 17: Example Configurations Using MPH

When an external test tool, acting as a client, is connected to a process running an MPH server, the IMessageHandler and IMessageReception interfaces are used. The interfaces are used both for setting up the connection and for communication between the entities. The two additional interfaces IMphRemote and IMphRemoteResult are used for connection setup between two peer entities with MPH servers. In the example configurations shown in Figure 17, the SEA process uses a special MPH component implementing both a server and a client part. The external test tool implements its MPH client part by means of an MPH client library.

Communication between an external test tool and a SEA internal component, for example, is established by the following steps:

1. The SEA internal component publishes itself as a connectable entity, by calling the MPH component within the SEA process.
2. The external test tool sets up a connection between itself and the SEA internal component, by calling an MPH client library function. The host name and IP address of the remote entity are needed when a connection is setup.
3. The external test tool sets up a communication channel for the connection, by calling an MPH client library function. Up to 255 different channels can be setup on the same port.

Table 1 gives an overview of the service primitives defined by the different interfaces.

Service Interface	Primitive	Description
IMessageHandler	AddMessageReceiver	Publish a component as a connectable entity.
	CloseChannel	Close an open channel.
	DeleteMessageReceiver	Unpublish a component.
	SendMessage	Send a message.
IMessageReception	ChannelClosed	Notification that a channel has been closed.
	HandleMessage	Receive a message.
	NewChannel	Notification that a channel has been established.
IMphRemote	ConnectRemote	Connect an internal component with an external one.
	DisconnectRemote	Disconnect an internal component from an external one.
IMphRemoteResult	ConnectionClosed	Notification that a connection has been closed.
	ConnectResult	Result from ConnectRemote.

Table 1: MPH Service Primitives

The communication between peer MPH entities is implemented by means of an MPH protocol. The MPH protocol defines the following packet format.

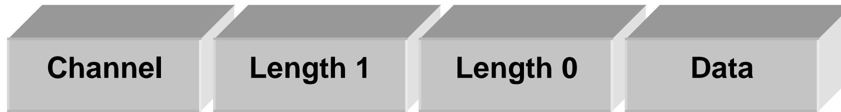


Figure 18: MPH Protocol Packet Format

The Channel field is a one byte long number specifying a channel number from 0 through 255, where channel 255 is reserved for a special control channel. Length 1 and 0 are two bytes specifying the length of the data; max packet size is 64 kB. The data format is not specified, but can be any stream of bytes agreed upon by the applications using the MPH.

The control channel has the following data format.

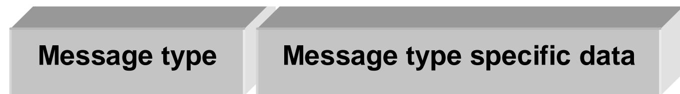


Figure 19: MPH Control Channel Data Format

The control channel is, among others, used for the following:

- Open a channel (request/response)
- Close a channel (request/response)
- Search for published components (request/response)

2.5.3 The CPP Emulator

The CPP Emulator [77],[78] is intended to behave as the real CPP platform. It has the same code, functions, communications, boards and test-suites. The most important differences compared to the real CPP platform is that the CPP Emulator is cheaper and provides higher availability for developers and testers.

There are several advantages with emulators. Two advantages are, as mentioned, that it is cheaper and provides higher availability, but there are more benefits. With an emulator, checkpoints can be created, that is a state (hardware, software and memory) can be saved and

loaded again. The possibility to use checkpoints saves a lot of time and money, since the setup of the emulated CPP node can easily be changed. It is also much more inexpensive to emulate in terms of hardware. The real CPP environment incorporates a variety of expensive hardware and software, while the emulated CPP environment can be run on an ordinary desktop PC. High availability enables testing and debugging before going to the real CPP environment, which also saves time and money. High availability also leads to increased quality. Other advantages are debugging on source code level and single step instructions, both providing quality insurance.

The objective with an emulator-based system is to make the system behave as the real target, and to act deterministic. The CPP Emulator is based on simulation of the instruction sets of the CPUs in the CPP target. Figure 20 shows the real CPP environment compared to the emulated, the CPP binary executable code is the same for both. The emulator replaces the real CPP hardware, which is also known as a hardware emulator.

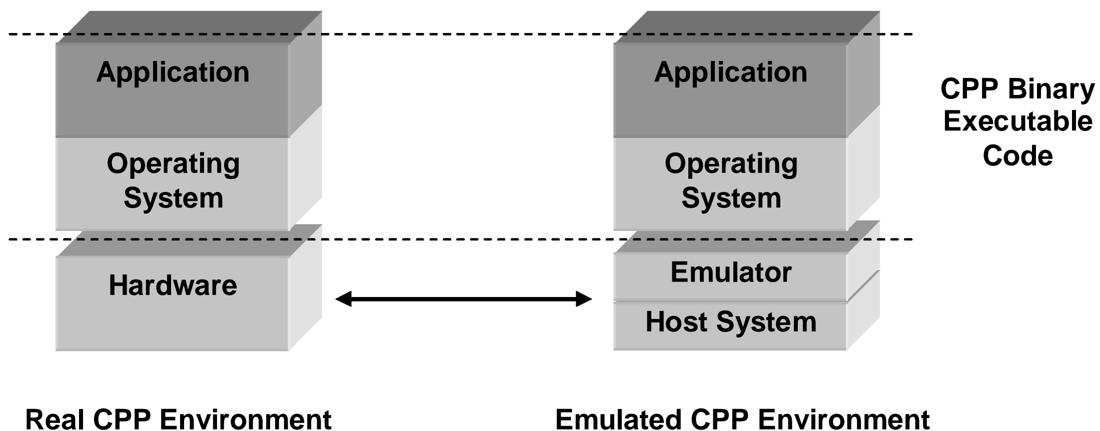


Figure 20: The Real versus the Emulated CPP Environment

2.5.4 Vega and MessageDriver

Another example of a test bed currently used by TietoEnator and Ericsson is Vega. Vega is a simulator for one of the processors in the TSP platform running the DICOS operating system. It is a part of the TSP platform and is used for function testing.

MessageDriver is a test tool for testing TSP platform software loaded into the Vega simulator. MessageDriver is only used for function testing in a simulated environment, not on target. MessageDriver uses its own script language for writing test cases, with code for preparation, action and expected result.

3 Test Tool Framework

3.1 Introduction

The requirements of a test tool framework were briefly described in Section 1.3. The purpose of this section is to give a more detailed description of the requirements of a test tool framework, as they are defined in this thesis, and also to discuss the advantages with an integrated test environment.

3.2 Background

The investigation for the TSP platform, see Section 1.2, showed that different TSP sub systems have developed different test tool solutions. The TSP platform is used in distributed telecom systems with many cooperating nodes. Each node may provide several interfaces, for different applications and services. In many cases, there is a specialized test tool for a specific node interface. The specialized test tool may be dependent on details of the node interface and cannot easily be re-used in other test environments. The dependency between the test tool and the SUT can schematically be described as a one-to-one relationship, as shown in Figure 21. With this model one specific test tool is used for testing one specific SUT.

Figure 21: One-to-One Relationship between the Test Tool and the SUT.

The MPH described in Section 2.5.2 provides a solution for publishing the interfaces of internal SEA entities so that external test tools can connect to them. The test bed, in this case the SEA system, implements an MPH server part and the test tool implements an MPH client

part. The test tool with the MPH client has hard dependencies to the test bed with the MPH server, and cannot easily be re-used, for example in a target environment. Therefore, this can also be seen as a one-to-one relationship between the test tool and the SUT, as in Figure 21.

The MessageDriver, see Section 2.5.4, uses its own test script language for testing TSP platform software loaded into the Vega simulator. MessageDriver cannot easily be re-used in other test environments.

With one specialized tool for each node interface, there will be many different tools. If the tools are developed independently of each other, there will also be many differences between the tools.

The conclusion is that there seems to be many advantages in creating an integrated test environment, these advantages are presented in the following section.

3.3 An Integrated Test Environment

Instead of hard dependencies between the test tool and the SUT, there should be as loose coupling as possible. With loose coupling between the test tool and the SUT, the possibility for re-using the test tool in other test environments increase. Another advantage is that changes to the interface of the SUT do not imply direct changes to the test tool. Instead of a one-to-one relationship between the test tool and the SUT, as in Figure 21, there should be a many-to-many relationship, as in Figure 22. With this model, a specific test tool is re-used for testing different SUTs. It should also be easy to connect many different test tools to a specific SUT, for different testing purposes.

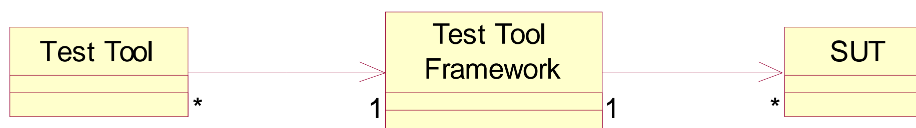


Figure 22: Many-to-Many Relationship between the Test Tool and the SUT.

With an integrated test environment there should be a few number of test tools for different testing purposes. It should be easy to adapt each test tool for use with a new SUT. From the

user's point of view the different test tools should have a uniform look and feel with similar user interfaces. With a reduced number of test tools, well integrated with each other; test preparation, test execution and test evaluation will be easier to perform. With an integrated test environment it will also be easier to create new test tools and to maintain existing ones. Creation of new tools will be easier because the unifying of different tools will require some base of design and implementation that can be extended. Maintenance of test tools will be much easier and cheaper because of the reduced number of tools and the separation of the test tools from the SUTs.

In conclusion, there are four different groups of users that will benefit from an integrated test environment: test environment responsibles, test developers, test executors and test tool developers, see Figure 23.

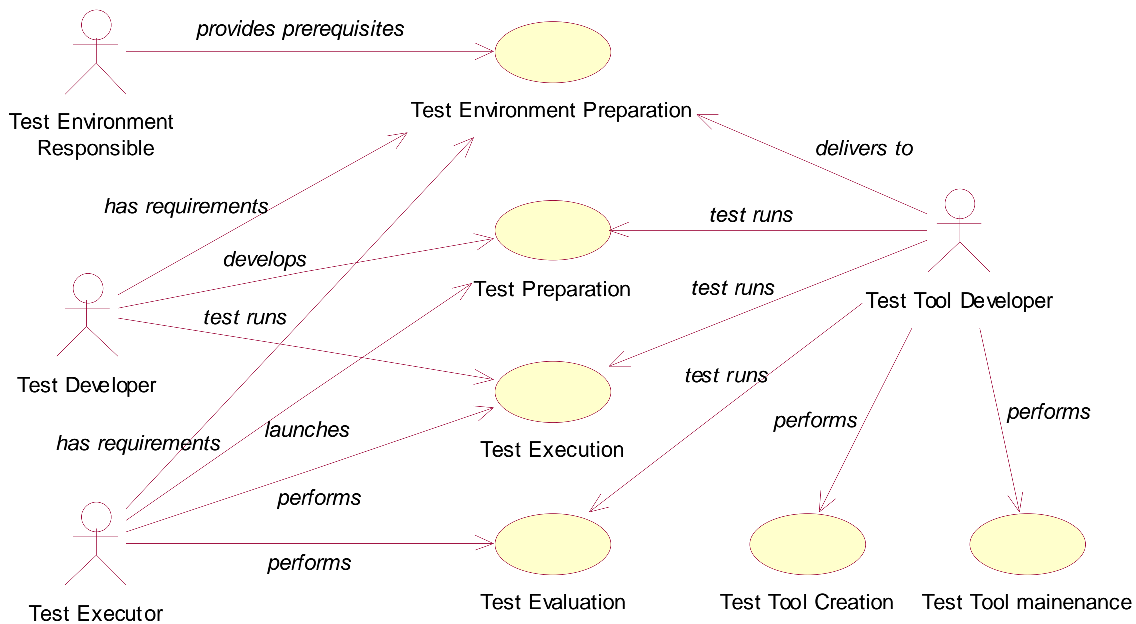


Figure 23: Users that Benefit from an Integrated Test Environment.

3.4 Test Tool Framework Requirements

3.4.1 Connection to the System Under Test (SUT)

An important requirement for a test tool framework is that it must provide some general mechanism for letting different test tools connect to the SUT, to support the model in Figure 22. This thesis focuses on a test tool framework to be used in the Telecom domain. An example of a SUT in the telecom domain may be a complete network of cooperating nodes, as the example in Figure 13 on page 27. The actual Implementation Under Test (IUT) is typically one of the nodes in the network. The node is tested by checking responses from stimuli sent to any of its external interfaces, that is black box testing. The external interface may be any man or machine interface, examples are given in Figure 24.

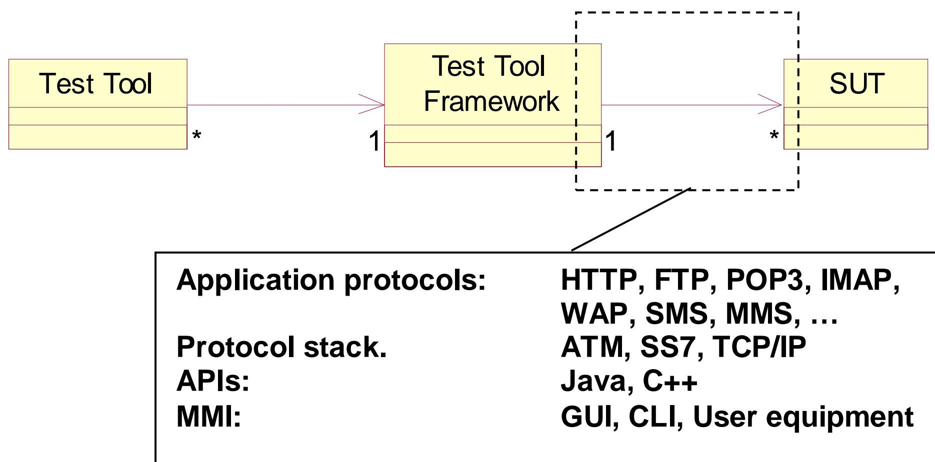


Figure 24: External Interfaces.

The test tool framework must provide some means for letting a test tool access a SUT through any of these interfaces. The test tool framework must also provide an infrastructure allowing for remote test bed launch, see Section 2.4.1. There must be functions for deploying different test input data to a remote machine, launching the test bed and executing the test, and for collecting different output produced during the test execution.

Another related requirement is that the test tool framework should use standardized techniques allowing for integration with other tools.

3.4.2 Centralized Functions

A second high-level requirement for a test tool framework is that it should provide a base of testing related functionality that can be used as is or extended. The test tool framework should support common functionality that different test tools can use, instead of re-inventing the functionality from scratch. The infrastructure for test launch and execution described in the previous section is one example of such common functionality. Another example is that the test tool framework should provide a common user interface to be used by different test tools. The testing related functions to be supported can be divided into three groups: test preparation, test execution and test evaluation.

Test preparation includes preparation of test plans, test cases and test data. Test cases can be created in an editor or re-used by importing them from an external repository. Test cases can also be generated automatically by a capture/play-back function, examples are capture of GUI events or capture of protocol messages such as HTTP. Test data can also be manually created, generated automatically or re-used. There should also be support for associating test data with test cases.

Test execution includes drivers for the actual test execution and different comparators for verifying SUT responses against expected results. Test execution should also support different kinds of runtime monitoring. Runtime monitoring includes both monitoring of the test script execution and different monitoring of the SUT. Both logs and traces from the test scripts or the SUT may be viewed and analysed during runtime. There may also be additional information such as performance monitoring gathered from the runtime environment of the SUT. Runtime debugging should also be supported, to help locate the source of defects found.

Test evaluation consists of analysis of different output generated from the test execution. There must be functions for analysing the test execution history, with verdicts for the different test cases. There should be support for analysing logs and traces generated by the test scripts

or by the SUT. Test evaluation may also include support for analysing different profile data such as code coverage, performance or statistical data. Profiling data normally implies some sort of instrumentation of the SUT, which is not normal when function testing, but can also be gathered from the runtime environment by means of different probes.

Figure 25 summarizes some centralized functions.

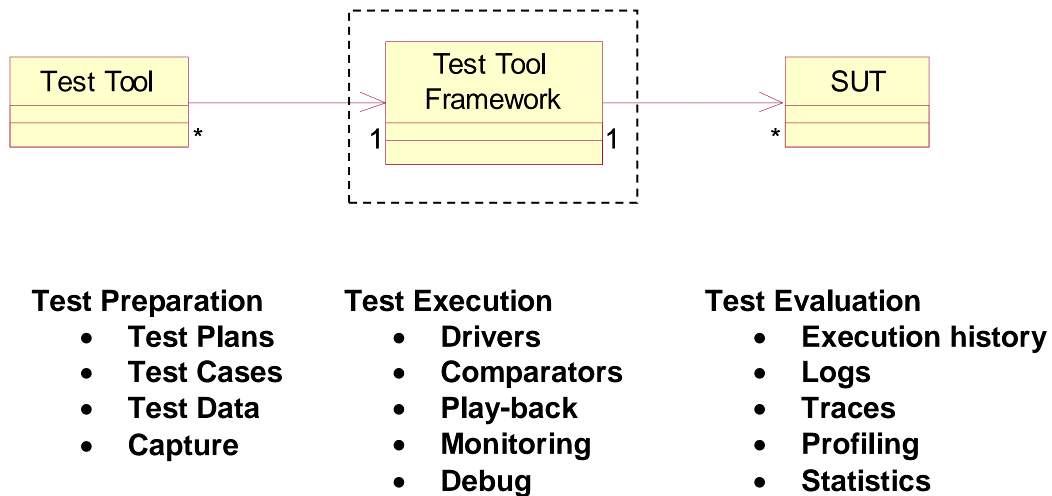


Figure 25: Centralized Functions.

4 Market Analysis

4.1 Introduction

This chapter describes the result from the market analysis that was carried out in the project. The purpose of the market analysis was to increase the knowledge about different test tools, and specifically about available test tool frameworks that may be used for creating an integrated test environment. The main objective was to find a test tool framework, product or technical solution, that TietoEnator could use in future projects. The goal was to find an existing product on the market, which was as complete as possible and ready to use with as small modifications as possible. The product should use standard, open techniques and preferably be open source.

The following describes the method used for carrying out the market analysis. Furthermore, the different results from the market analysis are described.

First, a background study was performed, to get a deeper knowledge about testing concepts in the telecom domain. The background study is described in Chapter 2. To be able to describe different products in a common way, and also to be able to compare different products, a necessary activity was to establish a common terminology for all different testing-related concepts. Different terms and concepts had to be defined and described, together with the relations between them. Testing literature, articles, the Internet and different testing-related standards were studied in order to find the correct terminology and definitions to use. General software testing concepts are described in Section 2.2. A list defining all testing related concepts used in this thesis can be found in Appendix A.

After the background study, some basic requirements of a test tool framework were specified. The main issue that TietoEnator wanted to solve was to find a standard technique for connecting different test tools to the SUT. An example of a test tool integration currently used by TietoEnator and Ericsson is the MPH solution described in Section 2.5.2. The MPH is

an “in-house” design that TietoEnator wanted to replace with some kind of standardized technique or product. We then extended the requirements to also include centralized functions to be used by different test tools. The requirements are summarized with the model of a test tool framework that integrates many different test tools and that adapts to many different SUTs, see Section 3.4.

The list of testing related concepts and requirements of a test tool framework were extended when different products were studied during the market analysis, as a consequence of getting a deeper understanding of different concepts and of functionality that should be supported.

The first activity in the market analysis was to search for similar studies performed by other people. We did not find any similar studies.

Secondly, a thorough search for candidate products was made, in order to create a list of products to investigate. The product search mostly included Internet search, but also database searches at Karlstad University. The most interesting products were then studied in detail. We chose to both study ready-to-use products and frameworks for building new test tools. Each candidate product was studied by reading documentation available on the Internet, such as technical sheets, white papers and user manuals. Evaluation copies of the products were downloaded whenever possible, in order to get access to on-line documentation as well as more detailed technical documentation. There was no time for test running the different products, but the investigation was purely theoretical, based on written information only. An overview of the products studied is given in Section 4.2. The frameworks that were found are described in Section 4.4 and Section 4.5 respectively. A full description of the ready-to-use products studied can be found in Appendix D.

Furthermore, the functionality of each product was summarized in a table with different comparison points. The comparison points are based on the requirements of a test tool framework described in Section 3.4. The purpose with the comparison points was to be able to compare different products with a common terminology. The comparison points are

described in Section 4.3 and summarized in a table in Appendix E. The evaluation for each product studied, based on the comparison points, can be found in Appendix F.

Finally, the description and evaluation table for each product were mailed to the respective companies for review.

The result of the market analysis is summarized in Section 4.7. Summary and evaluation of the market analysis can be found in Chapter 6.

4.2 Candidate Products

The following candidate products were studied in the market analysis:

- Danet TTCN-3 Toolbox
- Eclipse TPTP
- IBM Rational Test Manager
- JUnit
- OpenTTCN Tester for TTCN-3
- Scapa Test and Performance Platform 3.1
- Software Testing Automation Framework (STAF)
- Telelogic TAU/Tester
- Testing Technologies TTworkbench

Danet TTCN-3 Toolbox, OpenTTCN for TTCN-3, Telelogic TAU/Tester, and Testing Technologies TTworkbench are all TTCN-3 based products. An introduction to TTCN-3 can be found in Appendix C. Eclipse TPTP is an open source framework for testing and profiling tools. IBM Rational Test Manager is one of many testing products from IBM Rational that together cover the full life-cycle of software testing and are well integrated with each other as well with other IBM Rational products. JUnit is a simple open source framework for unit testing Java classes. Scapa Test and Performance Platform is a performance testing tool with the main focus to help locating performance related problems in server-based systems. The

Software Testing Automation Framework (STAF) is an open source product that provides a re-usable infrastructure for remote test execution.

Appendix D contains more detailed descriptions of these products and Appendix F contains an evaluation table for each product.

Out of the nine products studied there are two that can be classified as test tool frameworks according to the requirements defined in this thesis: Eclipse TPTP and STAF. JUnit is a simple framework for unit testing Java classes. The remaining products are designed to be used as they are, and not as a base for building new tools.

During the market analysis we found several other interesting products, but we did not have time to study them, among others these products:

- ApTest Manager
- Ascert TestPilot
- Mercury TestDirector
- Segue SilkTest
- Tata SmarTest

4.3 Comparison Points

The points that were used for comparing and summarizing different products were categorized into six groups:

- General
- Test methods supported
- Interoperability
- Test Preparation
- Test Execution (real-time)
- Test Evaluation (post mortem)

The general group summarizes the use of the product with its main target SUT environment, different test management phases supported and different platforms supported.

Test methods supported includes support for automatic testing, testing of distributed systems, GUI-testing, load/stress testing, manual testing, protocol testing and unit testing. Testing of distributed systems is considered to be supported if remote test deployment and execution are supported. There should also be means for collecting different data from remote machines, such as logs and traces generated by the SUT. GUI-testing should, for example, include support for automatically creating test scripts by recording user-interface events.

Interoperability contains important functionality that enables integration with different external systems. One example is data models used by the product for handling different test artifacts, such as test cases or test execution histories. Other examples are support for exporting or importing information to/from external databases and support for remote test bed launch. Different standards used or supported by the product are also very important for enabling integration with other products, such as storing different test artifacts in XML-format for example. Another interoperability point is the technique used for adapting to the interfaces of the SUT.

The division of the three remaining groups; test preparation, test execution and test evaluation, very nicely maps to the typical phases of a project; with a distinct preparation, execution and evaluation phase. Test preparation contains different points necessary to complete before the test can be executed. Examples are re-use of test cases by importing them from external systems and which test script languages that are supported by the product. Test execution contains different runtime support, such as runtime monitoring of logs and traces generated by the SUT. Test Evaluation contains points for analyzing test execution history as well as log, traces and different profiling data generated by the SUT during the test execution.

4.4 Eclipse TPTP

4.4.1 Introduction

Eclipse [25] is an open source platform designed for building Integrated Development Environments (IDEs). The Eclipse platform has been designed in a general way, meant to be useful for a wide range of applications. “Eclipse is a kind of universal tool platform – an open extensible IDE for anything and nothing in particular” [25].

The design is based upon the concept of plugging in tools (plug-ins) to a common infrastructure [32]. The Eclipse platform provides a framework and an infrastructure with building blocks accessible through open APIs that facilitate the development of new tools. There is also a mechanism for automatically discovering, integrating and running plug-ins. By using the building blocks as a base, different IDEs, or tools can be created. One example of a tool built upon the Eclipse platform is the widely used Java Development Tooling (JDT) [26] included in the Eclipse Software Development Kit (Eclipse SDK).

One of the many Eclipse subprojects is the Test and Performance Tools Platform project (TPTP) [27], formerly Hyades, which goal is to provide an open development platform for test and performance tools, collectively referred to as Automated Software Quality (ASQ) tools. The platform includes both a general infrastructure for test and related activities, as well as example tool implementations, which can be used as is or extended.

The TPTP project was formed in August 2004 and the Hyades project was formed in December 2002. The following organizations are participating in the development of the TPTP project [29]:

- Computer Associates (Test and monitoring design)
- Compuware (Monitoring Design)
- FOKUS (Test design)
- IBM (Trace, Test, Log, Data Collection)
- Intel (Data collection)
- OC Systems (BCI -Data Collection, Probekit)

- SAP (Test design)
- Scapa Technologies (Test, Trace)

An important goal with the TPTP project is to achieve interoperability between different testing tools. Interoperability will be achieved in two ways: by building a generic, extensible infrastructure and, by wherever possible, using existing standards.

4.4.2 Functionality

Most of the Eclipse platform, including the TPTP platform, is written in Java; and Java development and Web application development are also the main focus for the TPTP functionality.

The TPTP platform provides a common infrastructure for testing, tracing, profiling, monitoring and logging tools:

- The testing functionality includes test editors and supports test deployment and execution on remote and distributed systems. There are also functions for creating data pools to provide a test with variable data.
- The tracing and profiling functionality consists of data collection and analysis for Java: collection of local or distributed execution stacks as well as heap information. The purpose with profiling is to help finding performance and memory usage problems in Java and Web applications.
- The logging and monitoring functionality includes support for importing log events generated by the SUT into a common format: Common Base Event format (CBE) [34]. The imported logs can be monitored and analyzed, post execution or in real time, with functions such as navigating, sorting, filtering and searching.

New test tools can be created by using the common infrastructure in the TPTP platform as a base. The TPTP platform also includes the following example tool implementations that are ready to use:

- JUnit based unit testing tool
- Test tool for browser-based applications
- Manual testing tool

The JUnit based unit testing tool has functions for creating a test suite with test methods (test cases), generating Java code, running a test and analyzing the test results.

The browser-based applications testing tool includes functionality for recording user interactions with a browser-based application, editing a test, generating an executable test, running a test and analyzing the test results.

The manual testing tool may be used for creating and running a test suite with manual test cases.

4.4.3 Architecture

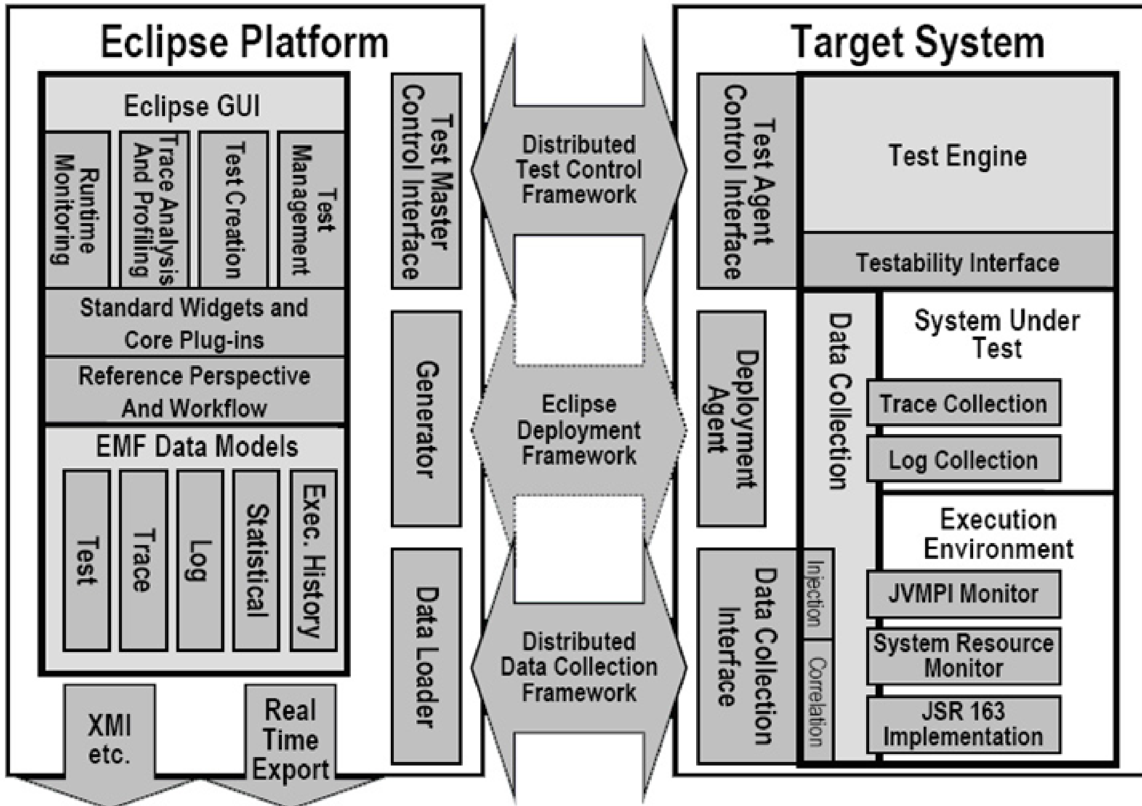


Figure 26: TPTP Architecture Overview

Figure 26 shows an overview of the TPTP Architecture. The architecture includes three sub-frameworks: a test control framework, a deployment framework and a data collection framework.

The test control framework handles test execution by the use of a group of three components: the test agent control interface, the testability interface and the test engine. The test engine is responsible for generating the actual stimuli to the SUT.

The deployment framework supports deployment of agents for collecting data as well as deployment of tests with associated meta data (data pools).

The data collection framework includes functionality for collecting and importing trace and log data generated during test execution into the Eclipse Modeling Framework (EMF).

4.4.4 Eclipse Modeling Framework (EMF)

In order to achieve tool interoperability, an important and central strategy in the design of the Eclipse platform is the use of standardized data models. The data models are abstract descriptions in the Unified Modeling Language (UML) of different assets (tests, traces, logs etc), and they are provided with a concrete implementation through the Eclipse Modeling Framework (EMF) [35], [31]. The data models are provided in the following areas [27]:

- A test data model that supports test cases, input data, results and execution history. It consists of three models:
  1. A data model for creation, definition and management of test artifacts. This model implements the UML 2 Testing Profile meta model [33].
  2. A data model for test case behaviors. It implements the UML 2 Interaction Meta model.
  3. A data model for test execution history. It supports execution traces and results from different test types.
- A trace data model supports traces of local and distributed execution stacks and heaps.
- A log data model that supports sequence of CBEs and other logged messages that are transformable into CBE.
- A statistical data model that supports snapshots of arbitrary data over time.

4.4.5 Standards

TPTP uses the following standards:

- Unified Modeling Language (UML) [36]. EMF data model descriptions.
- UML 2 Test Profile (U2TP) [33]. EMF test data model.
- Common Base Event (CBE) [34]. EMF Log Model.

- Java™ Virtual Machine Profiler Interface (JVMPI) [39]. Trace model (maps closely to JVMPI [27]).
- JVM™ Tool Interface (JVMTI) [40]. Trace model (maps closely to JMTI [27]).
- Java Management Extensions (JMX) [41]. Statistical model (maps well onto JMX, Microsoft PerfMon [27]).
- Extensible Markup Language (XML) [37].
- XML Metadata Interchange (XMI) [38]. EMF data model persistence.

4.5 Software Testing Automation Framework (STAF)

4.5.1 Introduction

Software Testing Automation Framework (STAF) [61] is an open source product from IBM released under the GNU Lesser General Public License (LGPL) [64]. STAF is an automation framework intended to make it easier to create and manage automated test cases and test environments [62]. STAF is designed around the idea of reusable components, called services (such as process invocation, resource management, logging, and monitoring).

STAF was designed with the following points in mind [62]:

- Minimum machine requirements - This is both a hardware and a software statement.
- Easily useable from a variety of languages, such as Java, C/C++, Rexx, Perl, and TCL, or from a shell (command) prompt.
- Easily extendable - This means that it should be easy to create other services to plug into STAF.

STAf eXecution Engine (STAX) is an execution engine which can help automate the distribution, execution, and results analysis of test cases. STAX is built on top of three existing technologies: STAF, XML, and Python.

4.5.2 Functionality

The functionality in STAF is provided through services. The following are examples of services that are ready to use.

Service	Description
PROCESS	Start, stop, and query processes.
EVENT	Provides a publish/subscribe notification system.
LOG	Provides a logging facility for test cases.
RESOURCE POOL	Manages exclusive access to pools of elements, e.g. VM UserIDs or Software Licenses.
MONITOR	Allows a test case to publish its current running execution status for others to read.

Table 2: Example STAF Services

4.5.3 Architecture

STAF runs as a daemon process called STAFProc on each machine, see Figure 27. The collection of machines on which STAF has been installed is referred to as the STAF Environment.

STAF operates in a peer-to-peer environment; in other words, there is no client-server hierarchy among machines running STAF.

STAF services are reusable components that provide all the capability in STAF. Each STAF service provides a specific set of functionality (such as Logging) and defines a set of requests that it will accept. STAF Services are used by sending STAF requests to them. A STAF request is simply a string which describes the operation to perform. STAF requests can be sent to services on the local machine or to another, remote, machine in the STAF Environment. In either case, the STAFProc daemon process handles the sending and receiving of requests.

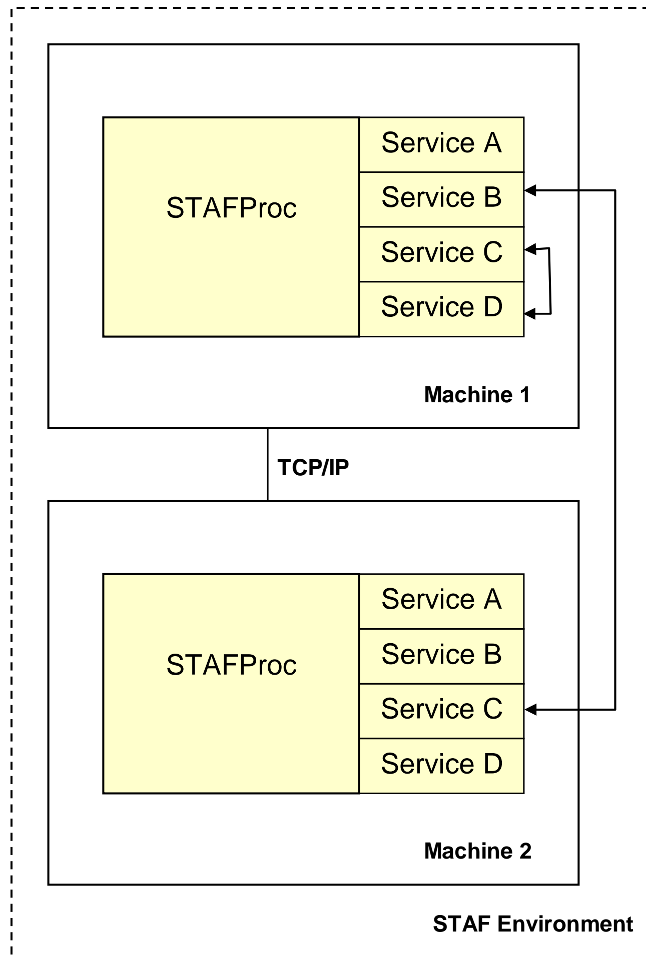


Figure 27: STAF Architecture Overview

4.6 Eclipse TPTP versus STAF

This section compares Eclipse TPTP and STAF. For a more detailed evaluation of Eclipse TPTP and STAF based on common comparison points, see Appendix F.2.1 and Appendix F.2.2 respectively.

TPTP and STAF both provide an infrastructure that enables remote test execution. Both products can be used for integrating many test tools and to adapt to many different SUTs. Both Eclipse TPTP and STAF also use component technology, but in very different ways.

Eclipse TPTP supports components by the concept of plug-ins that can be added both to the client Workbench and to the Remote Agent Controller (RAC). Eclipse TPTP is also very Java-centric and builds on common object oriented design patterns. Another design solution that makes Eclipse TPTP very extensible is the concept of extensions and extension points. Any plug-in can provide its own extension points that other plug-ins can extend.

STAF implements a "mini-CORBA" solution [63] with a peer-to-peer network of daemon (STAFProc) processes that runs on each machine. Components are supported in the form of services that can be registered (plugged-in) with the STAFProc processes. Communication between services is provided by means of simple request/reply pairs. To ask for different tasks to be performed by a specific service or to get different kind of information, different requests are used. Both requests and replies are sent as simple text strings, which means that many different programming languages can be used for implementing services.

The main difference between Eclipse TPTP and STAF is the size of the products. Eclipse, with TPTP, is a much bigger product with a broad user community. STAF was created by IBM, and was used internally at IBM only, before being released as open source. Eclipse TPTP is the result from the cooperation among several companies.

An advantage with STAF compared to Eclipse TPTP should be performance. STAF was designed to consume a small amount of system resources such as memory usage. Eclipse TPTP, however, uses a great amount of system resources, both on client and server sides. The

RAC is implemented in C for performance reasons, but the test execution components are implemented in Java, and a remote test bed launch starts two different JVMs on the remote machine.

However, there are several advantages with Eclipse TPTP compared to STAF. Eclipse TPTP contains a framework with a rich set of functions for building new test tools. The Eclipse Modeling Framework (EMF) constitutes a solid base for supporting many different models, with both runtime access and persistence. Different standards are supported, see Section 4.4.5, and the members of Eclipse TPTP are even contributing to the work of defining new standards, such as the UML 2 Testing Profile [33].

Therefore, the conclusion is that Eclipse TPTP better fulfills the requirements for a test tool framework as defined in this thesis.

4.7 Summary

The main objective with the market analysis described in this chapter was to find a test tool framework, product or technical solution, that TietoEnator could use in future projects. We found two products: Eclipse TPTP and STAF. Both should be possible to use for building an integrated test environment, but Eclipse TPTP has much more centralized functions that facilitate building new test tools, and is therefore considered a better choice.

5 Prototype

5.1 Introduction

A prototype was implemented as a proof of concept to show that Eclipse TPTP can be used as a general test tool framework. The support for connecting different test tools to the SUT was of special interest for further investigation.

The main objective for the prototype was to implement support for executing Expect [60] test cases against CPP system software loaded into the CPP-Emulator. TietoEnator currently develops and maintains the CPP Emulator. Expect was chosen as script language because it is currently used for testing target CPP system software at Ericsson.

This section contains many concepts that are further described in the user manual, see Appendix G. Appendix G.2 contains Eclipse vocabulary and Appendix G.3 describes the pre-defined architecture of Eclipse TPTP. These appendices are recommended to read before reading this chapter, or to be used as a reference when reading this chapter.

The prototype makes it possible for a tester to prepare a test configuration to be run on a remote machine. The test configuration includes selection of the Expect test scripts, the SUT configuration, and the CPP Emulator version. When the tester starts the test from the client Workbench, the test bed is automatically launched on the remote machine, the test scripts are executed, and the test bed is torn down. The tester can evaluate the test result by viewing the test execution history in the client Workbench. Figure 28 shows a conceptual diagram for the interactions when the tester prepares, runs and evaluates the test from the client Workbench.

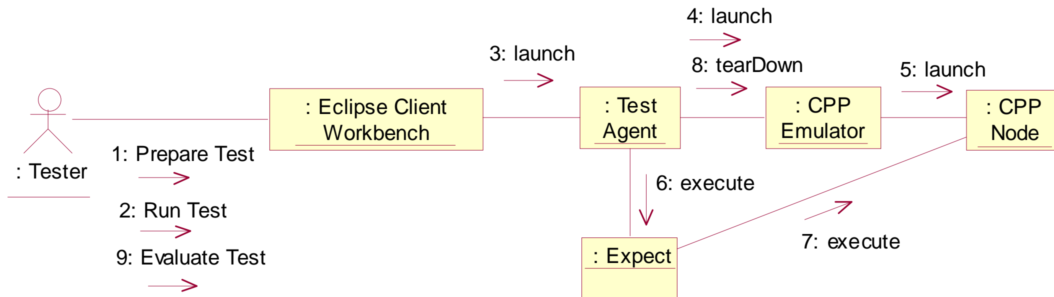


Figure 28: Remote Test Bed Launch

The CPP platform is described in Section 2.3.4 and the CPP Emulator is described in Section 2.5.3. The CPP hardware and CPP software configuration to be loaded into the CPP Emulator are specified in different configuration files. The configuration files are given as parameters when starting the CPP Emulator.

5.1.1 Prototype Components

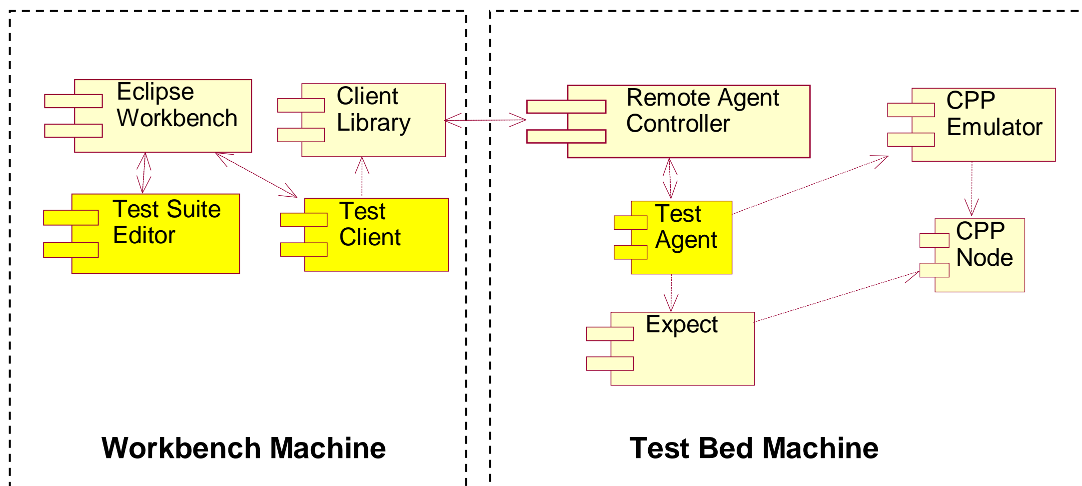


Figure 29: Prototype Components

Figure 29 shows the different components in the prototype implementation. The prototype implements the Test Suite Editor, the Test Client and the Test Agent. The Test Suite Editor

and the Test Client are plug-ins in the Eclipse Workbench. The Test Agent is a plug-in in the Remote Agent Controller (RAC). In addition, the Test Client uses a test execution part of the Client Library. The Test Suite Editor incorporates the graphical components for creating and editing the TPTP Expect Test Suite in the Eclipse Workbench. The Test Client makes up the client side for launching and executing the Expect test suites, while the Test Agent makes up the server side for launching and executing the Expect test suites. The Test Agent also generates execution events which are sent to the client Workbench during test execution.

5.1.2 Deployment

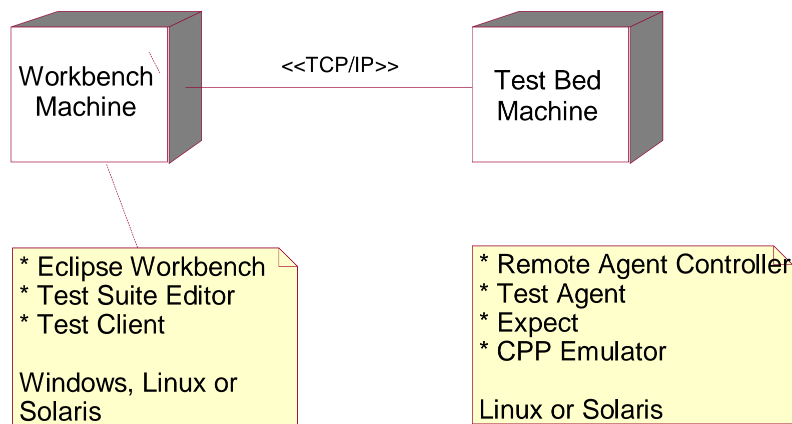


Figure 30: Deployment of the Prototype Components

Figure 30 shows the deployment of the prototype components. The Eclipse Workbench is run on a client machine with Windows, Linux or Solaris operating systems. The Test Suite Editor and Test Client are installed as plug-ins in the Eclipse Workbench. The RAC with the Test Agent plug-in runs on a remote Linux or Solaris machine. The CPP Emulator and Expect are run on this machine as well.

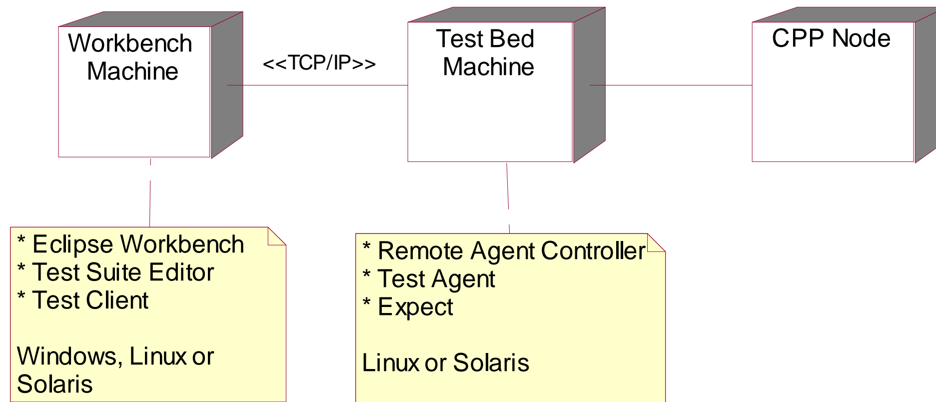


Figure 31: Target Environment Deployment

The prototype can be used in the target environment as well, as shown in Figure 31. The only difference is that the CPP Emulator is not used in the target environment.

5.2 Requirements

5.2.1 Use Case: Execute Expect Test Against the CPP Emulator

The requirements for the prototype can be summarized with the use case shown in Figure 32. The use case Execute Expect test against the CPP Emulator includes the use cases Prepare Test, Run Test and Evaluate Test.

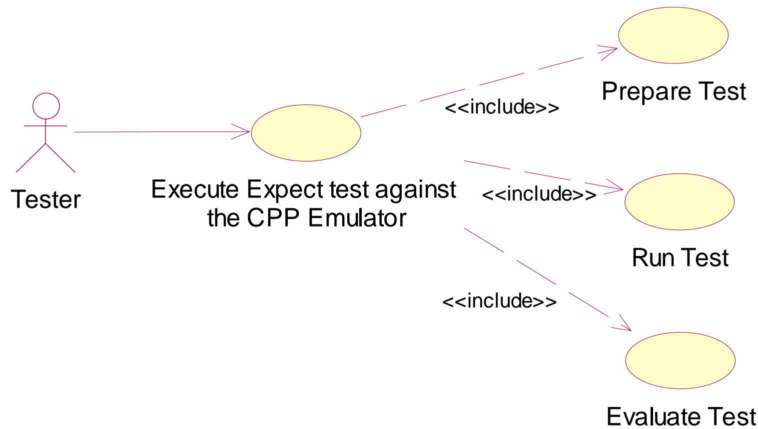


Figure 32: Prototype Use Case

Appendix G contains a User Manual for the prototype, which describes the use of the prototype in detail.

5.2.2 Use Case: Prepare Test

5.2.2.1 Purpose

The purpose of the Prepare Test use case is to let a Tester prepare a test by configuring the Expect test suite, the CPP system software configuration and the CPP Emulator configuration.

5.2.2.2 Precondition

The precondition for this use case is that the Tester has the following data:

- One or more Expect test scripts
- Host name of the remote machine to run the test on
- Telnet port number for the CPP node running in the CPP Emulator
- CPP Emulator configuration files
- A ClearCase view

The Expect test scripts must be configured to be run against the CPP Emulator. The scripts take two parameters: host name and telnet port.

The Expect test scripts communicate with the SUT, that is the CPP node loaded in the CPP Emulator, via Telnet. The Telnet port number for the CPP node must be configured in the CPP Emulator. The prototype uses port forwarding in the CPP Emulator, which assigns the default port number 4023 to the main board in the CPP node. If another CPP Emulator is running on the same machine, this port number may be occupied and the Tester can then set another port number to use.

The CPP Emulator configuration files define the CPP node configuration to be loaded and executed in the CPP Emulator. The configuration files define both CPP hardware and software configuration.

IBM Rational ClearCase [79] is a configuration management system. The ClearCase view defines which CPP Emulator version to use. When running the test, the ClearCase view is started on the server machine, in order to access the CPP Emulator executable. The view name is also used for setting the PATH environment variable, which is used by the CPP Emulator to access its installation files.

The test is run on a machine which has a RAC installed. The name of this machine must be specified.

5.2.2.3 Postcondition

The postcondition for this use case is that the Tester has an Expect test suite configuration that is ready to run.

5.2.2.4 Description

This use case starts when the tester creates a new TPTP Expect test suite. The tester may also change an already existing test suite. The tester fills in the fields in the TPTP Expect test suite editor and saves the test suite. The test suite is then assigned to an artifact element, which in turn is bound to a deployment element, together with a location element.

5.2.3 Use Case: Run Test

5.2.3.1 Purpose

The purpose with the Run Test use case is to let the tester run a prepared test suite.

5.2.3.2 Precondition

The precondition for the Run Test use case is that the Prepare Test use case has finished successfully. There must also be a RAC running on the remote machine to launch the CPP Emulator on and to run the test on.

5.2.3.3 Postcondition

The postcondition for this use case is that the Expect test scripts in the test suite have been executed. There is a test execution history showing the result of the test.

5.2.3.4 Description

This use case starts when the tester runs an Expect test suite configuration. The tester either creates a new test configuration or uses an existing one. The following points briefly describe the user and system interactions:

1. The tester starts the test execution.
2. The test client sets the PATH on the remote machine and orders the RAC to launch the remote test agent with the test suite configuration parameters.
3. The test agent starts the specified ClearCase view.
4. The test agent launches the CPP Emulator with the specified CPP node configuration.
5. The test agent executes the Expect test scripts. Each test script opens a Telnet connection to the CPP node on the specified host and port. The output from the Expect test scripts are sent back to the test client as execution message events and gets stored in a test execution history.
6. The tester can view the test execution events in an execution history view in the Eclipse Workbench.

7. When all test cases have been executed, the test agent terminates the CPP Emulator and stops the ClearCase view.

5.2.4 Use Case: Evaluate Test

5.2.4.1 Purpose

The purpose with the Evaluate Test use case is to let the tester view the test execution history in order to evaluate the test.

5.2.4.2 Precondition

The precondition for the Evaluate Test use case is that the Run Test use case has finished successfully.

5.2.4.3 Postcondition

The postcondition for this use case is that the tester has viewed the execution history and evaluated the test.

5.2.4.4 Description

The test execution history is saved in the current project in a file with suffix .execution. If the test just has been run, the test execution view can be opened directly by selecting the .execution file in the navigator view for the current project. The .execution file is a zip-file containing an XML document with the execution history events from the test agent. Test execution histories from earlier test executions can be opened via the File menu in the Eclipse workbench.

The prototype implementation does not add any functionality for this use case, but it uses already existing functionality in TPTP.

5.3 Design

5.3.1 Introduction

This section describes the design of the prototype. First an overview of the TPTP design is given, see Section 5.3.2. The scope of the prototype implementation is described in Section 5.3.3. The remaining sections are realization of the use cases described in Section 5.2. Realization of use case "Prepare Test" can be found in Section 5.3.4 and realization of use case "Run Test" can be found in Section 5.3.5. There is no description of realization of use case "Evaluate Test", since the prototype does not add any functionality to this use case.

5.3.2 Eclipse TPTP Design Overview

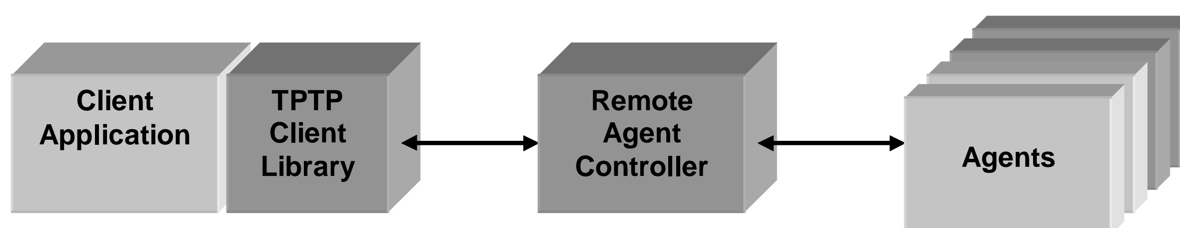


Figure 33: TPTP Basic System Structure

Figure 33 shows the basic system structure of TPTP. TPTP includes a client library and a Remote Agent Controller (RAC) that are ready to use. The Client Library supports the creation of different Client Applications written in Java or C++. Specialized Agents are written for different purposes, such as test execution or data collection. The Agents implement the interfaces to the SUT. The RAC is implemented in C. The Agents may be implemented in any language. There are sample Agent implementations in Java. The communication between the Client Library and the Agent Controller uses TCP/IP. The Agent Controller communicates with the different Agents by means of shared memory.

5.3.2.1 TPTP Test Tools Project

The Eclipse Test & Performance Tools Platform (TPTP) Project is divided into four projects:

- TPTP Platform
- Monitoring Tools
- Testing Tools
- Tracing and Profiling Tools

The TPTP Platform project provides the core functionality in TPTP, which the other projects extend. The prototype uses functionality provided by the Testing Tools project. The testing functionality includes test editors and supports test deployment and execution on remote and distributed systems. There are also functions for creating data pools to provide a test with variable data. Furthermore, the Testing Tools project provides three tool example implementations that are ready to use. The common infrastructure used by these tools can be used to create new customized test tools. The prototype has been realized by using this functionality.

5.3.2.2 Test Execution Overview

The design of the test execution components in TPTP is described in a presentation from Joe Toomey et al [30].

To run a test from the Eclipse Workbench, a test launch configuration has to be created. The test is started by running the launch configuration. The launch configuration calls the Test Execution Harness to launch the test. The Test Execution Harness, in turn, invokes different Test Execution Components, see Figure 34.

Figure 34: Test Launch Interactions

Each Test Execution Component consists of a client part and a server part, which interact with each other. There are four different Test Execution Components: the ExecutionEnvironment, the ExecutableObject, the Executor and the RemoteHyadesComponent. The Test Execution Components are implemented in Java.

The ExecutionEnvironment component handles environment variables in the remote test environment. For example, the client part of the ExecutionEnvironment can set the CLASSPATH to be used in the remote test execution environment.

The ExecutableObject handles command line arguments, such as Java Virtual Machine (JVM) arguments, main class for the remote test agent and parameters to the remote test agent.

The Executor component launches a JVM with the arguments specified by the ExecutableObject in the environment specified by the ExecutionEnvironment.

The RemoteHyadesComponent provides the communication channel to control the test and to send back test results to the client.

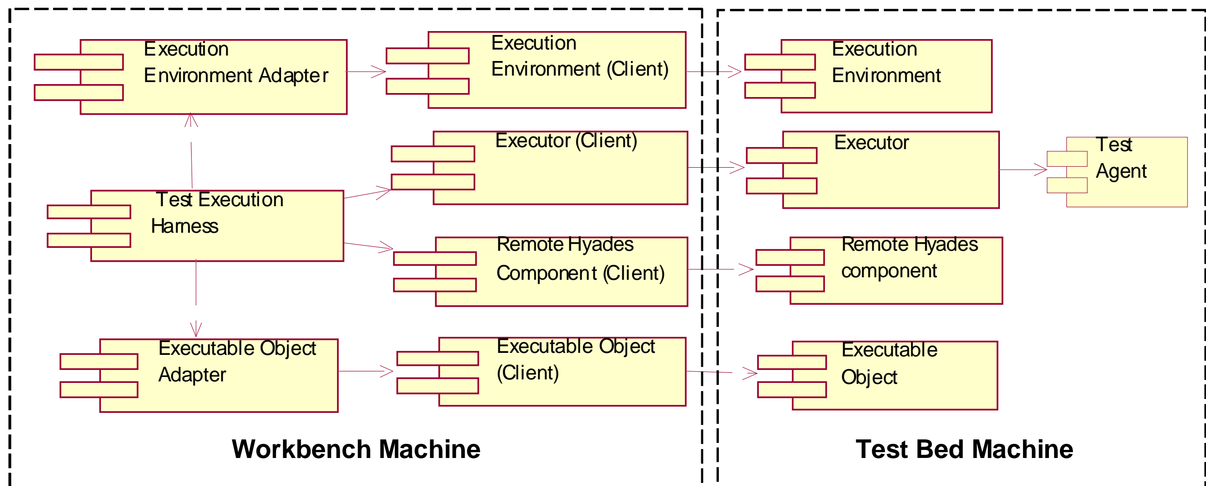


Figure 35: Test Execution Components

To use the execution components, for creating a custom test execution environment, two adapter classes are created: ExecutionEnvironmentAdapter and ExecutableObjectAdapter, see Figure 35. The Test Execution Harness calls these adapter classes during the test launch.

5.3.3 Scope of the Prototype

The prototype implements three Eclipse plug-ins, two Eclipse TPTP Client plug-ins and one Eclipse TPTP RAC plug-in:

- Eclipse TPTP Client plug-ins
  - com.tieto.eclipse.tptp.cpp.expect.ui_1.0.0
  - com.tieto.eclipse.tptp.cpp.expect.core_1.0.0
- Eclipse TPTP RAC plug-in
  - com.tieto.eclipse.tptp.cpp.expect

The com.tieto.eclipse.tptp.cpp.expect.ui plug-in implements the Test Suite Editor with related Test Suite Wizard, see Figure 36. The design of the Test Suite Editor and the Test Suite Wizard is described in Section 5.3.4.

The com.tieto.eclipse.tptp.cpp.expect.core plug-in implements the Test Client, see Figure 36. The Test Client makes up the client side for launching and executing the Expect test suites. The design of the Test Client is described in Section 5.3.5.

The com.tieto.eclipse.tptp.cpp.expect plug-in implements the Test Agent, see Figure 36. The Test Agent makes up the server side for launching and executing the Expect test suites and is also responsible for generating the execution events which are sent to the client Workbench during test execution. The design of the Test Client is described in Section 5.3.5.

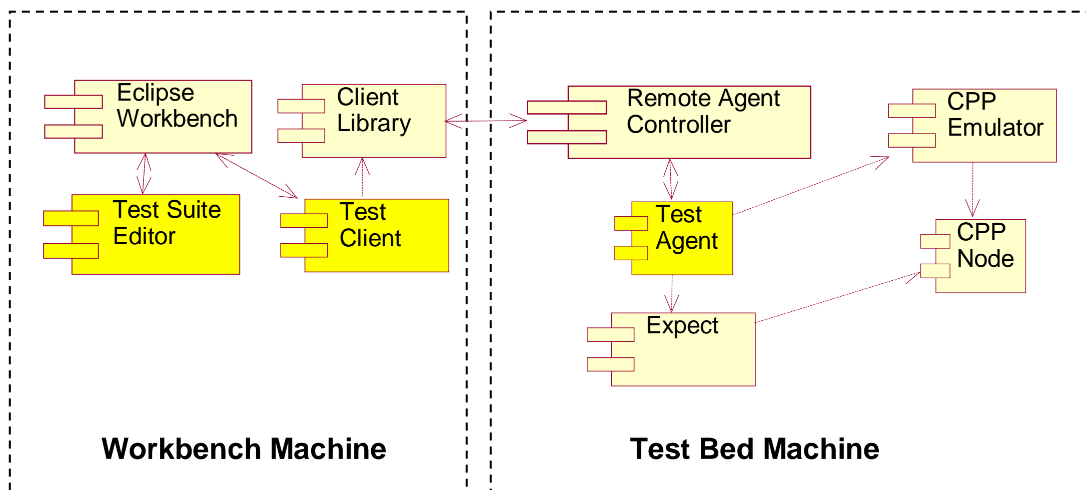


Figure 36: Prototype plug-ins

5.3.4 Prepare Test

5.3.4.1 Test Suite Wizard

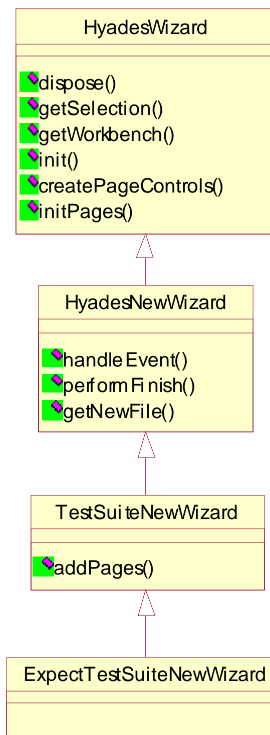


Figure 37: Expect Test Suite New Wizard

The prototype implements the class ExpectTestSuiteNewWizard, which responsible for creating a new Expect Test Suite, see Figure 37. The ExpectTestSuiteNewWizard class inherits from the TestSuiteNewWizard class. The ExpectTestSuiteNewWizard class is registered in the extension point org.eclipse.ui.newWizard.

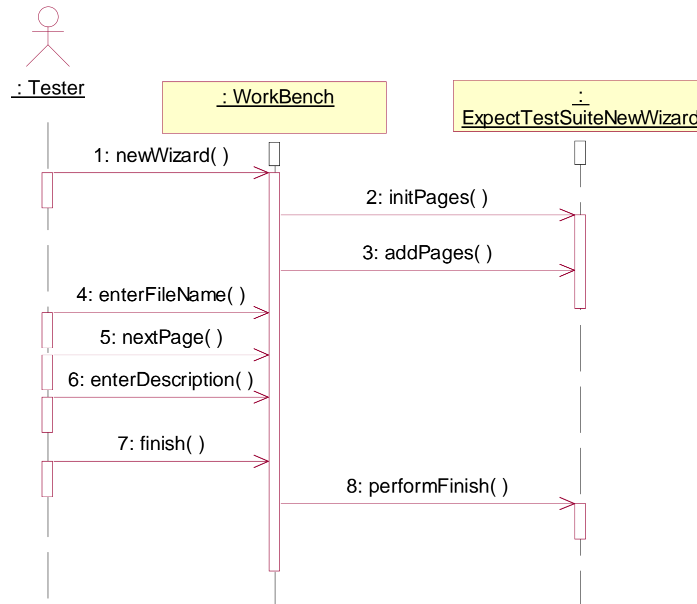


Figure 38: The Tester Creates a New Test Suite

Figure 38 shows the interactions when the Tester creates a new Expect Test Suite:

1. The Tester selects TPTP Expect Test Suite from the File menu in the Eclipse workbench.
2. The Workbench calls the ExpectTestSuiteNewWizard to initialize its pages to display in the wizard dialog.
3. The Workbench calls the ExpectTestSuiteNewWizard to add its pages.
4. The first page is displayed and the Tester is asked to enter a name for the test suite resource file.
5. The Tester presses the Next button.
6. The Tester is asked to enter a description for the test suite.
7. The Tester presses the Finish button. At this stage the resource file is created and the EMF-model for the test suite is created. The registered editor for the Expect Test Suite is displayed.

5.3.4.2 Test Suite Editor

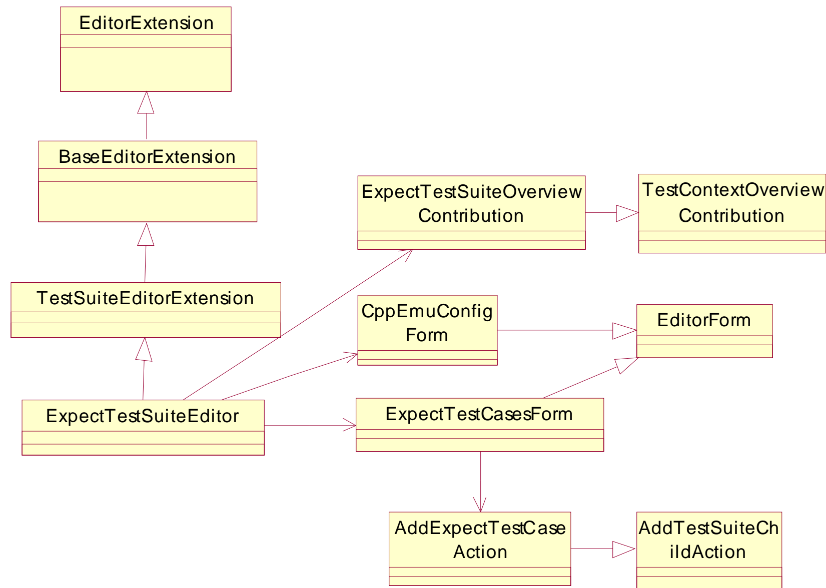


Figure 39: Test Suite Editor Classes

Figure 39 shows a diagram for the classes implementing the Expect Test Suite Editor. The prototype implements the following classes: the ExpectTestSuiteEditor, the ExpectTestSuiteOverviewContribution, the CppEmuConfigForm, the ExpectTestCasesForm and the AddExpectTestCaseAction. The ExpectTestSuiteEditor class is the main class and is registered in the org.eclipse.hyades.editorExtensions extension point, associated with the test suite resource. The classes ExpectTestSuiteOverviewContribution, CppEmuConfigForm and ExpectTestCasesForm implement the three tabs in the Expect Test Suite editor.

The CppEmuConfigForm contains the following fields:

- Host name
- Telnet port
- ClearCase view
- CPP Emulator file (.cppemu)
- Persistent file (.persistent)

- Checkpoint file (.checkpoint)

To get the fields stored with the test suite resource, the values of the fields must be stored in the EMF model related to the test suite resource. In the prototype, the values of the fields are stored in the location parameter for the SUT class for the Test Suite class. The values are stored $-separated in the location parameter. The class diagram in Figure 40 is an extract from the EMF test profile model [73]. The TPFSUT class contains the location field used for storing the SUT configuration values.

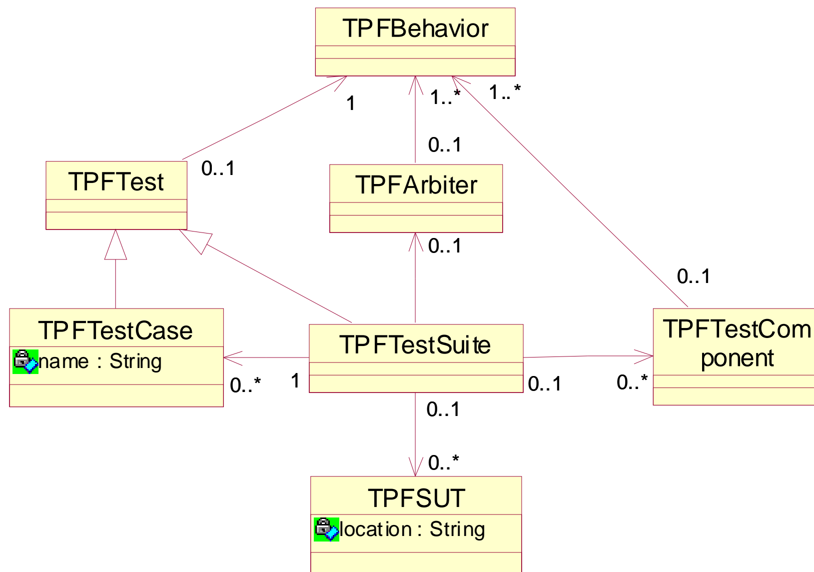


Figure 40: TPTP EMF Test Profile Model

The ExpectTestCasesForm provides a list of test cases for the Expect Test Suite. In the prototype each test case corresponds to an Expect test script. The TPFTestCase class, see Figure 40, is used for storing the path and file name for each Expect test script. The name and path of the Expect test script is stored in the name attribute.

5.3.5 Run Test

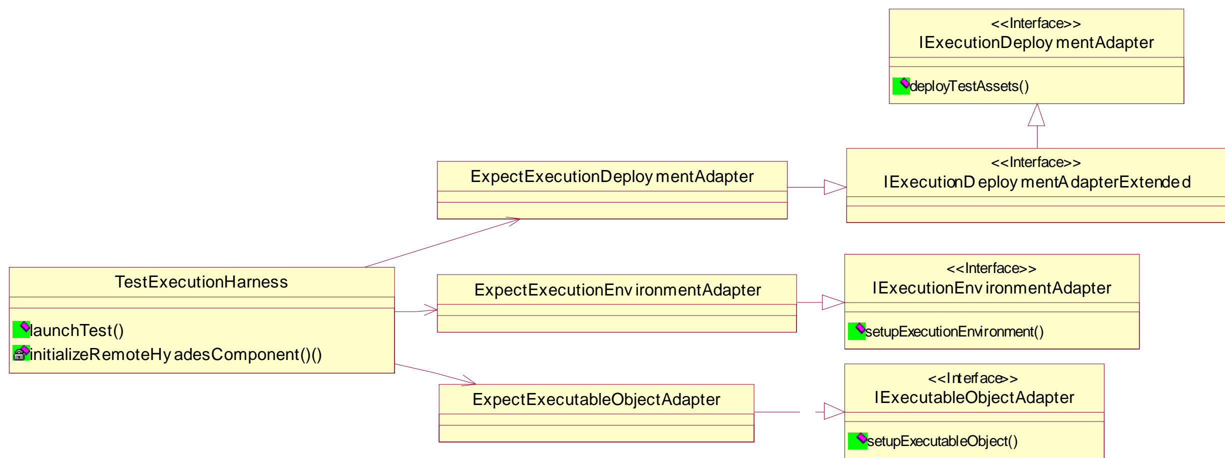


Figure 41: Text Execution Components

The class diagram in Figure 41 shows the execution components registered to be called from the Test Execution Harness during test launch. The prototype implements the three adapter classes: the ExpectExecutionDeploymentAdapter, the ExpectExecutionEnvironmentAdapter and the ExpectExecutablObjectAdapter.

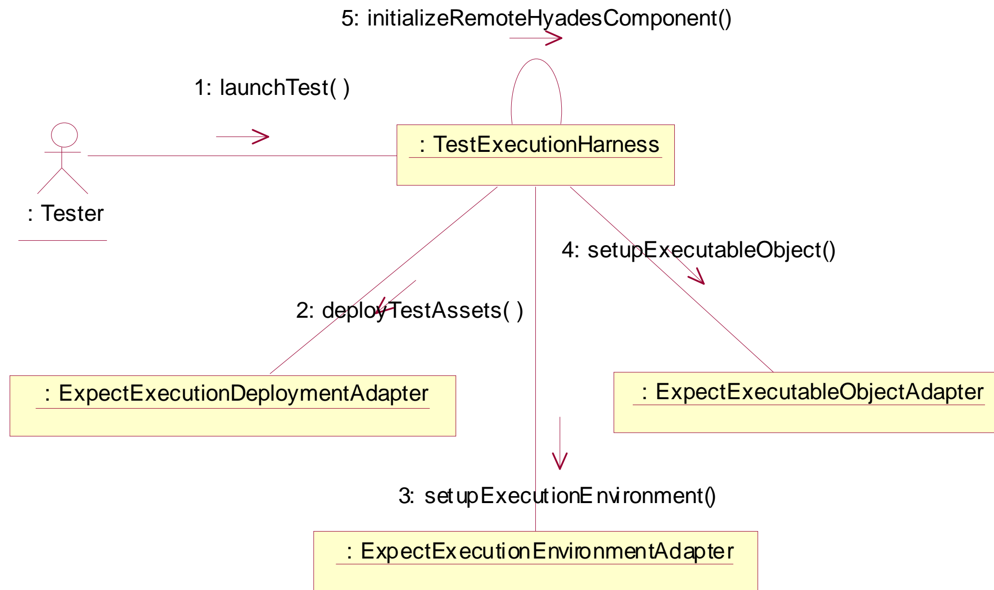


Figure 42: The Tester Starts the Test

Figure 42 shows the interactions when the Tester starts the test. The TestExecutionHarness calls the execution components to perform necessary setup. The server parts of the execution components, see Figure 35, are instantiated in a JVM on the remote test bed machine, and are called to prepare the test execution.

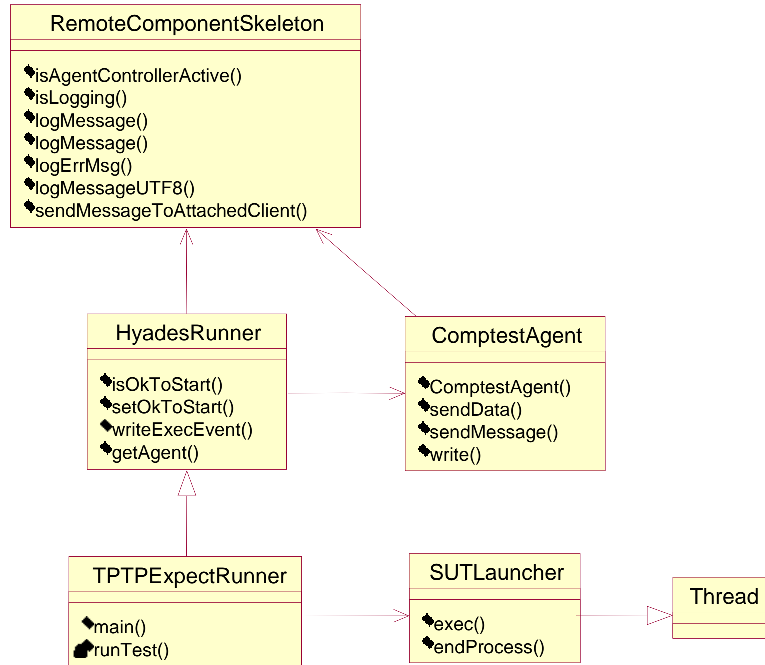


Figure 43: Test Agent

Figure 43 shows a class diagram for the test agent and related classes. The prototype implements the TPTPExpectRunner class and the SUTLauncher class.

The remote test agent is implemented in the TPTPExpectRunner class, which extends the HyadesRunner. The HyadesRunner uses a ComptestAgent, which in turn uses a RemoteComponentSkeleton. The RemoteComponentSkeleton implements the communication with the RAC. The TPTPExpectRunner uses a SUTLauncher to launch and tear down the SUT, which in the prototype case is the CPP Emulator.

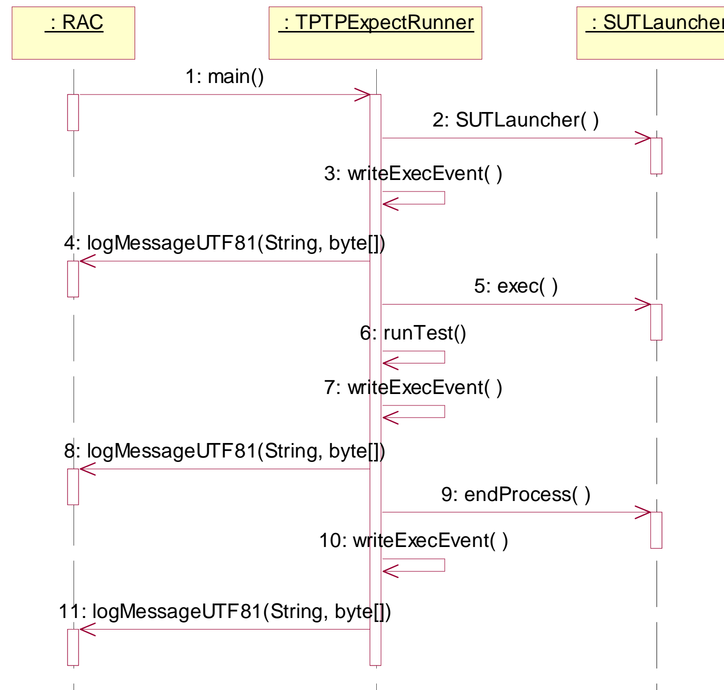


Figure 44: Test Bed Launch

During test bed launch, the remote test agent is started in a separate JVM on the test bed machine. Figure 44 shows the interactions when the RAC starts the test agent:

1. The RAC starts a JVM, which loads the remote test agent, implemented in the TPTPExpectRunner class in the prototype. The TPTPExpectRunner gets all its parameters as arguments to its main method.
2. The TPTPExpectRunner instantiates a SUTLauncher.
3. The TPTPExpectRunner calls its writeExecEvent method to log the start test event.
4. The RAC method logMessageUTF81 is called. This is a Java Native Interface (JNI) call to the C-implementation provided by the RAC.
5. The TPTPExpectRunner calls the exec method in the SUTLauncher to launch the CPP Emulator.
6. The TPTPExpectRunner calls its runTest method to execute the Expect test scripts.

7. The TPTPExpectRunner calls its writeExecEvent method to log events for start/stop of each test script and to log output from the Expect process.
8. The RAC method logMessageUTF81 is called.
9. The TPTPExpectRunner calls the endProcess method in the SUTLauncher to tear down the CPP Emulator.
10. The TPTPExpectRunner calls its writeExecEvent method to log messages that the test execution has finished.
11. The RAC method logMessageUTF81 is called.

The first six parameters passed to the TPTPExpectRunner are CPP Emulator configuration:

1. Host name
2. Telnet port
3. ClearCase view
4. CPP Emulator file (.cppemu)
5. Persistent file (.persistent)
6. Checkpoint file (.checkpoint)

If any of these parameters has not been filled in when the test starts, the string "null" is passed as parameter value instead. After the six parameters specifying the CPP Emulator configuration, parameter seven and above specify the Expect test scripts dynamically.

5.4 Improvements of the Prototype

If a real product is to be developed, based on the prototype implementation, there are a number of functions that should be improved. Some possible improvements are listed in this section.

5.4.1 Permissions of Remote Agents

In the prototype implementation, the ClearCase view name entered by the tester is used for setting the path to the CPP Emulator installation. The RAC must have been started with permissions allowed to access this path. The permissions needed by agents started versus the permissions for the RAC is a general design problem to be solved.

5.4.2 Telnet Port Forwarding

The prototype uses the simplest possible method for enabling port-forwarding to the CPP Node loaded in the CPP Emulator. The command that is used is *cppemu-connect-real-network-port*, which forwards all ports to default numbers. If another CPP Emulator instance is running on the same machine, the port may be occupied, or the Expect scripts may connect to CPP nodes loaded in the other CPP Emulator instead. A better method for port-forwarding should be used. The field for entering a Telnet port number in the test suite editor should be removed.

5.4.3 Test Agent Implemented in C

The goal for the prototype was to implement the remote test agent in C. The reason is that Java is not installed on the Linux machines used for running the CPP Emulators at TietoEnator, for performance reasons. But implementing an agent in C was not possible within the time available for implementing the prototype in the project. The test execution component infrastructure in TPTP is implemented in Java. Furthermore, TPTP launches two JVMs on the remote test machine for the test execution components and for the test agent. Support for implementing a test agent in C or C++ may be supported in future versions of TPTP.

5.4.4 Separate Launching Agent

Another improvement would be to implement a separate launching agent for launching and tearing down the CPP Emulator. The test agent can then call the launching agent to launch the CPP Emulator, before test execution, and call the launching agent to tear down the CPP

Emulator after test execution. With this design, a RAC and launching agent can be installed on machines without Java.

5.4.5 Test Management Integration

The prototype implementation automates testing tasks for a human tester. A tester can prepare, run and evaluate tests from a client Workbench. All resources, as test launch configuration and test execution history, are saved locally in the tester's workspace. But a more realistic scenario is that the test automation is integrated and controlled from a test management system. In that case, the test automation cannot be controlled from a client Workbench, but must be controlled via an API instead. Likewise, the resources cannot be saved in a local workspace, but must be handled by some kind of external repository. Therefore, an interesting new prototype to implement is a so called headless implementation, where the test automation is run without the Eclipse Workbench GUI. Another issue is the handling of the different resources. Coming versions of Eclipse TPTP will have support for storing resources in external repositories.

5.4.6 Port to TPTP 4.x

The prototype is implemented with TPTP 3.2A. In TPTP 4.x there will be additional features such as better feedback for the tester during test execution.

5.4.7 SUT Configuration as a New Resource Type

In the prototype implementation the SUT configuration is connected to the test suite resource. A possibility is to create a new resource type for the SUT configuration, with a related wizard and a related editor. The test suite can then store a link to the SUT configuration instead, which would mean a looser coupling and possibility to have several SUT configurations to easily choose between.

5.4.8 Deployment of Test Scripts

The prototype makes the assumption that the client Workbench and remote test agent share file system. The test scripts are passed from test client to test agent by passing the file path

only. An improvement would be real deployment of the test scripts, that is to transfer the actual files to the test bed machine. This improvement requires TPTP 4.x.

6 Summary and Evaluation

6.1 Market Analysis

The conclusion from the market analysis is that Eclipse TPTP is the product that best matches the requirements of a test tool framework, as defined in this thesis.

Another conclusion is that most commercial products are not designed to be extendable, but to be used as they are. These products do not match the model of a test tool framework, see Chapter 3. There is not a many-to-many relationship between the test tool and the SUT, but a one-to-many relationship, as in Figure 45.

Figure 45: One-to-Many Relationship between a Test Tool and SUTs.

A final conclusion from the market analysis is that it is a difficult task to describe different products in a common way. Different organizations use different testing terminology. The product information available is often of poor quality and written in a sales perspective.

6.2 Prototype

The conclusion from building a prototype in Eclipse TPTP as a proof of concept is that Eclipse TPTP is a test tool framework that can be used for creating an integrated test environment. However, working with Eclipse TPTP requires a great amount of time and effort before getting productive. There is not much documentation available, and therefore an active participation in the Eclipse TPTP community is required. The Eclipse TPTP project is still a young project, but the platform is expected to be much more mature in the coming

future, with more documentation and extended functionality. It will then be a quality test tool framework with a rich set of functions for building new test tools.

6.3 Discussion

6.3.1 Pros and Cons of a Common Framework

Many advantages can be seen in using a test tool framework with a common infrastructure that enables integration with different test tools and with different SUTs. The schematic model we have used for a test tool framework in this thesis is repeated in Figure 46.

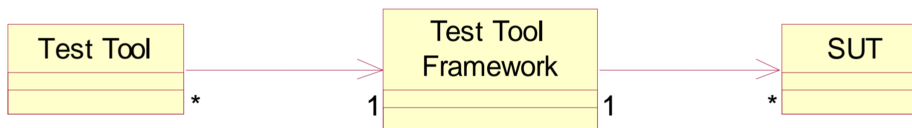


Figure 46: Integration by Means of a Common Test Tool Framework.

The main advantage with a common test tool framework is that an integrated test environment can be achieved, with benefits for different groups of users, see Section 3.3. Using a common infrastructure assures interoperability between different tools. Common solutions can be used by different tools, instead of re-inventing them. Instead of fragmented solutions where every single test environment develops its own solutions, there are advantages in centralizing common functions, so that different tools can re-use them. Examples of areas where common functions should be shared are infrastructure for test deployment and execution as well as collection of different data. Runtime monitoring of test execution as well as different SUT metrics are other examples of functionality that should be possible to re-use without modification from one test environment to another. Instead of specialized log viewers for every single SUT, for example, it should be possible to agree on a common solution to use globally.

The over-all advantages with centralized functions shared by different test tools are higher quality and reduced costs. Higher quality is achieved by common tools that can be shared by a

broader community. Experiences from many people and different organizations can be utilized to ensure correct functionality and high quality. Reduced costs are a direct consequence of re-use; there will simply be a reduced number of test tools to create and maintain. Another aspect is that looser coupling between the test tools and the SUT reduces test tool maintenance when external interfaces of the SUT changes.

The framework delivered by the Eclipse TPTP project very closely matches the model of a test tool framework described in this thesis, with a common infrastructure and other centralized testing related functions. A great strength with the Eclipse TPTP project is the cooperation between several software vendors, sharing experience from many people. Michael G. Norman et al summarizes the problem with re-inventing infrastructures in a paper describing the objectives of the Hyades project [28]: “80% of the effort that testers and developers spend today is on making testing possible, and only 20% focuses on making testing meaningful. 80% of the effort Automated Software Quality (ASQ) tool vendors spend today duplicates the work of others, recreating an infrastructure to enable testing and debugging activities. Only 20% of their work produces new function that’s visible and valuable to testers and developers.”

Commercial products with remote testing capabilities have their own proprietary infrastructures for integration of different test tools and with the SUT. Michael G. Norman et al [28] also describes this problem: “More significantly, the “plumbing” required to effectively drive and monitor the SUT is enormous and its maintenance expensive and highly dependent on obscure details of the runtime environment.”

But there are not only advantages with a common framework shared by different test tools. A problem pointed out by Mats Berglund [3], who has a great amount of experience from issues as this from previous work at Ericsson, is that a test solution with different test tools integrated in a common framework requires a party that takes the overall responsibility. Changes to the platform may imply changes to every single test tool dependent on it. The question is who should be responsible for coordination of the platform and the different tools, to ensure a constant high quality. Changes are inevitable and it might be a big risk if the

whole test environment is dependent on a single framework. A fragmented solution with one-to-one or one-to-many relations between the test tool and the SUT may therefore be preferable for a company as Ericsson. The schematic model we have used for specialized test tools in this thesis is repeated in Figure 47.

Figure 47: Specialized Test Tool

There is a choice between an open flexible solution and many specialized tools. A framework gives a more integrated environment, but may also imply a bigger risk. Many specialized tools means less integration and a number of disadvantages, but more safety and lower risks.

6.3.2 Standardization

Experiences gained from working with the project described in this thesis, and also from previous projects, is that homemade not-invented-here solutions are dominating the testing world of today. There should be great potential and many advantages in agreeing in common standards and solutions. The UML 2 Testing Profile [33] seems very promising and should be a great step forward. A seemingly simple matter like agreeing on a common terminology to use for different verdicts has probably enormous impact on the testing world as a whole. Instead of using its own terminology, every test environment can use the same *pass, fail, inconclusive, error.* Another interesting standardization is the Common Base Event [34]. Logs created in server-based and distributed systems are very important for operation, maintenance, fault localization etc. The log events normally contain timestamp, text message, source information and category such as *information, warning* and *error.* There should be great benefits in agreeing on a common format for these logs instead of each system using its own. With a common format there can be one single log viewer for all different logs created by all different systems. Other standards that seem important to use are UML and XML.

6.3.3 Open Source

When building the prototype we got some experience from using an open source product. There are both advantages and disadvantages with open source. Free licenses may be seen as the main advantage with open source. Free licenses may lead to greatly reduced costs for a company. An interesting point made at a meeting with Ericsson [2] was that free licenses are not at all important when Ericsson is making a decision in which technical solution to choose for the future. License costs are only a small fragment of the total cost for investing in a specific solution.

Our experience is that another important issue is documentation. Open source does not say anything about documentation. When you work with open source, there might not be any system or design documentation at all. And the little documentation available might be out-of-date, incomplete or otherwise of bad quality. It might require a great amount of time and effort to get productive. You are also dependent on other people to help you. The good thing is that open source projects normally have an active community with a good atmosphere for sharing information between members. But it feels like a big risk to be completely dependent on other people to answer your questions.

In the long term, building on open source should be preferable compared to designing a solution from scratch. By building on existing solutions you should be able to produce much more functionality than if starting from zero. There may be many functions that you get for free. In a short project, however, it might require too much time to find out how all functionality works. A well documented commercial product may then be preferable.

Our conclusion from working with the open source project Eclipse TPTP is that the main advantage with open source is that an open source project is a forum for cooperation outside company borders. An open source project is a unique opportunity for sharing experience from different organizations. Cooperation between companies is necessary in works with standardization and should also be very important for unifying software testing standards and techniques.

6.3.4 Eclipse-Based Products

The interest for the Eclipse platform and TPTP increases. When this thesis is written there are already a few commercial products based on Eclipse TPTP available. Out of the products studied in the market analysis in this project, the Scapa Test and Performance Platform 3.1, see Section D.5, and Testing Technologies TTworkbench, see Section D.7, are two examples of currently available Eclipse-based products. Scapa Technologies is one of the contributors of source code to Eclipse TPTP and their Test and Performance Platform extends the TPTP open source. The TTworkbench from Testing Technologies includes Eclipse plug-ins for editing TTCN-3 scripts and for compiling TTCN-3 modules into test executables. There will probably be several other TPTP based products available in the coming future, when the TPTP platform gets more mature. Eclipse integration may also be a requirement from different test tool customers who want to use the same Integrated Development Environment (IDE) for as many different software development tasks as possible. There are many advantages in being able to use the same IDE for both coding and testing an application, for example.

Eclipse TPTP should be very interesting to use as a framework for a company who wants to build an integrated test environment. Different plug-ins can be bought from different software vendors, which also should be important for spreading the risks and increasing the competition. The testing domain is too big for one single vendor to cover. One vendor may be specialized in unit testing while another may be specialized in TTCN-3 protocol testing. By using Eclipse as a base, it should be possible to keep an integrated test environment even though different tools (plug-ins) are bought from different vendors. There are, of course, not only advantages with a solution like this, as was discussed in Section 6.3.1. If the common platform requires updates that affect all dependent tools from different vendors, updating the tools will be more problematic than if all tools are coming from a single vendor, for example.

Another important aspect is that Eclipse integration may be provided at different levels, for example user interface level, file interoperability level and runtime interoperability level. A software vendor may sell a product marked as “Eclipse ready” even if the product in question

just provides a plug-in for a very small part of the entire system. The front-end user interface may be ported to the Eclipse design while the back-end parts are kept in a proprietary design. Common user interface is important for a uniform look and feel, and to achieve this it is important that the different plug-ins follow common design guides. Another level of integration is file interoperability, where integration is achieved if one tool can read the output from another tool. It is important that the plug-ins follow the standards and models used in Eclipse, and not its own home-made. An integrated test environment also requires that the different tools can communicate with each other in a runtime environment. A workbench launch for a distributed telecom system may include a number of tools that must run simultaneously, as for example test driver, test execution monitor and different runtime log/trace monitors. Each tool may work well when run stand-alone, but to be able to run all at once probably require additional actions. An integrated test environment should be independent of test script languages used. An interesting scenario that would require good integration, both in client and server parts, would be the possibility to run a test suite with scripts in different languages, such as Java, C++, Perl and TTCN-3, for example. Good integration with Eclipse TPTP will not come for free, but requires that the different plug-ins follow the Eclipse TPTP design. The goal should be the highest level of integration, that is runtime interoperability.

7 Conclusion

The purpose of the project described in this thesis was to study different test tool frameworks that can be used for creating an integrated test environment. The goal of the project was to find a product that TietoEnator could use in future projects. The method used was to first specify some basic requirements for a test tool framework, then carry out a market analysis to find candidate products, and finally build a prototype as a proof of concept for the product that best matched the specified requirements. The requirements include infrastructure for remote test bed launch and execution as well as centralized functions for building new test tools. We found two candidates for test tool frameworks in the market analysis: Eclipse TPTP and STAF. The conclusion from the market analysis was that the product that best fulfilled the stated requirements was Eclipse TPTP. Other results from the market analysis are a thorough study of a number of products, including product descriptions and evaluations based on common comparison points. A functioning prototype was built using Eclipse TPTP. The prototype makes it possible for a tester to create a test configuration to be run on a remote machine. The test configuration includes a selection of test scripts, test bed configuration and SUT configuration. When the tester starts the test from the client Workbench, the test bed is automatically launched on the remote machine, the test scripts are executed, and finally the test bed is torn down. The tester can view the test result in a test execution history in the client Workbench. A final conclusion from the project is that there remains some work with additional functionality and documentation before Eclipse TPTP is mature to be used in real projects, but that Eclipse TPTP has good potential for being a quality test tool framework with a rich set of functions in the future.

References

[1] Meeting at Ericsson Linköping, February 3, 2005. Attendees: Mats Berglund (testing expert at Ericsson), Lars-Lundegård, Per Johansson, Henrik Wallinder.

[2] Exam thesis presentation at Ericsson Linköping, May 9, 2005. Attendees: Mats Berglund, Per Emanuelsson, Patrik Nandorf, Conor White Ericsson Linköping, Lars Lundegård, Lars Ohlen, Johan Andersson, Per Johansson, Henrik Wallinder TietoEnator Karlstad.

[3] Telephone meeting, Exam thesis draft review, May 17, 2005. Attendees: Mats Berglund, Ericsson Linköping, Lars Lundegård, Per Johansson, Henrik Wallinder TietoEnator Karlstad.

[4] Victor Ferraro-Esparza, Michael Gudmandsen, Kristofer Olsson, *Ericsson Telecom Server Platform 4*, Ericsson Review No. 3, 2002.

[5] Göran Ahlforn, Erik Örnulf, *Ericsson's family of carrier-class technologies*, Ericsson Review No. 4, 2001.

[6] Lars-Örjan Kling, Åke Lindholm, Lars Marklund and Gunnar B. Nilsson, *CPP – Cello Packet Platform*, Ericsson Review No. 2, 2002.

[7] UMTS, *UMTS World*, http://www.umtsworld.com/technology/technology.htm, May 24, 2005.

[8] Glenford J. Myers, *The Art of Software Testing,* page 5, Wiley Interscience, 1979.

[9] J. Case, M. Fedor, M. Schoffstall, J. Davin, *RFC 1157*, *A Simple Network Management Protocol (SNMP)*, May 1990.

[10] Andrew S. Tanenbaum, *Computer Networks*, Third Edition, Prentice Hall, 1996.

[11] ANSI/IEEE Std 729-1983, *Glossary of Software Engineering Terminology*, New York:IEEE, 1983.

[12] ApTest, *Software Testing Glossary*, http://www.aptest.com/glossary.html, February 9, 2005.

[13] Bill Hetzel, *The Complete Guide to Software Testing*, Second Edition, QED Information Sciences, Inc., 1988.

[14] *Catalog of OMG CORBA®/IIOP® Specifications,* OMG, http://www.omg.org/technology/documents/corba_spec_catalog.htm, February 24, 2005.

[15] *COM: Component Object Model Technologies*, http://www.microsoft.com/com, Microsoft, February 9, 2005.

[16] Kent Beck, *Extreme Programming Explained*, Addison-Wesley, Harlow, USA, 1999.

[17] W. Yeong, T. Howes, S. Kille, *RFC 1777, Lightweight Directory Access Protocol,* March 1995.

[18] Object Mentor, Inc, *Test Driven Development*, http://www.objectmentor.com/writeUps/TestDrivenDevelopment, February 9, 2005.

[19] *SEA Architecture* (15553-CRL 119 007, Rev A), Ericsson internal document.

[20] *SS7 Protocol Suite*, http://www.protocols.com/pbook/ss7.htm, PROTOCOLS.COM, February 24, 2005.

[21] RiceConsulting, *Software Testing and Quality Glossary*, http://www.riceconsulting.com/new/index.php?option=displaypage&Itemid=116&op=page&SubMenu=&secret=-1, February 9, 2005.

[22] TestDriven.com, http://www.testdriven.com, February 9, 2005.

[23] The Testing Standards Working Party, Living Glossary, http://www.testingstandards.co.uk/living_glossary.htm, February 9, 2005.

[24] University of Oulu, Eletrical and Information Engineering, *Glossary of Vulnaribility Testing Terminology*, http://www.ee.oulu.fi/research/ouspg/sage/glossary/, February 9, 2005.

[25] *Eclipse*, http://www.eclipse.org, February 21, 2005

[26] Object Technology International, Inc, *Eclipse Platform Technical Overview*, February 2003, http://www.eclipse.org/articles, February 21, 2005.

[27] Eclipse test & performance tools platform project, Project Descriptions, http://www.eclipse.org/test-and-performance/index.html, February 21, 2005.

[28] Michael G. Norman, Sam Guckenheimer, Harm Sluiman, Marc R Erickson, Sara Mariani, *The Hyades Project Automated Software Quality for Eclipse*, December 10, 2002.

[29] *Eclipse test & performance tools platform project, Frequently Asked Questions*, http://www.eclipse.org/test-and-performance/index.html, February 24, 2005.

[30] *Building a Custom Test Execution Environment*, February 28, 2005, TPTP Testing Tools Project, Documentation, http://www.eclipse.org/tptp/index.html, February 28, 2005.

[31] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose, *Eclipse Modeling Framework,* Addison-Wesley, 2004.

[32] Eric Clayberg, Dan Rubel, *Eclipse Building Commercial-Quality Plug-ins*, Addison-Wesley, 2004.

[33] *UML 2.0 Testing Profile Specification*, OMG, April 2004, http://www.omg.org/technology/documents/modeling_spec_catalog.htm, February 24, 2005.

[34] David Ogle, Eric Labadie, James Schoech, Heather Kreger, Mandy Chessell, Mike Wamboldt, Abdi Salahshour, Bill Horn, Jason Cornpropst, John Gerken, *Canonical Situation Data Format: The Common Base Event V1.0.1*, November 04, 2003, http://www.eclipse.org/hyades/, February 24, 2005.

[35] Catherine Griffin, *Using EMF*, December 9 2002, http://www.eclipse.org/articles/index.html, February 25, 2005.

[36] *Unified Modeling Language*, OMG, http://www.uml.org/, February 24, 2005.

[37] *Extensible Markup Language (XML)*, W3C, http://www.w3.org/XML, February 24, 2005.

[38] *XML Metadata Interchange (XMI) Specification*, Version 2.0, OMG, May 2003, http://www.omg.org/docs, February 24, 2005.

[39] *Java™ Virtual Machine Profiler Interface (JVMPI)*, http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html, February 24, 2005.

[40] *JVM™ Tool Interface Version 1.0*, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html, February 24, 2005.

[41] *Java Management Extensions (JMX)*, http://java.sun.com/products/JavaManagement/, February 24, 2005.

[42] *Scapa Technologies*, http://www.scapatech.com/home.html, February 28, 2005.

[43] Scapa Technologies, *Scapa Test and Performance Platform 3.1*, Product Datasheet, http://www.scapatech.com/home.html, February 28, 2005.

[44] IBM, *Rational TestManager*, http://www-306.ibm.com/software/awdtools/test/manager, Mars 1, 2005.

[45] IBM, *Rational Test RealTime*, http://www-06.ibm.com/software/awdtools/test/realtime/, Mars 9, 2005.

[46] IBM, Rational Testing Products, Rational TestManager, *User's Guide*, Rational Software Corporation, June, 2003, ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/testing.html, Mars 1, 2005.

[47] IBM, Rational Testing Products, Rational TestManager, *Extensibility Reference*, Rational Software Corporation, June, 2003,

ftp://ftp.software.ibm.com/software/rational/docs/documentation/manuals/testing.html, Mars 1, 2005.

[48] ISO, *ISO/IEC 9646-3:1998*, Information technology -- Open Systems Interconnection -- Conformance testing methodology and framework -- Part 3: The Tree and Tabular Combined Notation (TTCN), Published ISO Standard.

[49] ETSI, *TTCN-3 TRI*, ETSI ES 201 873-5 V1.1.1 (2003-02), Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI), Published ETSI Standard.

[50] ETSI, *TTCN-3 TCI*, ETSI ES 201 873-6 V1.1.1 (2003-07), Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI), Published ETSI Standard.

[51] ISO, *PICS/PIXIT*, http://www.iec.org/online/tutorials/ttcn/topic03.html, Mars 2, 2005.

[52] JUnit.org, *JUnit*, http://www.junit.org/index.htm, Mars 3, 2005.

[53] ETSI PTCC, Protocol & Testing Competence Centre, *Latest TTCN-3 Specifications*, ETSI ES 201 873, http://www.etsi.org/ptcc/ptccttcn3.htm, Mars 9, 2005.

[54] Telelogic, http://www.telelogic.com, Mars 9, 2005.

[55] Telelogic, *Telelogic TAU*, http://www.telelogic.com/products/tau/index.cfm, Mars 9, 2005.

[56] Telelogic, Telelogic TAU/Tester Technical Integration Documentation, provided with Telelogic TAU/Tester, September 2004.

[57] Danet, *Test-automation systems and services*, http://www.danet.com, Mars 9, 2005.

[58] International Telecommunication Union (ITU), Telecommunication Standardization Sector of ITU (ITU-T), Z.120, *Annex B: Formal semantics of Message Sequence Charts*, SDL Forum Society, Publication – Standards, http://www.sdl-forum.org/Publications/Standards.htm, Mars 9, 2005.

[59] European Telecommunications Standards Institute (ETSI), http://www.etsi.org, Mars 9, 2005.

[60] *The Expect Home Page*, http://expect.nist.gov/, Mars 14, 2005.

[61] *Software Testing Automation Framework (STAF)*, http://staf.sourceforge.net/index.php, Mars 14, 2005.

[62] *Frequently Asked Questions about STAF, STAX, and STAF services*, http://staf.sourceforge.net/current/STAFFAQ.htm, Mars 14, 2005.

[63] C. Rankin, *The Software Testing Automation Framework*, http://www.research.ibm.com/journal/sj/411/rankin.html, IBM, June 8, 2005.

[64] *GNU Lesser General Public License*, http://www.gnu.org/copyleft/lesser.html, Mars 14, 2005.

[65] *OpenTTCN*, http://www.openttcn.com, Mars 17, 2005.

[66] *OpenTTCN Tester for TTCN-3*, http://www.openttcn.com/Sections/Products/OpenTTCN3, Mars 17, 2005.

[67] *OpenTTCN XPress for TTCN-3*, http://www.openttcn.com/Sections/Products/Xpress3, Mars 17, 2005.

[68] ITU-T, X.683 (12/97), Abstract Syntax Notation One (ASN.1), Published ITU-T Standard.

[69] *Testing Technologies*, http://www.testingtech.de, Mars 18, 2005.

[70] Fraunhofer FOKUS, http://www.fokus.gmd.de/home, Mars 18, 2005.

[71] ISO/IEC 14750:1999, Information technology -- Open Distributed Processing -- *Interface Definition Language*, Publiched ISO/IEC Standard.

[72] *Citrix*, http://www.citrix.com/lang/English/home.asp, April 8, 2005.

[73] Hyades Data Models, http://eclipse.org/tptp/home/archives/hyades/data_models/index.htm, May 12, 2005.

[74] Jan Tretmans, *An Overview of OSI Conformance Testing*, Formal Methods & Tools group, University of Twente, January 25, 2001, http://www.cs.auc.dk/~kgl/TOV03/iso9646.pdf, May 15, 2005.

[75] Wikepedia, http://en.wikipedia.org, May 18, 2005.

[76] Open Process Framework (OPF), http://www.donald-firesmith.com/index.html?Glossary, May 15, 2005.

[77] Ericsson AB, *Using the CPPemu – R2A, User's* Guide, Ericsson internal document, 198 17-CRL 119 071 Uen C.

[78] Ericsson AB, *CPP Emulator Training, Hands-on Course R2,* Ericsson internal document, January 12, 2004.

[79] IBM, IBM Software – Rational ClearCase, http://www-306.ibm.com/software/awdtools/clearcase/, June 8, 2005.

A Definitions

Architecture

An architecture is “the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution” [11].

Black-box testing

“Functional test case design: Test case selection that is based on an analysis of the specification of the component without reference to its internal workings” [24].

Branch coverage

“Metric of the number of branches executed under test; "100% branch coverage" means that every branch in a program has been executed at least once under some test (also link coverage)” [24].

Capture/Play-back

Functionality for automatically creating test scripts by recording events, which can be automatically repeated with a play-back function. Example of events are user-interface interaction or HTTP requests.

Comparator

A function or tool for comparing the response from the SUT against expected results as specified by the test.

Component

“A minimal software item for which a separate specification is available” [22].

Configuration Testing

"The system testing of different variations of the application against its configurability requirements" [76].

Conformance Testing

"The process of testing that an implementation conforms to the specification on which it is based" [23].

Debug

The process of locating the source of different defects.

Driver

See Test Driver.

Emulator

"A device, computer program, or system that accepts the same inputs and produces the same outputs as a given system" [12].

Execution History

See Test Execution History

Framework

"The software environment tailored to the needs of a specific domain. Frameworks include a collection of software components that programmers use to build applications for the domain the framework addresses. Frameworks can contain specialized APIs, services, and tools, which reduce the knowledge a user or programmer needs to accomplish a specific task" [12].

Functional Testing

"Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions" [12].

Glass Box Testing

See White Box Testing.

Implementation Under Test (IUT)

The actual components within the SUT that are the target test objects for the current test.

Instrumentation

"Devices or instructions installed or inserted into hardware or software to monitor the operation of a system or component" [24].

Integration Testing

"Testing performed to expose faults in the interfaces and in the interaction between integrated components" [76].

Interoperability

"The ability of systems, units, or forces to provide services to and accept services from other systems, units or forces and to use the services so exchanged to enable them to operate effectively together" [75].

Interoperability Testing

Testing for interoperability between systems or components, see Interoperability.

Interworking Testing

See Interoperability Testing.

Load Testing

See Performance Testing.

Log

Listing that contains a record of events often stored in a file. The events can be generated by test scripts or by the SUT, producing different type of logs. Often categorized by tags such as information, warning, error. Standardized format in Common Base Event (CBE) [34].

Monitor

The activity of observing different aspects of the test execution in real-time. Examples are real-time monitoring of test execution and monitoring of logs created by the SUT.

Path coverage

“Metric applied to all path-testing strategies: in a hierarchy by path length, where length is measured by the number of graph links traversed by the path or path segment; e.g. coverage with respect to path segments two links long, three links long, etc. Unqualified, this term usually means coverage with respect to the set of entry/exit paths. Often used erroneously as synonym for statement coverage” [24].

Performance Testing

“Testing conducted to evaluate the compliance of a system or component with specified performance requirements. Often this is performed using an automated test tool to simulate large number of users. Also known as "Load Testing"” [12].

Profiling

The process of analyzing the performance, resource utilization, or execution of a running program or process. Normally requires source code instrumentation. Examples of profiling analysis are: code coverage analysis, execution time analysis and memory usage analysis.

Protocol Testing

See Protocol Conformance Testing.

Protocol Conformance Testing

"Testing conducted to verify that an implementation of a protocol conforms to the specification of the protocol" [74].

Regression Testing

"Retesting a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made" [12].

Robustness Testing

"The testing that attempts to cause failures involving how the system behaves under invalid conditions (e.g., unavailability of dependent applications, hardware failure, and invalid input such as entry of more than the maximum amount of data in a field)" [76].

Simulator

"A device, computer program or system used during software verification, which behaves or operates like a given system when provided with a set of controlled inputs" [22].

Smoke Test

"A quick-and-dirty test that the major functions of a piece of software work. Originated in the hardware testing practice of turning on a new piece of hardware for the first time and considering it a success if it does not catch on fire" [12].

Software Tool

A software tool is "a computer program used to help develop, test, analyze, or maintain another computer program or its documentation" [11].

Stress Testing

"Testing in which a system is subjected to unrealistically harsh inputs or load with inadequate resources with the intention of breaking it" [24].

"Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements" [23].

Structural Testing

See White Box Testing.

Stub

"A skeletal or special-purpose implementation of a software module, used to develop or test a component that calls or is otherwise dependent on it" [22] .

System Testing

"Testing that attempts to discover defects that are properties of the entire system rather than of its individual components" [12].

System Under Test (SUT)

"The real open system in which the Implementation Under Test (IUT) resides" [24].

Test

"A set of one or more test cases" [24].

Test Artifact

See Test Asset.

Test Asset

A test asset is any resource, normally persisted to file, that is either used as input to a test or result from a test execution, e.g. test plan, test suite, test case, test script, test execution history, trace, log, profiling information, statistical data or test report.

Test Architecture

See Test Environment.

Test Bed

"An environment containing the hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test" [24].

Test Campaign

See definition of Test.

Test Case

"A specific set of test data along with expected results for a particular test objective, such as to exercise a program feature or to verity compliance with a specific requirement" [11].

Test Comparator

"A test tool that compares the actual outputs produced by the software under test with the expected outputs for that test case" [22].

Test Cycle

"A formal test cycle consists of all tests performed. In software development, it can consist of, for example, the following tests: unit/component testing, integration testing, system testing, user acceptance testing and the code inspection" [24].

Test Data

Test input data used in association with different test cases. Used to generate desired stimuli to the SUT. Test data may be generated automatically by different tools, for example different properties of virtual users.

Test Driver

"A program or testing tool used to execute and control testing. Includes initialization, data object support, preparation of input values, call to tested object, recording and comparison of outcomes to required outcomes" [24].

Test Environment

"A description of the hardware and software environment in which the tests will be run, and any other software with which the software under test interacts when under test including stubs and test drivers" [24].

Test Execution History

The result of a test execution containing the test cases executed with verdicts.

Test Framework

A framework for making it easy to add new tests and to run various suites of tests.

Test Harness

"A program or test tool used to execute tests. Also known as a Test Driver" [12]. See also Test Tool.

Test Item

"A software item which is an object of testing" [24].

Test Plan

"A document describing the scope, approach, resources, and schedule of intended test activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning" [24].

Test Procedure

"A document providing detailed instructions for the execution of one or more test cases" [22].

Test Script

"Commonly used to refer to the instructions for a particular test that will be carried out by an automated test tool" [12].

"Commonly used to refer to the automated test procedure used with a test harness" [22].

Test Suite

A set of test cases and/or test scripts that are related to a particular function or feature of an application.

Test Tool

A test tool is a computer program used to test another computer program. Based on the definition of Software Tool. See also Test Harness.

Test Tool Framework

A test tool framework is a framework for creating an integrated test environment. In this document, there are two main purposes with a test tool framework:

1. To make it easy for test tools to connect and communicate with the SUT.
2. To make it easy to create new test tools.

Testing

"Testing is the process of executing a program or system with the intent of finding errors" [8].

Trace

Listing of the path of execution in a system or between systems. May require instrumentation of source code or use of probes to collect the necessary data. Trace information can be collected at different levels, for example method calls between classes or messages sent via an interface between two network components. Trace data may also be gathered from system stacks and heap information. Trace information can be represented in textual form or in graphical form, for example as UML sequence diagram.

Validation

"Determination of the correctness of the products of software development with respect to the user needs and requirements" [23].

Verdict

"Verdict is the assessment of the correctness of the SUT. Test cases yield verdicts. Verdicts can also be used to report failures in the test system. Predefined verdict values are pass, fail, inconclusive and error. Pass indicates that the test behavior gives evidence for correctness of the SUT for that specific test case. Fail describes that the purpose of the test case has been violated. Inconclusive is used for cases where neither a Pass nor a Fail can be given. An Error verdict shall be used to indicate errors (exceptions) within the test system itself. Verdicts can be user-defined. The verdict of a test case is calculated by the arbiter." [33]

Verification

"The process of evaluating a system or component to determine whether the products of the given development phase satisfy the conditions imposed at the start of that phase" [23].

White-box testing

"Testing based on an analysis of internal workings and structure of a piece of software. Includes techniques such as Branch Testing and Path Testing. Also known as Structural Testing and Glass Box Testing. Contrast with Black Box Testing" [12].

B Acronyms and Abbreviations

AAA	Authentication, Authorization and Accounting
ANSI	American National Standards Institute
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
ASQ	Automatic Software Quality
ATM	Asynchronous Transfer Mode
ATS	Abstract Test Suite
AXE	An open architecture, Ericsson's communications platform. AXE is a system for computer-controlled digital exchanges that constitute the nodes in large public telecommunications networks. AXE is the basis for Ericsson's wire line and mobile systems.
BER	Basic Encoding Rules
BSC	Base Station Controller
BSS	Base Station Subsystem/System
BTS	Base Transceiver Station
CADE	CPP Application Development Environment
CBE	Common Base Event
CD	Codec and Decoding
CDMA	Code Division Multiple Access
CDMA2000	Code Division Multiple Access 2000
CH	Component Handling
CLI	Command Line Interface
CM	Configuration Management
COM	Component Object Model
CORBA	Common Object Request Broker Architecture

CPP	Connectivity Packet Platform
CPU	Central Processing Unit
DOS	Disk Operating System
E2E	End-to-end
ECU	Electronic Control Unit
EMF	Eclipse Modeling Framework
EPL	Eclipse Public License
ETS	Executable Test Suite
ETSI	European Telecommunications Standards Institute
FM	Fault Management
GNU	GNU's Not Unix
GPB	General Processor Board
GPL	General Public License
GPRS	General Packet Radio Service
GSM	Global System for Mobile communications
GSN	GPRS Support Node
GUI	Graphical User Interface
HCE	Hyades Collection Engine
HD	Home Agent
HLR	Home Location Register
HP	Hewlett Packard
HTTP	HyperText Transfer Protocol
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
IDL	Interface Description Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical & Electronic Engineers
IIOP	Internet Inter-ORB Protocol

IP	Internet Protocol
IP&C	IP & Connectivity
ISO	International Organization for Standardization
ITU	International Telecommunications Unit
IUT	Implementation Under Test
J2EE	Java 2 Enterprise Edition
JDK	Java Development Kit
JDT	Java Development Tooling
JMX	Java Management Extensions
JNI	Java Native Interface
JRE	Java Runtime Environment
JVM	Java Virtual Machine
JVMPI	Java Virtual Machine Profiler Interface
JVMTI	JVM Tool Interface
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
LGPL	Lesser GPL
MCN	Mobile Core Network
MGCF	Media Gateway Control Function
MGW	Media Gateway
MPH	Message Protocol Handler
MS	Mobile Station
MSC	Message Sequence Chart (MSC-96)
MSC	Mobile services Switching Center
MTS	Methods for Testing and Specification
O&M	Operation and Maintenance
OMG	Object Management Group
ORB	Object Request Broker

OS	Operating System
OSI	Open Systems Interconnection
PA	Platform Adapter
PC	Personal Computer
PCO	Point of Control and Observation
PDSN	Packet Data Serving Node
PER	Packet Encoding Rules
PICS	Platform for Internet Content Selection
PIXIT	Protocol Implementation Extra Information
PL	Platform Layer
PM	Performance Management
PSTN	Public Switched Telephone Networks
PTCC	Protocol & Testing Competence Centre
QoS	Quality of Service
R&D	Research & Development
RAC	Remote Agent Controller
RBS	Radio Base Station
RNC	Radio Network Controller
RTL	Runtime Layer
RTPAR	Rational Test Asset Parcel
RTS	Runtime System
RXI	Radio access network aggregator / IP router
SA	System Adaptor
SCCI	Source Code Control Integration
SDK	Software Development Kit
SDL	Specification and Description Language
SEA	Simulated Environment Architecture
SNMP	Simple Network Management Protocol

SS7	Signaling System #7
STAF	Software Testing Automation Framework
STAX	STAF Execution Engine
SUT	System Under Test
TAG	Telephony Access Gateway
TCI	TTCN-3 Control Interface
TCL	Tool Command Language
TDM	Time Division Multiplexing
TE	TTCN-3 Executable
TelORB	Telecommunications Object Request Broker
TM	Test Management
TMC	Test Management and Control
TMN	Telecommunications Management Network
TPTP	Eclipse Test & Performance Tools Platform Project
TRI	TTCN-3 Runtime Interface
TSO	Test Suite Operation
TSP	Ericsson Telecom Server Platform
TTCN	The Testing and Test Control Notation
U2TP	UML 2 Test Profile
UE	User Equipment
UML	Unified Modeling Language
VB	Visual Basic
VoIP	Voice over IP
WCDMA	Wide-band CDMA
WLAN	Wireless LAN
WSN	WLAN Serving Node
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

XP	eXtreme Programming
XSD	XML Schema Definition

C Introduction to TTCN-3

Testing and Test Control Notation generation 3 (TTCN-3) is an internationally standardized, multi-purpose test language. TTCN-3 was standardized by European Telecommunications Standards Institute (ETSI) [59] and Telecommunication Standardization Sector, of International Telecommunication Union (ITU), (ITU-T) [58]. TTCN-3 is a formal, dedicated and independent language, making it precise and distinct (interpreted in the same way by everyone). TTCN-3 is generally focused on black box testing, but not only intended for conformance testing, it can be used in many areas. Examples of areas where TTCN-3 can be used are integration testing, interoperability/inter-working testing, load/stress testing, performance testing, regression testing, robustness testing and system testing. Since TTCN-3 is suitable for testing in a variety of areas, it can be used to test a variety of applications such as automotive applications, broadband technologies, cordless phones, Internet protocols, middleware platforms, mobile communications, smart cards, and wireless LANs.

The advantages of TTCN-3 being a dedicated testing language, is that it is especially designed with testing in mind. TTCN-3 is also abstract (the testing code has to be interpreted and compiled before it can be executed), making it SUT independent and increases the level of reusability. TTCN-3 supports systematic testing and test automation. Another advantage is TTCN-3s standardization. Concepts based on standardization are generally safer investments than non-standardized in-house solutions, since they are more future-proof.

The line-of-action when using TTCN-3 can roughly be divided into three steps: specification, compilation and implementation. The specification part comprises specifying test data descriptions, test cases, test verdicts and test configuration, among others. The second part, compilation, is used to compile the abstract TTCN-3 code into an executable code. Examples of executable codes are C/C++ and Java. Finally, the implementation part is when the compiled code is implemented to an existing system. Most often test suites are executed in test devices and PCs.

The general structure of a TTCN-3 test system defined by ETSI, shown in Figure 48, includes three main parts: the Test Management and Control (TMC), the TTCN-3 Executable (TE) and the adaptation against SUT. The TMC consists of the Test Management (TM), the Component Handling (CH) and the Coding and Decoding (CD). The adaptation against the SUT is performed with the SUT Adapter (SA) and the Platform Adapter (PA). Two interfaces are defined: the TTCN-3 Control Interface (TCI) and the TTCN-3 Runtime Interface (TRI).

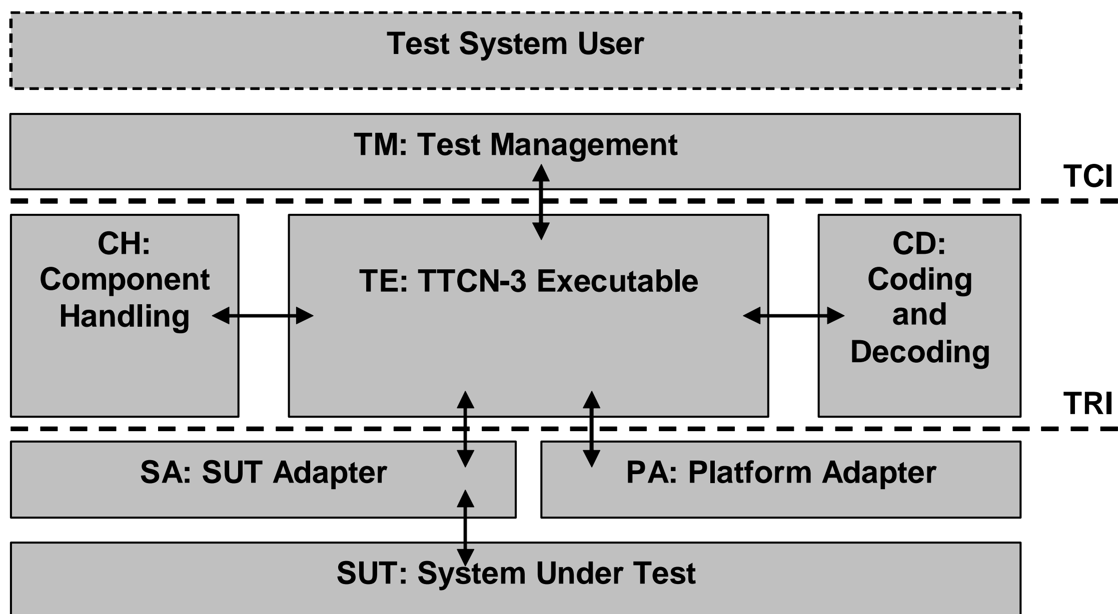


Figure 48: The General Structure of a TTCN-3 Test System

The TM entity is responsible for the overall management of the test system. It is in the TM that the test execution starts after the test system has been initialized. The TM is also the entity that performs the test event logging and presentation to the test system user.

The CD entity is responsible for the encoding and decoding of the TTCN-3 values. The TTCN-3 values are instances of TTCN-3 data types used in the test scripts. To be able to pass TTCN-3 values between the TE and the SUT, encode and decode functions have to be provided. An encode function takes values and encodes them into a transferable binary representation so that they can be sent in an appropriate way. The decoder does the opposite

of the encoder, it takes binary representations and decodes them back to TTCN-3 values. Examples of standardized encoding schemes are Basic Encoding Rules (BER), Packed Encoding Rules (PER) and Abstract Syntax Notation One (ASN.1). ASN.1 is a formal language for abstractly describing messages to be exchanged among an extensive range of applications, while BER and PER are encoding techniques. The TE determines what codecs that should be used, so that the values/bit-strings get properly encoded/decoded.

During a test execution, the execution can be distributed among several test devices or test system components. In order to properly perform a distributed test execution, the CH is needed. The CH implements communication between distributed test systems components. In short, the CH entity provides the needed functionality to synchronize test system components that might be distributed onto several nodes.

The task of the TE part is to execute or interpret TTCN-3 modules, so that they can be executed. The TTCN-3 modules contain the test specification, and before they can be executed, the TE has to identify a number of structural elements that represents different functionality. The structural elements that the TE identifies from the TTCN-3 modules are Control, Behavior, Components, Types, Values and Queues. Often in relation to TTCN-3 the terms Abstract Test Suite (ATS) and Executable test Suite (ETS) are mentioned. Roughly, the ATS can be compared to the TTCN-3 module and the ETS to the TTCN-3 module after compilation.

The SA is used to adapt message and procedure based communication of the TTCN-3 test system with the SUT to the particular execution platform of the test system. The SA is also responsible for sending requests and operations from the TE to the SUT and notifies the TE of received test events from the SUT.

The PA is used to implement TTCN-3 external functions and timers. The PA is responsible for notifying the TE of expired timers as well as letting the TE control the external functions and timers.

The TCI is the interface between the TMC and the TE, and defines the interaction between the TE and the TM, the TE and the CD, the TE and the CH. The TCI provides means for the

TE to manage test execution, distribute execution of test components and encoding and decoding test data.

The TRI is the interface between the TE and the SUT, and defines the interaction between the TE and the SA, the TE and the PA. The TRI provides means for the TE to send test data to the SUT or to manipulate timers, and similarly to notify the TE of received data and timeouts.

D Market Analysis – Product Descriptions

This appendix contains descriptions of products studied during the market analysis in the project.

D.1 Danet TTCN-3 Toolbox

D.1.1 Introduction

Danet's TTCN-3 Toolbox [57], see Figure 49, is a TTCN-3 [53] compiling and processing system that allows users to use available test suites or to develop their own test suites. TTCN-3 Toolbox performs code generation to create executable test suites and provides flexible management of test campaigns using open interfaces. TTCN-3 Toolbox is designed for testing multiple protocols on development platforms and can be used to automate end-to-end/network integration testing.

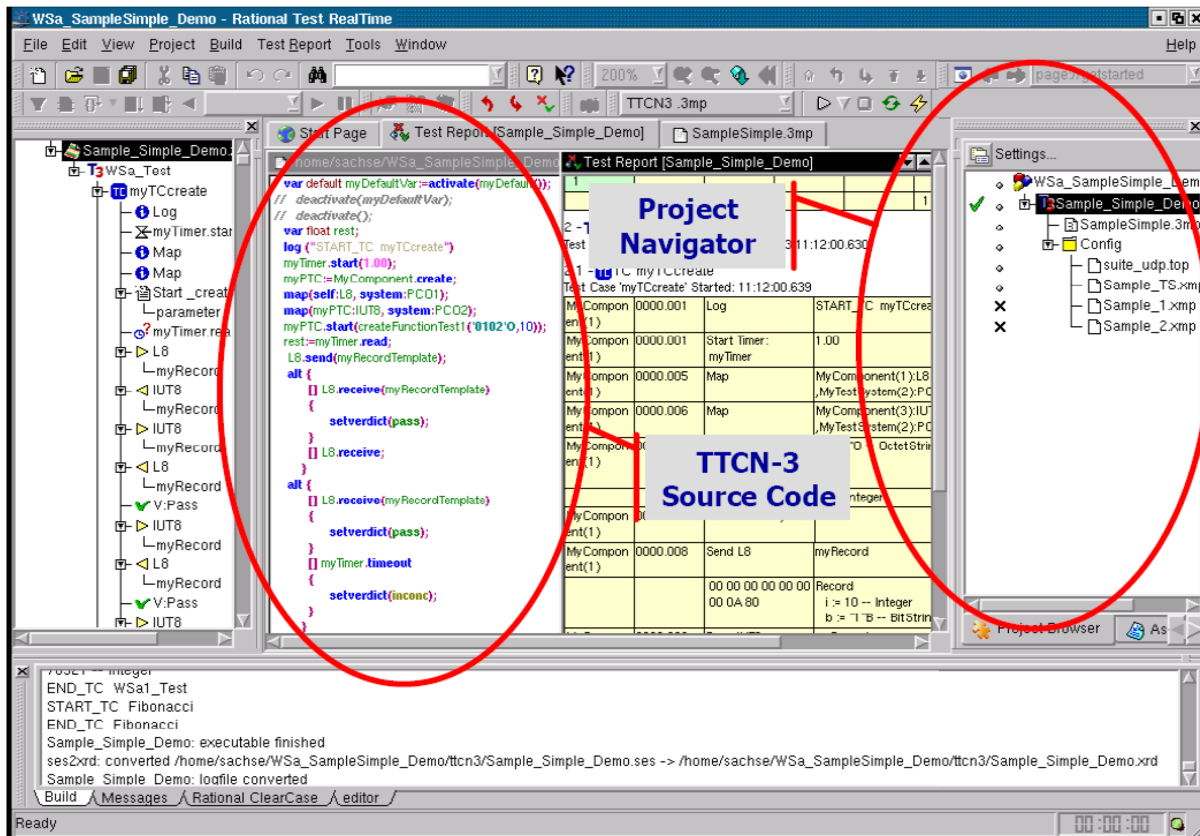


Figure 49: Danet TTCN-3 Toolbox

TTCN-3 Toolbox is based on TTCN-3, see Appendix C. TTCN-3 Toolbox is not only suitable for data communication testing (synchronous) but also telecommunication (asynchronous) and web service testing (XML based).

D.1.2 Functionality

Key features of TTCN-3 Toolbox, according to Danet [57], are to provide support across major industry platforms including multi-platform support, and to provide efficient code generation and execution with possibilities to customization. Other key features of TTCN-3 Toolbox are tracing functionality, result analysis and support for debugging. Since TTCN-3 Toolbox is built on the TTCN-3 TRI [49], adaptability against the SUT is also provided.

Danet TTCN-3 Toolbox has a GUI Framework and an integrated TTCN-3 editor. The TTCN-3 Editor has syntax highlighting for easier usage. TTCN-3 Toolbox is embedded in Rational's Test RealTime Studio GUI framework [45] which eases integration with Rational's modeling tools and version control system.

TTCN-3 Toolbox also features Syntax Analysis. The Syntax Analysis functionality let users easily find errors in their source code. Users can use their external functions and user-defined libraries and link their own code with TTCN-3 test cases. TTCN-2 source code can be converted to TTCN-3 with Danet's TTCN-2 to TTCN-3 source code converter.

In terms of test campaign management, TTCN-3 Toolbox features interactive (user) or test suite (automatically) controlled test campaign management. The TTCN-3 Toolbox GUI admits more than one execution in sequence. The parameters are passed in an XML-based form. TTCN-3 Toolbox has an integrated command line which enables the integration of TTCN-3 Toolbox execution environment into other test management systems.

The runtime interface in TTCN-3 Toolbox is based on the TTCN-3 standardized runtime interface TRI. The TRI contains several sample implementations, which will help users to quickly get started. When using TRI and making the SUT adaptation, TTCN-3 ports have to be mapped to TRI ports. Danet has a 'testconfig' control feature, which helps users map their TTCN-3 ports to the TRI ports without having to re-compile, thus helping users save time.

The encoding and decoding part of messages is done automatically, based on a test suite's type and template information. TTCN-3 Toolbox supports ASN.1 encoding rules, BER, PER and direct encoding. Users can implement their own or third-party codecs via the Test Control Interface – Codec (TCI-CD) [50].

After the execution process, test case trace analysis and reporting can be performed. TTCN-3 Toolbox uses an XML-based trace logging. To show the message structure in a readable form and reduce the need for implementing decoding functionality, automatic mnemonic decoding of events based on the test suite's type and template information is performed. TTCN-3 Toolbox shows online trace display and TTCN-3 log statement events

during a test case/test suite execution. The trace viewing is based on UML sequence diagrams (similar to Message Sequence Charts (MSC) [58].

Debugging is supported by TTCN-3 Toolbox via the built in debugger. The debugger makes it easy to find errors and let the users see how the TTCN-3 code is executed step by step.

TTCN-3 Toolbox supports the Windows 32, Linux and Solaris platforms.

D.1.3 Architecture

Danet has used four phases when describing the architecture of TTCN-3 Toolbox (see Figure 50). The four phases are: Test Generation, Test Execution, Result Analysis and Test Reporting. The first phase Test Generation creates TTCN-3, opens TTCN-3 or imports TTCN-2 source code. The TTCN-3 code is then compiled along with test parameters and environment libraries. User Libraries can also be defined. During the Test Execution phase the Test Campaign Manager executes the test with the compiled TTCN-3 code (executable code). The third phase Result Analysis enables the user to do result comparison, tracing (both textual and graphical) and debugging. Finally, the fourth step Test Reporting is used for post processing and not really a part of Danet TTCN-3 Toolbox.

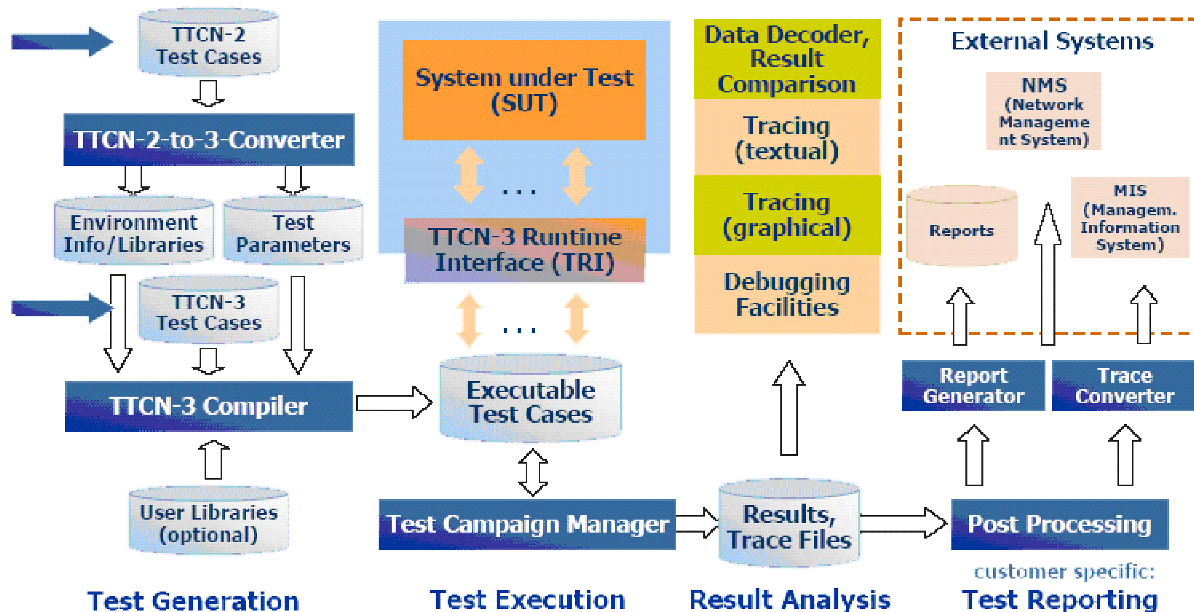


Figure 50: An Architectural Overview over TTCN-3 Toolbox

Before entering the first phase Test Generation, test development has to be done. When performing test development, the test environment is created and test suites and test cases are developed. Before compiling the abstract TTCN-3 test suites and the generation of the executable test code, TTCN-3 Toolbox offers a restrictive syntax and semantic checking. The syntax and semantic check is performed according to ISO 9646 [48]. Users are given the opportunity to convert existing TTCN-2 test cases to TTCN-3 in order to reuse them in the Test Execution phase. When converting TTCN-2 test cases to TTCN-3, it is important to remember that the TTCN-3 source code created when performing the conversion is still TTCN-2 test cases in TTCN-3 format. To take full advantages of TTCN-3, test cases should be created from scratch or at least be modified to fully benefit the TTCN-3 format.

When it is time to compile the TTCN-3 source code to executable C source code, TTCN-3 Toolbox has support for incremental and selective compiling. Incremental compiling will automatically only compile the changed parts since the last compilation. Selective compiling

will let the user choose which parts to compile. Make files are supported and so are user-defined libraries. During the code generation/compilation, Platform for Internet Content Selection/Protocol Implementation Extra Information (PICS/PIXIT) [51] template files are generated as well. The PICS/PIXIT template files specify the test suite parameters and can be further edited with Danet's integrated PICS/PIXIT editor. In short, the purpose of the PICS/PIXIT is to enable information to be provided during startup of the execution.

In the Test Execution phase, the Test Campaign Manager lets the user control the test suite execution interactively. Test cases can be selected manually, automatically or can be controlled by test execution lists. The pre-selection of test cases is performed according to ISO 9646. Users can still modify or override the pre-selection. During the execution, a log window shows test case output and can also show the SUT output. The adaptation towards SUT in TTCN-3 Toolbox is an implementation of the reference interface TRI specified by TTCN-3 [49].

The TTCN-3 test suites are mapped to the communication links in SUT with Danet's Point of Control and Observation (PCO)/Port Editor. The term PCO is used with TTCN-2 while it is called a port in TTCN-3.

The Test Execution phase will generate test results. TTCN-3 Toolbox features a result analysis with general (mnemonic) data decoder, trace analysis and debugging support. The results can be compared using the result comparison. The result analysis let users see what is going on in a test campaign. Statistics, tracing and logging preserves test case and SUT activities. Users can also perform step-by-step analysis of test case execution via hyperlinks to the TTCN editors. Within the result analysis a protocol independent mnemonic decoding is used for all TTCN test suites. The mnemonic decoder shows the TTCN message data structure and description in a general form. The test suite debugging feature does not change test case verdicts and handles all SUT/test suite timing problems correctly. The debug tracing can be performed both online and offline. The execution itself can be performed in parallel according to TTCN-3, and can also be performed on both single and distributed platforms.

D.2 IBM Rational Testing Products

D.2.1 Introduction

IBM Rational provides several Software Quality products: IBM Rational Test Manager, IBM Rational Functional Tester, IBM Rational Manual Tester, IBM Rational Performance Tester, IBM Rational Purify, IBM Rational Robot, IBM Rational Test RealTime.

D.2.2 IBM Rational TestManager

D.2.2.1 Introduction

Rational TestManager [44] includes support for many testing-related activities: planning, design, implementation, execution and evaluation. Different test methods are supported: unit testing, functional regression testing, performance testing and configuration testing [46].

Rational Test Manager can be integrated with several other IBM Rational products: Rational RequisitePro for requirements management, Rational ClearQuest for change management and Rational ClearCase LT for configuration management. Rational Robot is needed in order to develop automated test scripts.

There are also possibilities for customizing and extending the environment by defining new test inputs and new test types [47].

D.2.2.2 Functionality

The following are examples of functionality included in Rational Test Manager [44]:

- Traceability between requirements and test cases.
- Create data pools for supplying data values to the variables in a test script.
- Create test plans with test cases.
- Implement test cases in test scripts.
- Monitor test execution.
- Test report generation.

- Configuration testing support through integration with VMWare server virtualization software. Parallel configuration testing possible by use of agent computers.
- Submit a defect to ClearQuest, with build, configuration, and script information automatically filled in, from a failed log event, in an execution history (test log).

The test script types supported by TestManager [47] are listed in Table 3.

Test Script Type	Description
GUI	A functional test script written in SQABasic, a proprietary Basic-like scripting language.
VU	A performance test script written in VU, a proprietary C-like scripting language.
VB	A test script written in the Visual Basic language.
Java	A test script written in the Java language.
Command Line	A test script (written in any language) that can be executed from the command line – for example, a DOS batch file, a Perl or Bourne shell script, or compiled C program.
Manual	A procedure explaining how to perform a test manually that, when executed, prompts a tester to verify the result of the test.

Table 3: IBM Rational TestManager Test Script Types

IBM Rational TestManager can be extended in two ways: by adding custom test script types or by adding custom test input types [47].

Rational TestManager has an built-in data store for recording test assets and artifacts such as test suites, test plans, test cases, reports, test logs, scripts, users, groups, computers. The test assets and artifacts are stored in XML-format in Rational Test Asset Parcel (.RTPAR) files and can be exported from TestManager and imported to TestManager in this format.

Test data to be used in test scripts are managed by means of data pools. There are functions for creating and editing data pools within TestManager. Data pools are stored in two files: a .csv text file with data pool values and a .spc specification file with data pool column names. Data pools with data types and data values can also be imported into TestManager.

In TestManager, a test plan is created to manage folders with test cases. A test case is implemented by building a test script and then associating that test script with the test case.

D.3 JUnit

JUnit [52] is a unit testing framework for Java. The interfaces and classes in the JUnit API are shown in the class diagram in Figure 51. A test is created in a specialized test class, by subclassing TestCase. Typically, the test class contains separate methods for testing the corresponding methods in the class being tested, the Implementation Under Test. There are convenience functions in the framework for setting up the test data before each test method is run and tearing down the test data afterwards. There are also different assert methods for testing various expected states.

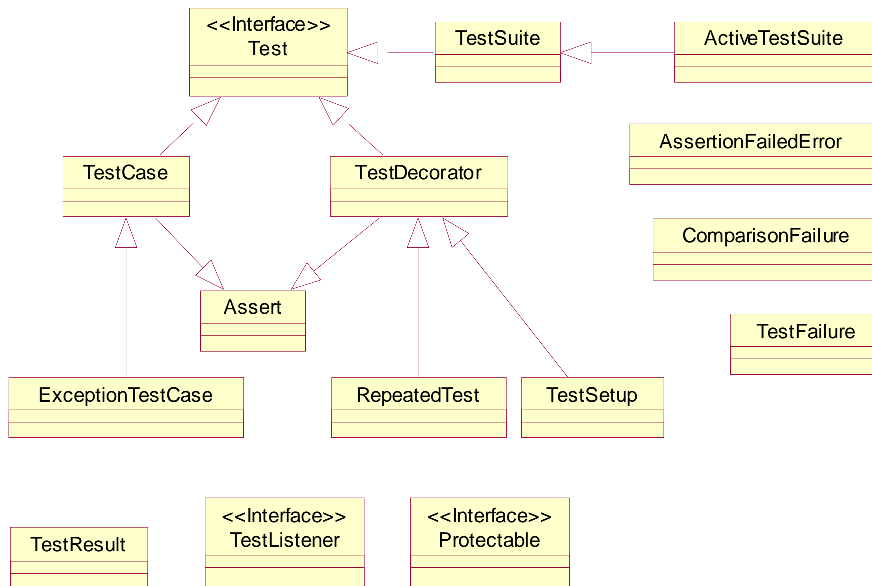


Figure 51: JUnit Interfaces and Classes

D.4 OpenTTCN Tester for TTCN-3

D.4.1 Introduction

OpenTTCN Oy [65] is a Finish company that specializes in test execution tools. OpenTTCN Oy has been active since 1993 and their primary focus is development of TTCN based testing tools and components.

OpenTTCN Tester for TTCN-3 [66] is a test automation tool for executing both in-house developed and standard test suites written in TTCN-3 [53] and TTCN-2 [48]. OpenTTCN Tester for TTCN-3 is compliant with the ISO/IEC 9646 [48] conformance testing methodology and framework.

OpenTTCN Tester for TTCN-3 is available for Windows and Linux, both versions support CLI interaction and the Windows version also features a GUI, OpenTTCN Campaign Manager (see Figure 52). Finally, OpenTTCN Tester for TTCN-3 can be integrated with

third-party test management software using the standardized TTCN-3 Control Interface – Test Management (TCI-TM) [50] interface.

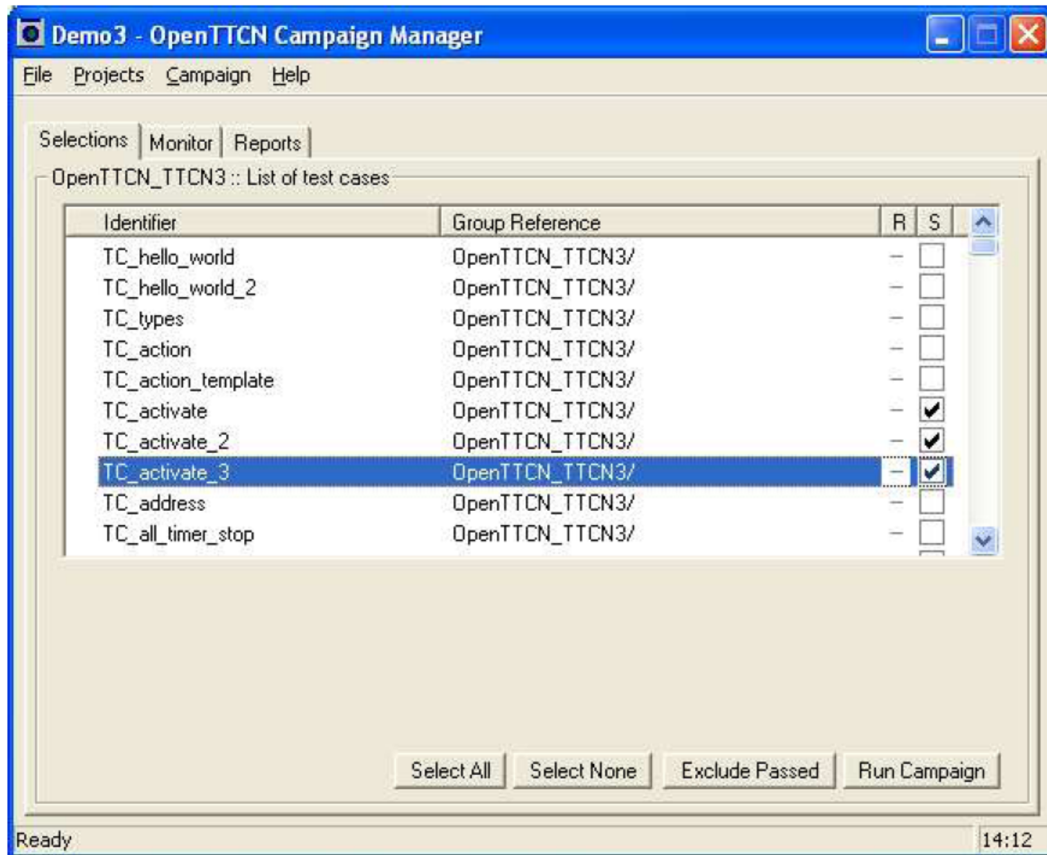


Figure 52: The OpenTTCN Campaign Manager

D.4.2 Functionality

The OpenTTCN virtual machine handles TTCN-3, TTCN-2, and ASN.1 [68] languages as a hybrid just-in-time compiler and interpreter. OpenTTCN Tester for TTCN-3 includes one tool to process TTCN-3 Core Language and ASN.1 language files, while another included tool process files in TTCN-2 Machine Processable format. Parameters for both TTCN-3 and TTCN-2 can be loaded and specified via XML files. A more natural approach when using TTCN-3 is specifying module parameters in a file using the TTCN-3 Core Language syntax

in a modulepar section. The test suite parameters term is used with TTCN-2 and the module parameters term is used for TTCN-3 for the same type of parameters.

OpenTTCN Tester for TTCN-3 complies with the ISO/IEC 9646 OSI Conformance Testing Methodology and Framework, the general framework where TTCN-2 was originally developed. OpenTTCN Tester for TTCN-3 features an interactive display during the test campaign execution and display of TTCN-3 and TTCN-2 log events. The Log events can be saved in text or XML format.

Statistics of previously executed test campaigns is also shown in form of pass, fail, inconclusive, none and error verdicts along with the total number of test cases.

D.4.3 Architecture

The OpenTTCN architecture is built upon distributed modular components. The architecture has been used to implement the OpenTTCN Tester for TTCN-3 among other OpenTTCN products. The architecture can be divided into six components:

- OpenTTCN User Interface
- OpenTTCN Virtual Machine
- OpenTTCN Repository
- OpenTTCN Coding and Decoding
- OpenTTCN SUT Adapter
- OpenTTCN Platform Adapter

OpenTTCN Tester for TTCN-3 has two alternative user interfaces: a GUI (available for the Windows platform) and a CLI (available for the Windows platform and the Linux platform). OpenTTCN Tester for TTCN-3 can also be integrated with existing Test Management software using the TCI-TM interface, which is available as a library with an ANSI C API. OpenTTCN also has a web-based test management and control tool user interface, OpenTTCN Xpress [67], which is suitable when sharing testing through Internet or intranets. OpenTTCN Xpress has more sophisticate functionality besides the features of OpenTTCN

Campaign Manager (the Windows GUI). Features are user authentication and authorization for his/her test projects, GUI for editing test suite/module parameters, a database to hold test results, dynamic ISO/IEC 9646 compliant test reports and more detailed test logs.

The OpenTTCN Virtual Machine component, also referred to as the “Server-part”, consists of TTCN-3 and TTCN-2 interpreters and implements part of the TTCN-3 Executable. It implements necessary functionality to control and execute test specifications.

The OpenTTCN Repository component enables storage of TTCN-3, TTCN-2 and ASN.1 test suites and values of test suite and module parameters. The OpenTTCN Repository implements part of the TTCN-3 Executable (together with OpenTTCN Virtual Machine).

The OpenTTCN Coding and Decoding component is used for encoding and decoding TTCN-3 and TTCN-2 messages. OpenTTCN Coding and Decoding is an implementation of the TCI Codec and Decoding (TCI-CD) [50] interface. The same TTCN-3 TCI-CD interface is used for implementing encoders and decoders for both TTCN-3 and TTCN-2 test systems.

The OpenTTCN SUT Adapter is an implementation of the TTCN-3 TRI [49] interface. The same TTCN-3 TRI is used for implementing SUT Adapters (SA) for both TTCN-3 and TTCN-2 test systems. The OpenTTCN SUT Adapter component consists of one or more processes that contain the required functionality to be able to connect the SUT to the Tester (the test executable). In TTCN-3 ports are used to perform the connection, while Points of Control and Observation (PCO) are used in TTCN-2. Both TTCN-3 ports and TTCN-2 “ports” are implemented using TTCN-3 TRI interface.

The OpenTTCN Platform Adapter is an implementation of the triPlatform interface. The same triPlatform interface is used for implementing Platform Adapters (PA) for both TTCN-3 and TTCN-2 test systems. Both TTCN-3 external functions and TTCN-2 Test Suite Operations are implemented using TTCN-3 triPlatform interface. The OpenTTCN Platform Adapter can be implemented as a separate process or combined with the OpenTTCN SUT Adapter. The OpenTTCN Platform Adapter component consists of one process that contains external functions defined in the test specification in abstract terms. The external functions are

used to add new operations to the test language that cannot be specified otherwise. In TTCN-2 the term Test suite Operation Declarations (TSO) are used.

OpenTTCN Tester for TTCN-3 supports Microsoft Windows 32 (NT 4.0, 2000, XP) and RedHat Linux (9.0) and SuSE Linux (8.2) platforms.

Standards used in OpenTTCN Tester for TTCN-3 are:

- ASN.1
- ISO 9646
- TTCN-2
- TTCN-3 TCI
- TTCN-3 TRI
- PIXIT [51]
- XML

D.5 Scapa Test and Performance Platform 3.1

Scapa Test and Performance Platform 3.1 [43] is a performance testing tool from Scapa Technologies [42] based on the Eclipse Test and Performance Tools Platform (TPTP) [27].

Scapa Test and Performance Platform 3.1 works by simulating multiple users of a computer system and can be used for performance testing or stress testing. End-user response times are measured together with different performance metrics gathered from the SUT.

There are diagnosis and monitoring functionality to help analysing performance and systems information, such as different resource usage, during runtime. Diagnosis and monitoring of collected performance data may be used for optimizing system response times, in order to achieve better end-user experience. Scapa Test and Performance Platform 3.1 helps in discovering and locating the bottlenecks of a system. By optimizing the bottlenecks found, response time can be optimized in an efficient way. Scapa Test and Performance Platform 3.1 may be used as a testing tool when constructing a system, but the most important usage is perhaps the possibility to use it as a help for fixing problems in existing installations.

Problems in existing installations may be purely performance related, but may also involve more serious defects such as hang-ups or mal-functioning programs.

The target environments for Scapa Test and Performance Platform 3.1 are different server-based solutions such as Java/J2EE, Web Services, Windows Client/Server or Citrix [72] Application/Terminal Server.

D.6 Telelogic TAU/Tester

D.6.1 Introduction

Telelogic's TAU/Tester [55], see Figure 53, is a member of the TAU tool family provided by Telelogic [54]. The TAU family is a set of tools that provide support for automating design and development tasks. The TAU family is designed to support systems engineering, software development for embedded and advanced systems, quality assurance and testing. The family consists of four products: TAU/Architect for systems architecture and design, TAU/Developer for model-driven software development, TAU/Logiscope for software quality assurance and metrics, and finally TAU/Tester which is specialized for systems and integration testing over multiple industries.

TAU/Tester is a stand-alone tool based on TTCN-3, see Appendix C. TAU/Tester is built to support the full test cycle. Test automation and support of multiple target environments due to its open structure are also key features. Since TAU/Tester is built on the TTCN-3 standard [53], which has a specified interface called TTCN-3 Runtime Interface (TRI) [49], it is flexible to use with everything from small local to large scaled distributed systems. TAU/Tester can be integrated with various configuration management systems.

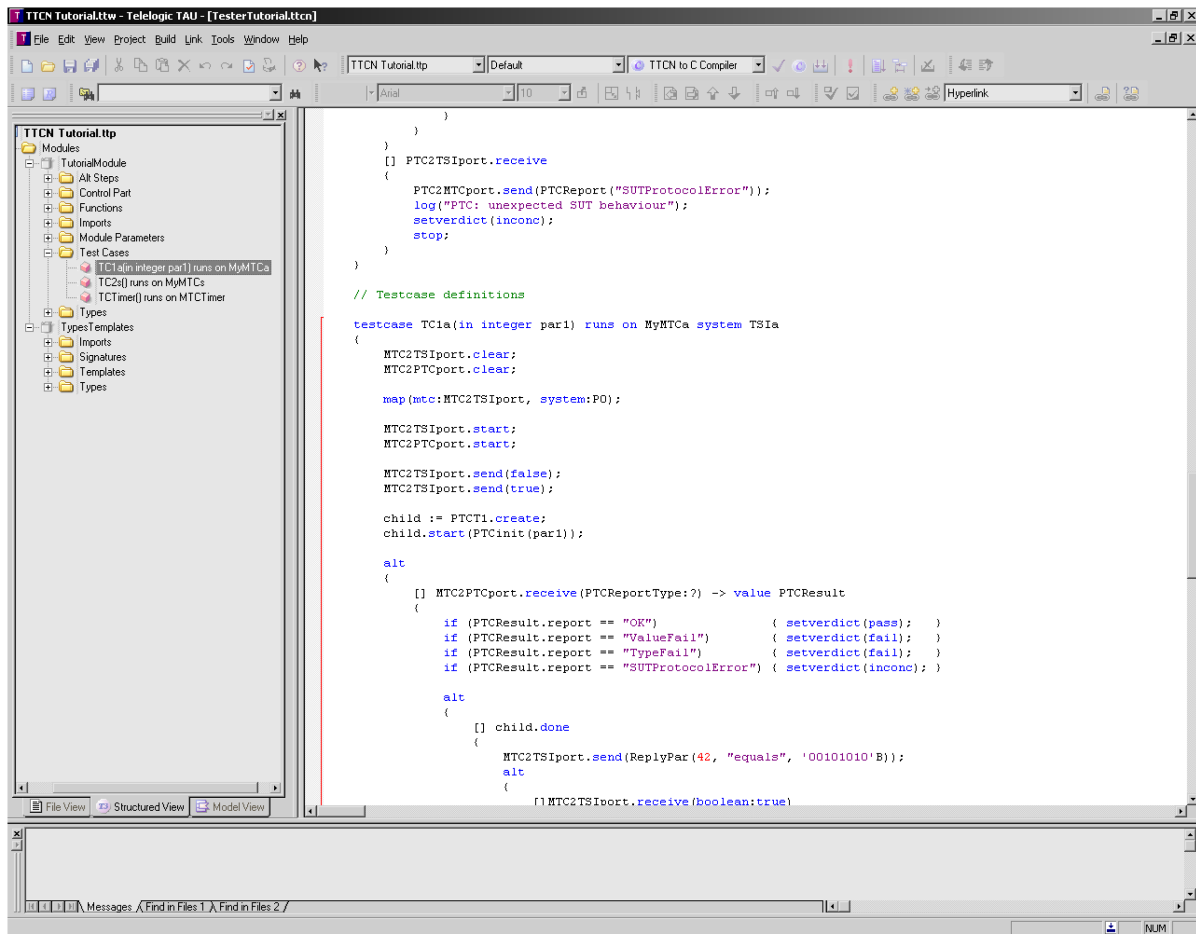


Figure 53: Telelogic TAU/Tester showing a TTCN-3 Tutorial

D.6.2 Functionality

Telelogic TAU/Tester can be used to test a variety of different products and applications within diverse industries. It is suitable for testing of communications such as: switches, infrastructure, datacom devices and Voice over IP (VoIP). It can also be used to perform different tests within military and aerospace, such as command and control systems, military and commercial aircrafts, and satellite systems. Testing can also be done using TAU/Tester

within the transportation sector including electronic control units (ECUs) and chassis systems, vehicle information and computing system and much more.

Testing is not just a phase in software development; it contains its own cycle. The Test cycle includes the following phases, according to Telelogic [54]: test design and development, analysis, execution and debugging. It is important to remember the importance of testing. Testing will help detecting defects and by finding errors early costs are reduced. Testing also improves the quality of the software and customer's requirements are more easily met. Telelogic has kept the full test cycle concept in mind when developing TAU/Tester, which will help the user find the red tread during the test phase.

One of the most important features of TTCN-3 is the ability to dynamically be able to construct and re-configure distributed components. To be able to dynamically construct and re-configure along with being able to execute and run components in parallel, makes TTCN-3 based test tools very efficient to perform testing in even the most complex distributed systems. TAU/Tester takes advantage of these benefits of TTCN-3 and has support for large distributed systems and for integration of configuration and version management tools. Being able to execute parallel tests also makes TAU/Tester feasible for load tests over distributed systems.

D.6.3 Architecture

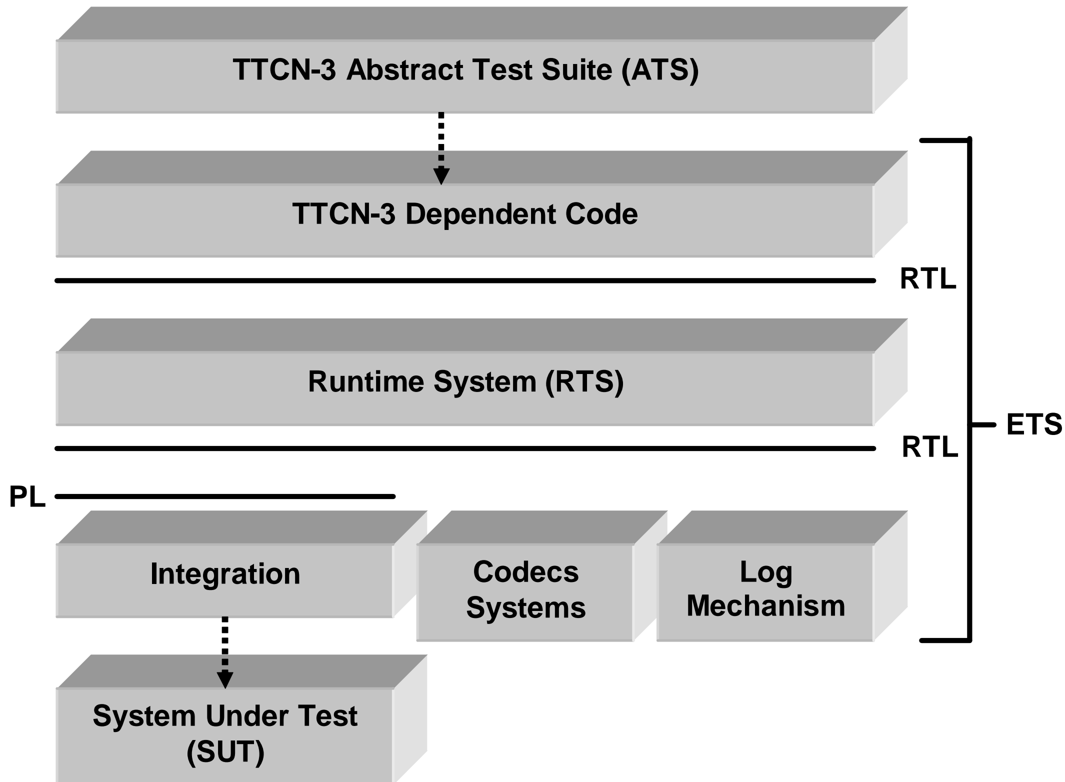


Figure 54: The Architecture of TAU/Tester's Executable Test Suite (ETS)

Telelogic [56] uses the concept Executable Test Suite (ETS) in TAU/Tester to describe the architecture used when executing tests, see Figure 54. The ETS consists of five parts: TTCN-3 Dependent Code, Runtime System (RTS), Integration, Codecs Systems and Log Mechanisms. The ETS also specifies two interfaces: Runtime Layer (RTL) and Platform Layer (PL).

The TTCN-3 Abstract Test Suite (ATS) Generated Code is used to compile the abstract test language (TTCN-3) into an executable language (C language). The generated C code is only used to be able to execute the test.

D.6.3.1 The Runtime System (RTS)

The Runtime System can be seen as the engine of the TTCN-3 test suite execution. It handles values, control components and much more. Two interfaces are defined by the RTS: Runtime Layer (RTL) and the Platform Layer (PL). The RTL interface defines services provided by the RTS. Services that can be accessed via RTL are used by non-TRI integrations and encoders/decoders as an example. The second interface defined by RTS, PL, defines services that RTS needs from the Integration module, to be able to function properly.

When executing tests, a great amount of memory is allocated. Allocation of the memory can be done permanently or temporarily. The RTS uses temporary memory allocation due to performance reasons and to minimize the potential risk of memory leaks. The memory allocation in RTS is performed with a dynamically growing memory area, which expands automatically when needed. The RTS is configurable to be able to change its behavior. To be able to save configurations and let integrations and codecs, among others, get access to the configurations, a storage facility is used. The storage facility is populated with key-value pairs represented as TTCN-3 RTS values. Source Tracking is used in RTS to keep track of source code locations during the execution. The source tracking is also used to track execution in other integrations modules like encoder and decoder functions and log mechanism implementations.

D.6.3.2 The Integration Module

The integration module of the ETS is something that has to be implemented by the user to make the RTS able to communicate with the SUT. What has to be done is an implementation of the PL interface, which is defined by the RTS. The PL interface defines what is needed by the RTS in forms of services to be able to provide integration functionality correctly. Examples of services are memory primitives, representation of timers, handling of time, SUT communications and task concurrency primitives. The implementation of the integration module can be performed in three different ways: by using the provided TRI integration, by extending and modifying the non-TRI example integration (which is provided by

TAU/Tester) or by implementing a non-TRI integration from scratch. According to Telelogic, the first way (using the TRI integration) is the easiest way. The second and third way, extending the non-TRI or building it from scratch, gives more flexibility but also requires more of the user (especially when building the non-TRI from scratch). When using the provided TRI integration, an implementation covering the System Adaptor (SA) and Platform Adaptor (PA), both specified by the TRI, has to be done. When using the one of the non-TRI integrations, the PL implementation has to be done more or less by the user.

D.6.3.3 The Codecs Systems

The RTS supports multiple codecs systems, which all are registered at runtime during the initialization phase. When the initialization phase is performed all codecs systems encoder and decoder functions have to be associated with the existing types in the system that needs to be encoded during execution.

D.6.3.4 The Log Mechanisms

Logging during the execution is an essential part of the RTS. Two mechanisms for logging are provided by TAU/Tester: a default text-based log mechanism and a log mechanism that logs to files with Message Sequence Charts (MSC-96) syntax. The RTS supports an easy way to plug in user-defined log mechanisms, in order to save logs in the way that the user wants. Each component has its own log instances. Only events and information that is related to every specific component will be logged.

D.7 Testing Technologies TTworkbench

D.7.1 Introduction

Testing Technologies [69], a spin-off of Fraunhofer FOKUS [70] research institute, develops test development tool series and solutions based on the standardized test specification language TTCN-3.

Testing Technologies' product TTworkbench, see Figure 52, is a graphical test development and execution environment using TTCN-3. TTworkbench is based on the Eclipse platform [25] and is available in three versions:

- TTworkbench Basic
- TTworkbench Professional
- TTworkbench Enterprise

All versions of TTworkbench have a built-in TTCN-3 compiler and a textual TTCN-3 editor, while the Professional and Enterprise versions of TTworkbench also have a graphical TTCN-3 editor and ASN.1 [68] and Interface Definition Language (IDL) [71] data support. Finally, all TTworkbench versions feature test management, execution and analysis; in addition, the Enterprise version of TTworkbench can perform distributed execution.

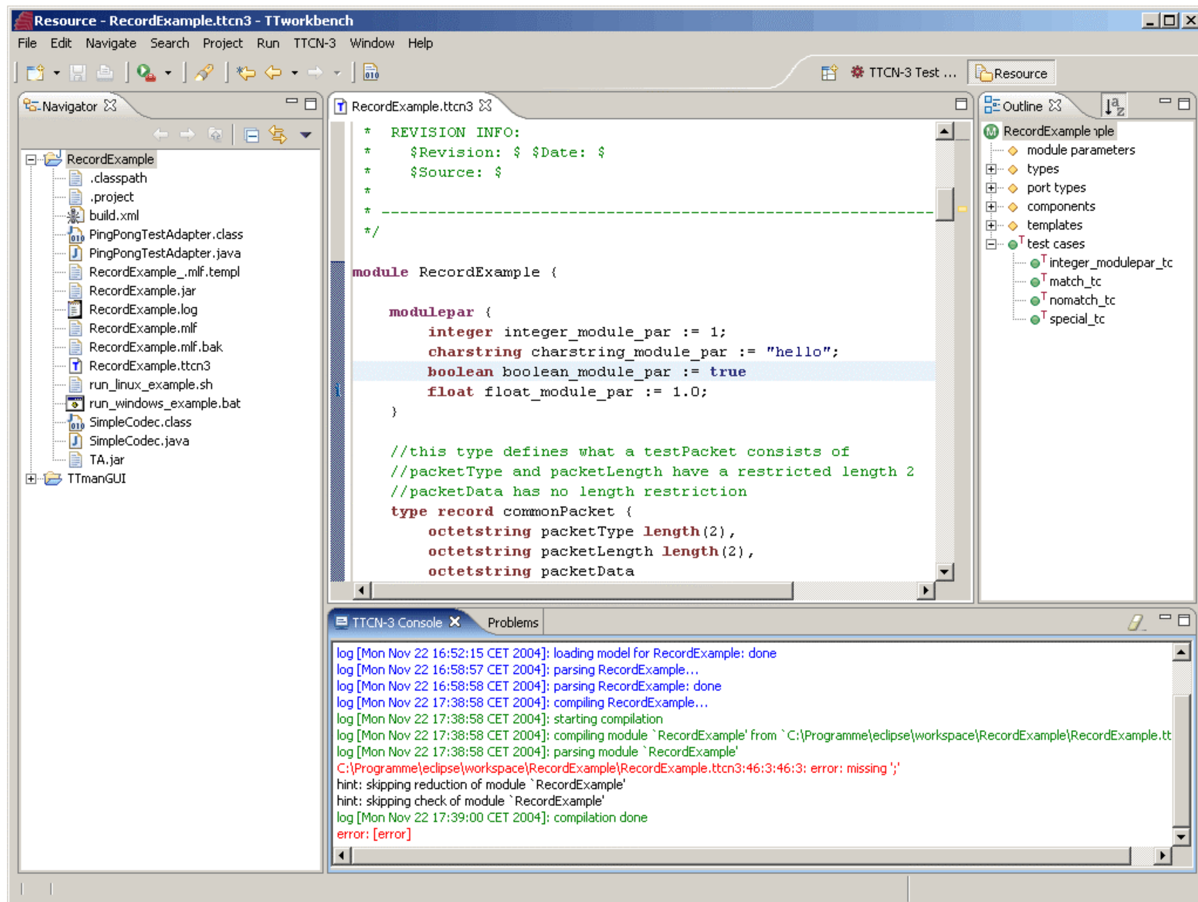


Figure 55: TTworkbench showing the Built-in Text Editor

D.7.2 Functionality

TTworkbench features:

- TTCN-3 Core Language editor
- Graphical TTCN-3 editor
- TTCN-3 Compiler
- Test management
- Execution
- Analysis

The TTCN-3 Core Language editor (CL-editor) fully supports the TTCN-3 ETSI standard. The CL-editor supports standard text editor functions, syntax highlighting, and validation of test specifications. Other features of the CL-editor are: Error reporting with source navigation, online parsing and a TTCN-3 console.

The Graphical TTCN-3 editor (GFT editor) enables graphical design and visualization of test cases. The graphical test cases are represented in GFT sequence diagrams. Native TTCN-3 data can be imported for type and template definitions, messages and data handling. The GFT can be generated out of TTCN-3 core language, and GFT can be generated on-line to TTCN-3 core language. The GFT can be exported to Graphic Interchange Format (GIF) images for documentation purposes. XML is used as storage format for the GFT for possible future interoperability.

The TTCN-3 Compiler enables compilation from the CL-editor and TTCN-3 console. It fully supports the TTCN-3 ETSI standard with dynamic configuration, both message-based and procedure-based communication, modularization with importing and test control. The TTCN-3 Compiler supports error reporting with source navigation.

TTworkbench supports a variety of functions to manage, execute and analyze TTCN-3 compiled test suites. Logging of the TTCN-3 test case execution results can be performed both online and offline, and filtering of the logs is also possible. Test data/results can be saved and opened in order to view statistics, analyze and validate data. Generation of test reports and scripting for batch mode tests can also be performed with TTworkbench.

D.7.3 Architecture

TTworkbench is implemented in Java and based on the Eclipse 3.0.1 platform. Eclipse Modeling Framework (EMF) is also needed as well as Java 1.4.2 to get TTworkbench up and running. The architecture consists of several plug-ins for Eclipse. Some of them are TTworkbench CLEditor plug-in, TTworkbench TTthree plug-in and TTworkbench TTman plug-in. The plug-ins can be seen as functionality divided into smaller parts.

The TTworkbench CLEditor plug-in contains the functionality required for the CL-editor. It has been implemented to fully support the TTCN-3 ETSI standard. The Ttworkbench CLEditor plug-in adds the TTCN-3 CL-editor and a TTCN-3 menu into the Eclipse platform.

The TTworkbench TTthree plug-in holds the functionality necessary to compile TTCN-3 modules into test executables. Adaptation against SUT is supported via the TTCN-3 Runtime Interface (TRI). Integration of external codecs is supported via the TTCN-3 Control Interfaces (TCI).

Finally, the TTworkbench TTman plug-in (or TTmex in TTworkbench Enterprise version, which supports distributed execution) is responsible for test management, execution and analysis functionality. Additionally plug-ins installed are plug-ins for core functionality, help and meta models.

Since TTworkbench is built upon Java, all platforms supported by Eclipse and Java, are supported by TTworkbench.

Standards used in TTworkbench are:

- ASN.1
- IDL
- TTCN-3
- XML

E Market Analysis – Comparison Points

Table 4 contains a summary of the points used to compare different products in the market analysis.

Comparison Point	Description
General	
Main target SUT environment	For example Telecom, Desktop, Distributed Systems, Web-based applications etc.
Test Management	Test Management phases supported: Preparation, Execution, Evaluation.
Open Source, license	If the product is Open Source, and if so, which license is used.
Platform	OS support
Test methods supported	
Automatic testing	Support for automated testing.
Distributed systems	Support for remote test execution, data collection etc.
GUI-testing	Support for testing Graphical User Interfaces.
Load/Stress testing	Type of load testing supported; for example parallel execution with virtual users or interaction with external load generators. (This point also includes support for stress testing; the assumption is that the same functionality is required.)
Manual testing	Support for manual test cases, as a special case of automatic test cases.
Protocol testing	Support for protocol testing.
Unit testing	Support for unit testing.

Interoperability	
Data models	Data models used for test assets, artifacts, etc.
Database support	Built-in database. Support for import, export from external database.
Launching support (remote)	Support for remote launching of test bed, for example: start up of nodes, connection of nodes, data sanity check, test execution and test bed tear down.
Standards used/supported	Which standards (data formats, models etc) that the product uses and/or supports.
SUT interface connection	Technical solution for remote connection to SUT, for example adapters, interfaces etc.
Test asset export	Support for exporting different test assets, for example test cases, execution history, logs, traces.
Test asset import	Support for importing different test assets, for example test cases, execution history, logs, traces.
Tool integration	Support for integrating different external tools

Test Preparation	
Capture/playback	Support for capture/playback, for example GUI events or HTTP packets.
Editor	Built-in editor for test development; test plans, test cases, test scripts, test data. External editors supported.
SUT Instrumentation	Support for instrumenting SUT source code for coverage, performance, or trace data.
Test case re-use	Support for re-using test cases by importing them from an external system (external repository or external tool).
Test data	Support for creating, editing and associating test data with test cases.
Test data re-use	Support for re-using test data by importing it from an external repository.
Test script languages	Test script languages supported.

Test Execution (real-time)	
SUT log monitoring	Runtime monitoring of log events created by the SUT.
SUT performance/load/statistics monitoring	Runtime monitoring of SUT performance, load or other measurements.
SUT trace monitoring	Runtime monitoring of SUT traces (graphical/textual), for example sequence diagram of method calls between classes (normally requires instrumented SUT source code).
Test case execution monitoring	Runtime monitoring of test execution (test cases with pass/fail verdicts).

Evaluation (post mortem)	
Execution history analysis	Analysis of test execution history (search, filter etc).
SUT log analysis	Analysis of SUT logs.
SUT profiling analysis	Analysis of coverage and/or performance profiling data.
SUT trace analysis	Support for viewing SUT traces (graphical/textual) after test execution, for example sequence diagram of method calls between classes.

Table 4: Comparison Points used in the Market Analysis

F Market Analysis – Product Evaluations

The product evaluations are divided into two groups: ready-to-use products see Section F.1, and frameworks for building new test tools, see Section F.2.

F.1 Ready-to-Use Products

F.1.1 Danet TTCN-3 Toolbox

General	
Main target SUT environment	TTCN-3: • Telecom (asynchronous message exchange) • Datacom (synchronous client/server communications) Web Service (XML-based communications)
Test Management	Preparation (development, generation), Execution, Evaluation (analysis, reporting)
Open Source, license	No
Platforms supported	Win32, Solaris, HP-UX, Digital Tru64 Unix, Linux, LynxOS, VxWorks
Test Methods Supported	
Automatic testing	Yes
Distributed systems	Yes
GUI-testing	No
Load/Stress testing	Yes (Parallel execution TTCN-3)
Manual testing	No
Protocol testing	Yes
Unit testing	No
Interoperability	
Data models	Based on the TTCN-3 standard
Database support	No
Launching support (remote)	No
Standards used/supported	TTCN-3, ASN.1, BER, PER, XML-based TTCN-3 trace logging
SUT interface connection	TTCN-3 TRI standard
Test asset export	Export of execution history (TTCN-3 trace) in XML based format
Test asset import	TTCN-2 test cases (TTCN-2-to-3 Converter)
Tool integration	Integration with IBM Rational Test RealTime
Test Preparation	
Capture/playback	No
Editor	Yes
SUT instrumentation	No
Test case re-use	Yes (TTCN-2 and TTCN-3 test cases)
Test data	No
Test data re-use	No
Test script languages	TTCN-2 via import, TTCN-3, ASN.1
Test Execution (real-time)	
SUT log monitoring	No
SUT performance/load/statistics monitoring	No
SUT trace monitoring	No
Test case execution monitoring	Yes
Evaluation (post mortem)	
Execution history analysis	Yes, TTCN-3 trace analysis, hyperlinks to TTCN-3 source code
SUT log analysis	No
SUT profiling analysis	No
SUT trace analysis	No

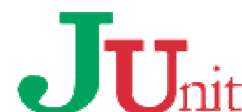
F.1.2 Eclipse TPTP 3.2 (as a ready-to-use product)

General	
Main target SUT environment	Java, Web-app
Test Management	Preparation, execution, evaluation
Open Source, license	Eclipse Public License (EPL)
Platforms supported	Win32, Linux, Solaris, AIX, HPUX, Mac, IBM iSeries, IBM zSeries
Test Methods Supported	
Automatic testing	Yes
Distributed systems	Yes
GUI-testing	Test tool for browser-based applications
Load/Stress testing	No
Manual testing	Yes, Test tool for manual test cases
Protocol testing	No
Unit testing	Yes, Test tool for JUnit tests
Interoperability	
Data models	Test definition, Test execution history, Log, Trace, Statistical
Database support	No
Launching support (remote)	Java JVM
Standards used/supported	UML (2.0), UML2 Test Profile (U2TP), Common Base Event (CBE), XML, XMI, JMTI
SUT interface connection	Testability interface (Remote Agent Controller and remote agents)
Test asset export	Yes, but limited. Resource files in zip file format saved in user's workspace.
Test asset import	No
Tool integration	Integration with Web browsers for recording HTTP requests. Integration with Performance Monitor on Windows and Linux. Integration with JVMPI for Java profiling (code coverage, execution times, memory analysis).
Test Preparation	
Capture/playback	Web-app
Editor	JUnit, Web-app, Manual test cases
SUT instrumentation	Java profiling, see Tool integration
Test case re-use	No
Test data	Yes (data pools)
Test data re-use	No
Test script languages	Java
Test Execution (real-time)	
SUT log monitoring	No. Requires customized adapter.
SUT performance/load/statistics monitoring	Yes, Performance monitor
SUT trace monitoring	Yes, Java profiling
Test case execution monitoring	Yes, Test Execution History
Evaluation (post mortem)	
Execution history analysis	Yes
SUT log analysis	No. Requires customized adapter.
SUT profiling analysis	Yes, Java profiling: code coverage on method level, execution times on method level, memory usage analysis.
SUT trace analysis	Yes, Java profiling

F.1.3 IBM Rational Test Manager

General	
Main target SUT environment	Desktop
Test Management	Preparation (planning, design, implementation), execution, evaluation
Open Source, license	No
Platforms supported	Win32, Linux
Test Methods Supported	
Automatic testing	Yes
Distributed systems	Support for local and remote test execution
GUI-testing	Yes (Rational Robot)
Load/Stress testing	Yes (virtual users, data pools)
Manual testing	Yes (manual test scripts)
Protocol testing	No
Unit testing	Yes
Interoperability	
Data models	See database support.
Database support	Built-in datastore for test suites, test plans, test cases, reports, test logs, scripts, users, groups, computers
Launching support (remote)	No
Standards used/supported	XML, in .RTPAR asset files
SUT interface connection	No
Test asset export	Yes, via XML .RTPAR file format
Test asset import	Yes, via XML .RTPAR file format, Data pools (.csv files)
Tool integration	External scripts via command line interface. Integration with several other IBM Rational products. Test input adapters, execution adapters, test asset import/export (file level interoperability).
Test Development	
Capture/playback	Robot - record test scripts (QualityArchitect - generate test scripts from Rose models)
Editor	Editor for test Plans, test Cases, test scripts, datapools
SUT instrumentation	No
Test case re-use	Custom test scripts can be used
Test data	Datapools (.cvs + .spc files)
Test data re-use	Import via .csv files
Test script languages	Rational SQABasic (GUI test script), VU or VB (generated from Robot), Java, Custom test script type
Test Execution (real-time)	
SUT log monitoring	No
SUT performance/load/statistics monitoring	Local/Agent computer resource usage - for configuration testing
SUT trace monitoring	No
Test case execution monitoring	Yes (Progress Bar, Suite Views)
Evaluation (post mortem)	
Execution history analysis	Test log analysis; filter, defect creation, comparators (for verification points created in Robot)
SUT log analysis	No
SUT profiling analysis	Profiling when unit testing supported in Rational Test RealTime
SUT trace analysis	No

F.1.4 JUnit



General	
Main target SUT environment	Java
Test Management	Implementation, Execution
Open Source, license	Common Public License Version 1.0
Platforms supported	Java (Win32, Linux, Solaris)
Test Methods Supported	
Automatic testing	Yes
Distributed systems	No
GUI-testing	No
Load/Stress testing	No
Manual testing	No
Protocol testing	No
Unit testing	Yes
Interoperability	
Data models	No
Database support	No
Launching support (remote)	No
Standards used/supported	No
SUT interface connection	No
Test asset export	No
Test asset import	No
Tool integration	No
Test Preparation	
Capture/playback	No
Editor	No
SUT instrumentation	No
Test case re-use	No
Test data	No
Test data re-use	No
Test script languages	Java
Test Execution (real-time)	
SUT log monitoring	No
SUT performance/load/statistics monitoring	No
SUT trace monitoring	No
Test case execution monitoring	Graphical TestRunner
Evaluation (post mortem)	
Execution history analysis	No
SUT log analysis	No
SUT profiling analysis	No
SUT trace analysis	No

F.1.5 OpenTTCN Tester

OpenTTCN

General	
Main target SUT environment	TTCN-3: • Telecom (asynchronous message exchange) • Datacom (synchronous client/server communications)
Test Management	Execution
Open Source, license	No
Platforms supported	Win32, Linux
Test Methods Supported	
Automatic testing	Yes
Distributed systems	Yes
GUI-testing	No
Load/Stress testing	Yes (Parallel execution TTCN-3)
Manual testing	No
Protocol testing	Yes
Unit testing	No
Interoperability	
Data models	Based on the TTCN-3 standard
Database support	No
Launching support (remote)	No
Standards used/supported	ASN.1, ISO 9646, TTCN-2, TTCN-3 TCI, TTCN-3 TRI, PIXIT, XML
SUT interface connection	TTCN-3 TRI standard
Test asset export	TTCN logs as text/XML
Test asset import	No
Tool integration	Command-line user interface to OpenTTCN Tester for scripting and integration with other tools. TCI-TM ANSI C API for integration with other tools.
Test Preparation	
Capture/playback	No
Editor	No
SUT instrumentation	No
Test case re-use	Yes
Test data	No
Test data re-use	No
Test script languages	TTCN-2, TTCN-3, ASN.1
Test Execution (real-time)	
SUT log monitoring	No
SUT performance/load/statistics monitoring	No
SUT trace monitoring	No
Test case execution monitoring	Yes
Evaluation (post mortem)	
Execution history analysis	Yes
SUT log analysis	No
SUT profiling analysis	No
SUT trace analysis	No

F.1.6 Scapa Test and Performance Platform 3.1

General	
Main target SUT environment	Java/J2EE Web-apps Windows client/server Citrix Terminal Services Other Thin Client Environments
Test Management	Preparation, Execution, Evaluation
Open Source, license	No
Platforms supported	Eclipse supported
Test Methods Supported	
Automatic testing	Yes
Distributed systems	Yes
GUI-testing	Web-app
Load/Stress testing	Yes, simulation of multiple users
Manual testing	No
Protocol testing	No
Unit testing	No
Interoperability	
Data models	Eclipse based
Database support	No
Launching support (remote)	No
Standards used/supported	See Test asset export. Eclipse based.
SUT interface connection	Eclipse based
Test asset export	Exports results to Excel. HTML-based test report.
Test asset import	No
Tool integration	Integration with CVS. Interfaces to third-party repositories. See Main target SUT environment.
Test Preparation	
Capture/playback	Web-app (HTTP)
Editor	Yes
SUT instrumentation	No
Test case re-use	Possible to import test cases from other tests.
Test data	Virtual users
Test data re-use	No
Test script languages	Wintask, C++, VB, different object based scripts
Test Execution (real-time)	
SUT log monitoring	Yes, event logs
SUT performance/load/statistics monitoring	Yes, application service levels, system utilization. Performance seen by virtual users. Systems information and performance statistics.
SUT trace monitoring	No
Test case execution monitoring	Control of multiple tests and test variables in real-time.
Evaluation (post mortem)	
Execution history analysis	Correlation of test results and systems performance data.
SUT log analysis	Yes, event logs
SUT profiling analysis	No
SUT trace analysis	No

F.1.7 Telelogic TAU/Tester

General	
Test Tool Framework	No (see definition in Section 4.2)
Main target SUT environment	TTCN-3: Telecom (asynchronous message exchange) Datacom (synchronous client/server communications) Web Service (XML-based communications)
Test Management	Development, generation, execution
Open Source, license	No
Platforms supported	Win32, Solaris 8, Linux RedHat Enterprise 3.0
Test Methods Supported	
Automatic testing	Yes
Distributed systems	Yes
GUI-testing	No
Load/Stress testing	Yes (Parallel execution TTCN-3)
Manual testing	No
Protocol testing	Yes
Unit testing	No
Interoperability	
Data models	Based on the TTCN-3 standard
Database support	No
Launching support (remote)	No
Standards used/supported	TTCN-3 (abstract test language, TRI), ASN.1, BER, PER, MSC (Logging)
SUT interface connection	TTCN-3 TRI standard, non-TRI example implementation
Test asset export	SUT logging (text-based or MSC-96 or user-defined)
Test asset import	No
Tool integration	Configuration management via SCCI 1.1
Test Preparation	
Capture/playback	No
Editor	Own Simple
SUT instrumentation	No
Test case re-use	Yes
Test data	No
Test data re-use	No
Test script languages	TTCN-3
Test Execution (real-time)	
SUT log monitoring	No
SUT performance/load/statistics monitoring	No
SUT trace monitoring	No
Test case execution monitoring	No
Evaluation (post mortem)	
Execution history analysis	Yes
SUT log analysis	No
SUT profiling analysis	No
SUT trace analysis	No

F.1.8 Testing Tech TTWorkbench

<testing_tech>

General	
Test Tool Framework	Eclipse plug-in
Main target SUT environment	TTCN-3: Telecom (asynchronous message exchange) Datacom (synchronous client/server communications) Web Service (XML-based communications)
Test Management	Development, generation, execution, evaluation
Open Source, license	No
Platforms supported	See Eclipse
Test Methods Supported	
Automatic testing	Yes
Distributed systems	Yes
GUI-testing	No
Load/Stress testing	Yes (Parallel execution TTCN-3)
Manual testing	No
Protocol testing	Yes
Unit testing	No
Interoperability	
Data models	Based on the TTCN-3 standard
Database support	No
Launching support (remote)	No
Standards used/supported	TTCN-3, ASN.1, IDL, XML
SUT interface connection	TTCN-3 TRI standard
Test asset export	XML export of TTCN-3 sequence diagrams from Graphical Editor
Test asset import	No (TTCN-3 core (text) import in Graphical Editor)
Tool integration	See Eclipse
Test Preparation	
Capture/playback	No
Editor	Both textual (Eclipse plug-in) and graphical (stand-alone)
SUT instrumentation	No
Test case re-use	No
Test data	No
Test data re-use	No
Test script languages	TTCN-3, ASN.1
Test Execution (real-time)	
SUT log monitoring	No
SUT performance/load/statistics monitoring	No
SUT trace monitoring	No
Test case execution monitoring	Yes (On-line TTCN-3 logging)
Evaluation (post mortem)	
Execution history analysis	Yes, TTCN-3 trace analysis, hyperlinks to TTCN-3 source code
SUT log analysis	No
SUT profiling analysis	No
SUT trace analysis	No

F.2 Frameworks

F.2.1 Eclipse TPTP 3.2 (as a framework)

General	
Main target SUT environment	Java, Web-app
Test Management	Preparation, execution, evaluation
Open Source, license	Eclipse Public License (EPL)
Platforms supported	Win32, Linux, Solaris, AIX, HPUX, Mac, IBM iSeries, IBM zSeries
Test Methods Supported	
Automatic testing	Yes, infrastructure for automatic test deployment and execution
Distributed systems	Yes, Remote Agent Controller launches control and data collection agents on remote machines. TCP/IP (sockets) communication.
GUI-testing	Capture of HTTP requests supported.
Load/Stress testing	Possible to use TPTP as a base for a load/stress testing tool, for example simulation of users as with Scapa T&P Platform, see Section F.1.6.
Manual testing	Possible to use TPTP for customized manual testing tool, the ready-to-use manual testing tool may be used as a base.
Protocol testing	Possible to use TPTP as a base for a protocol testing tool, Testing Technologies TTworkbench is an example, see Section D.7.
Unit testing	Possible to use TPTP as a base for a customized unit testing tool, the ready-to-use JUnit based tool may be used as a base.
Interoperability	
Data models	Test definition, Test execution history, Log, Trace, Statistical
Database support	Not in TPTP 3.2, but Relational DB support in future releases of TPTP.
Launching support (remote)	Infrastructure for remote deployment, execution and data collection.
Standards used/supported	UML (2.0), UML2 Test Profile (U2TP), Common Base Event (CBE), XML, XMI, JMTI
SUT interface connection	Testability interface (Remote Agent Controller and remote agents)
Test asset export	Yes, but limited. Resource files in zip file format saved in user's workspace. Relational Data Base support in future releases of TPTP.
Test asset import	No explicit support functions.
Tool integration	Plug-ins in client Workbench. Plug-ins in Remote Agent Controller. Infrastructure for extensions and extension points.
Test Preparation	
Capture/playback	Capture/playback of HTTP requests supported.
Editor	Support for building customized editors, both textual and form based.
SUT instrumentation	Java profiling
Test case re-use	No explicit support functions.
Test data	Yes (data pools)
Test data re-use	No explicit support functions.
Test script languages	Possible to extend Eclipse for any script language.
Test Execution (real-time)	
SUT log monitoring	Support for creating log adapters to Common Base Events. Rich set of ready-to-use functions, may be extended/customized.
SUT performance/load/statistics monitoring	Support for performance monitoring. Support for collection and monitoring of statistical data.
SUT trace monitoring	Java profiling supported, including trace data.
Test case execution monitoring	Standardized test execution messages that can be extended via customized messages. Support for customized execution history viewer.
Evaluation (post mortem)	
Execution history analysis	Support for customized execution history viewer.
SUT log analysis	Rich set of ready-to-use functions, may be extended/customized.
SUT profiling analysis	Yes, Java profiling: code coverage on method level, execution times on method level, memory usage analysis.
SUT trace analysis	Java profiling supported, including trace data.

F.2.2 STAF

General	
Main target SUT environment	Distributed systems
Test Management	Execution
Open Source, license	LGPL
Platforms supported	Win32, Linux, AS/400, AIX, Solaris, HP-UX, Irix, z/OS
Test Methods Supported	
Automatic testing	Yes
Distributed systems	Yes
GUI-testing	No explicit support functions
Load/Stress testing	Possible to use STAF as a base for a load/stress testing tool.
Manual testing	Possible to use STAF as a base for a manual testing tool.
Protocol testing	Possible to use TPTP as a base for a protocol testing tool.
Unit testing	Not very suitable for creating a unit testing tool.
Interoperability	
Data models	No special data models supported.
Database support	No explicit support functions
Launching support (remote)	Peer-to-peer network with STAFProc daemon processes on each machine. Process service for starting, stopping and query processes.
Standards used/supported	XML
SUT interface connection	See Tool Integration
Test asset export	No explicit support functions
Test asset import	No explicit support functions
Tool integration	Pluggable services. Support for interaction from C/C++, Java, Rexx, Perl, Tcl and from the command line/shell prompt.
Test Preparation	
Capture/playback	No explicit support functions
Editor	No explicit support functions
SUT instrumentation	No explicit support functions
Test case re-use	No explicit support functions
Test data	No explicit support functions
Test data re-use	No explicit support functions
Test script languages	No explicit support functions
Test Execution (real-time)	
SUT log monitoring	No explicit support functions
SUT performance/load/statistics monitoring	No explicit support functions
SUT trace monitoring	No explicit support functions
Test case execution monitoring	Test Case Monitoring through Monitor service.
Evaluation (post mortem)	
Execution history analysis	No explicit support functions
SUT log analysis	No explicit support functions
SUT profiling analysis	No explicit support functions
SUT trace analysis	No explicit support functions

G Prototype – User Manual

G.1 Introduction

The usage of the prototype can be divided into three use cases, see Figure 56:

- Test Preparation
- Test Execution
- Test Evaluation

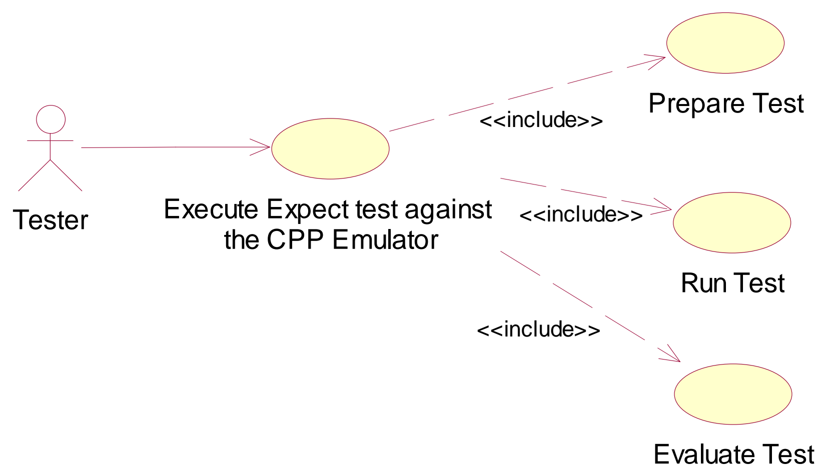


Figure 56: The Prototypes' Three Use Cases

The *Prepare Test* phase includes creating all resources needed for executing a test suite, and changing and editing required settings for the CPP Emulator and RAC. The *Run Test* (or *Execute Test*) is the phase where the test suite remotely executes. The final phase, *Evaluate Test*, is the phase where the user evaluates the test results from the test execution.

In the following instructions, "select" has the same meaning as clicking the left mouse button or pressing the enter key.

G.2 Eclipse Vocabulary

Before using the manual, it is useful and helpful to learn some Eclipse vocabulary. Figure 57 shows the Eclipse main window with a sample project opened.

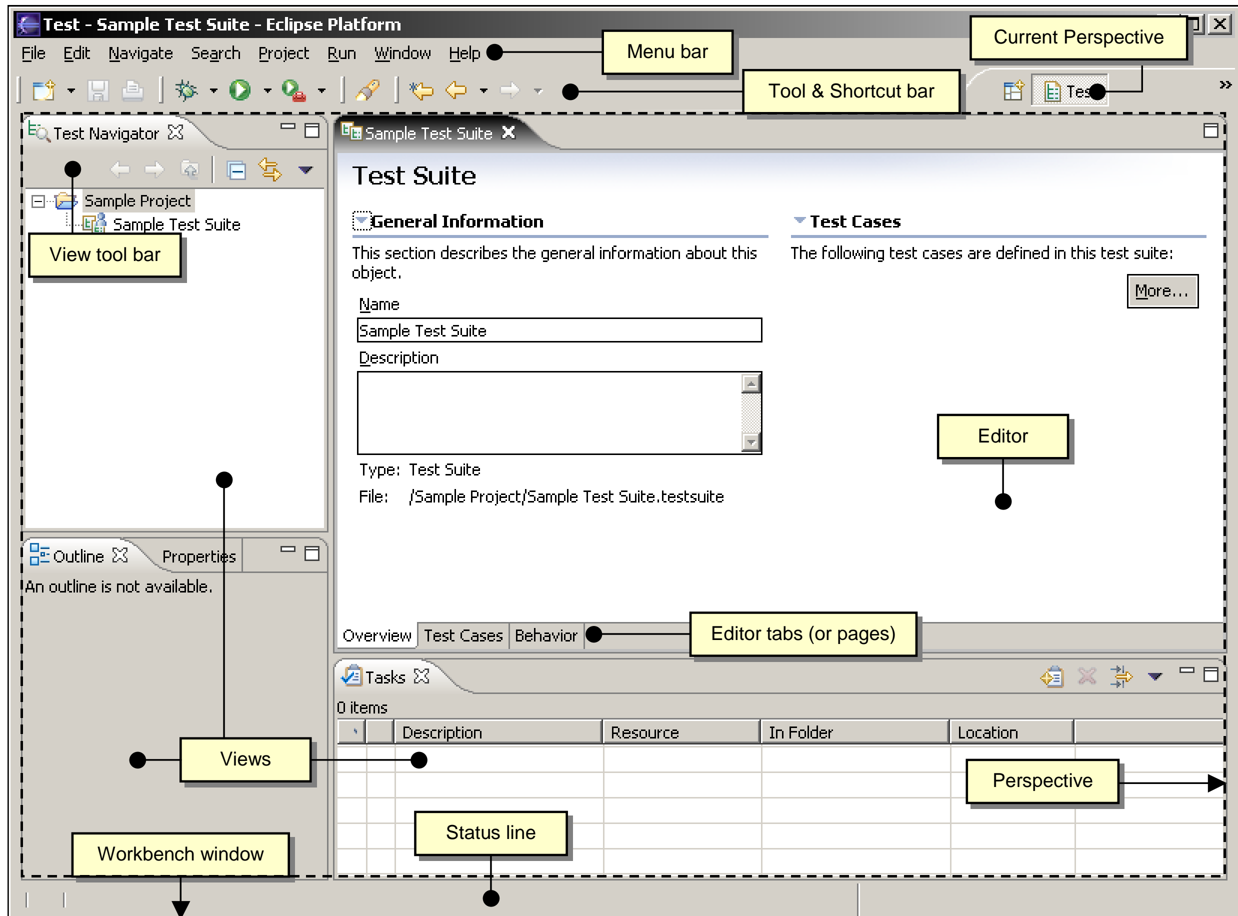


Figure 57: The Main Window of Eclipse

The *Workbench* window is the main window in Eclipse. In the *Workbench* window different *Parts* can be shown. A *Part* is a set of *Views* and *Editors*. Each file (which is represented as a *Resource* in the *Workbench*) has its own *Editor* in order to be displayed and edited correctly. Views support *Editors* and provide alternative presentations or navigations of the information in the Workbench. A predefined layout and initial set of *Views* and *Editors* in the *Workbench* is called a *Perspective*.

When creating and using plug-ins in Eclipse, the terms *Extensions* and *Extension Points* are often used. An *Extension* is used to extend functionality. By adding *Extensions* using *Extension points*, new functionality is contributed to the platform. Defining *Extension Points* enables other plug-ins to make use of the new functionality.

G.3 Eclipse Pre-Defined Architecture of Resources

Eclipse TPTP comes with a pre-defined test architecture. The pre-defined test architecture consists of four different resources:

- Artifact
- Deployment
- Location
- Test Suite

The *Artifact* resource has one or more *Test Suites* resources, where each test suite has one or more test scripts. The test suite editor plug-in of the prototype in this case represents the test suite. The *Location* resource describes the location of the RAC. Finally, the *Deployment* resource associates an *Artifact* with a *Location*. Figure 58 shows the resource architecture of Eclipse TPTP.

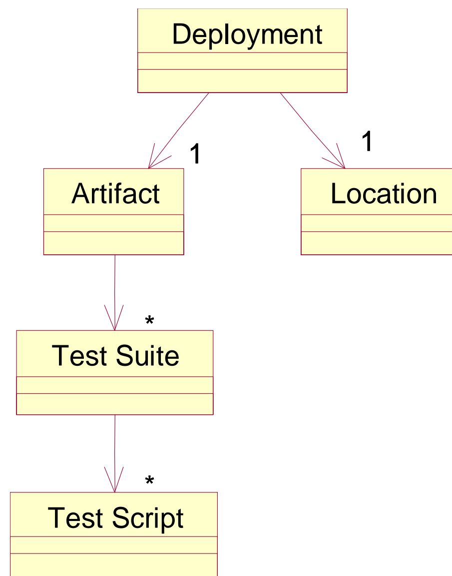


Figure 58: The Eclipse TPTP Resource Architecture

Each resource is created with a Wizard. All available Wizards can be found by selecting "Other" in the "File", "New" menu, see Figure 59.

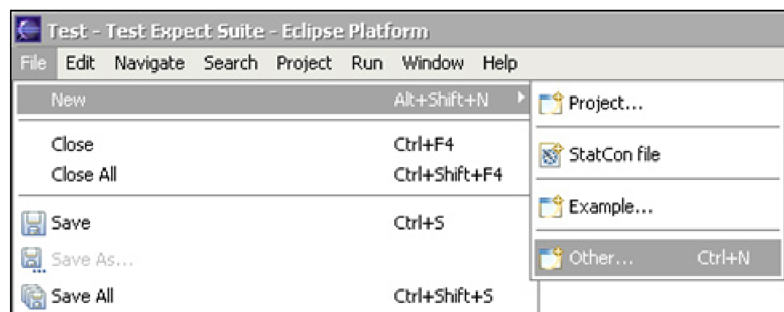


Figure 59: Wizards in Eclipse

G.4 Prepare Test

The use case Prepare Test consists of creating the needed resources to obtain the resource architecture described in Figure 58. When all resources needed for the execution are created,

settings have to be applied to the resource editors. It is important to save continuously when creating the resources since resources are directly saved to and read from the file system. If resources are not saved frequently when created, Eclipse TPTP has a tendency not to understand that changes have been done to the resources.

G.4.1 Changing to the Test perspective

The perspective used when testing is called “Test”. To open the “Test” perspective:

- In the “Window”, “Open Perspective” menu, select “Other…”.
- In the “Select Perspective” dialog, mark the “Test” item and then select the “OK” button, see Figure 60.

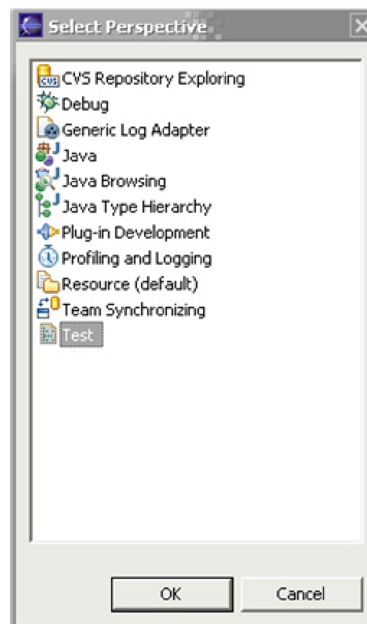


Figure 60: Select Perspective Dialog Window

G.4.2 Creating the Project

A project is needed to hold the resources, so start by creating a Simple Project, see Figure 61:

- In the “File”, “New” menu, select “Project…”.
- Under the Wizard folder “Simple”, mark “Project” and select the “Next” button.

- Enter a name for the project in the field “Project Name”. Select the “Finish” button.

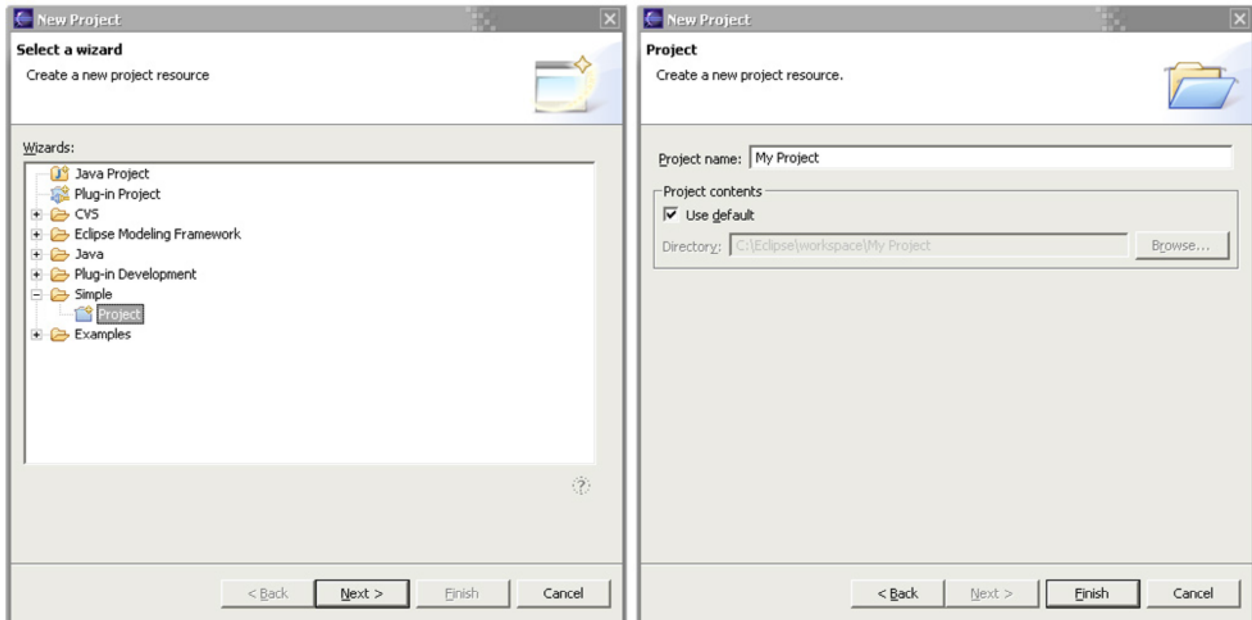


Figure 61: Creating a Simple Project

Another way to create the project is simply by clicking with the right mouse button in the “Test Navigator” view and selecting “Project…” in the “New” menu, given that the “Test” perspective has been selected, see Figure 62.

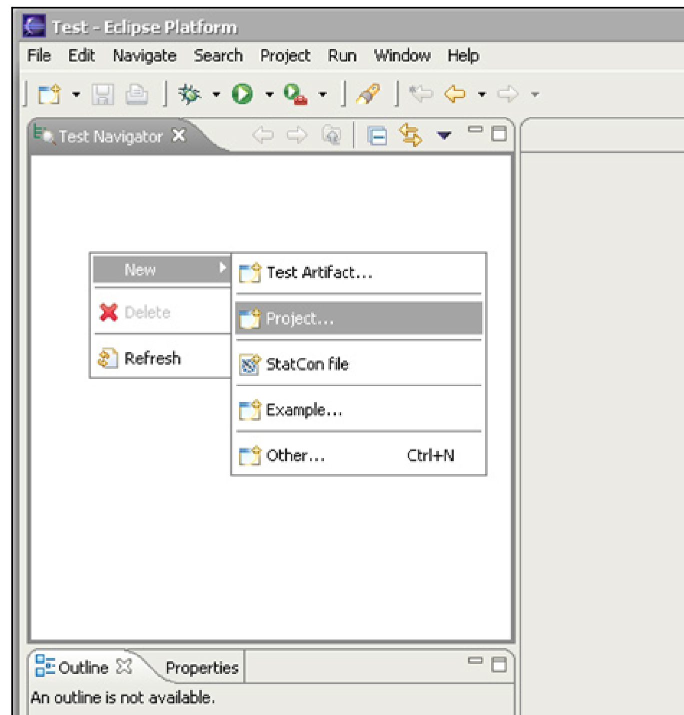


Figure 62: Another Way to create a Project in Eclipse

G.4.3 Creating and Editing the TPTP Expect Test Suite Resource

The next step is to create the TPTP Expect Test Suite resource:

- Click with the right mouse button in the "Test Navigator" view and select "Test Artifact…" in the "New" menu, see Figure 63.

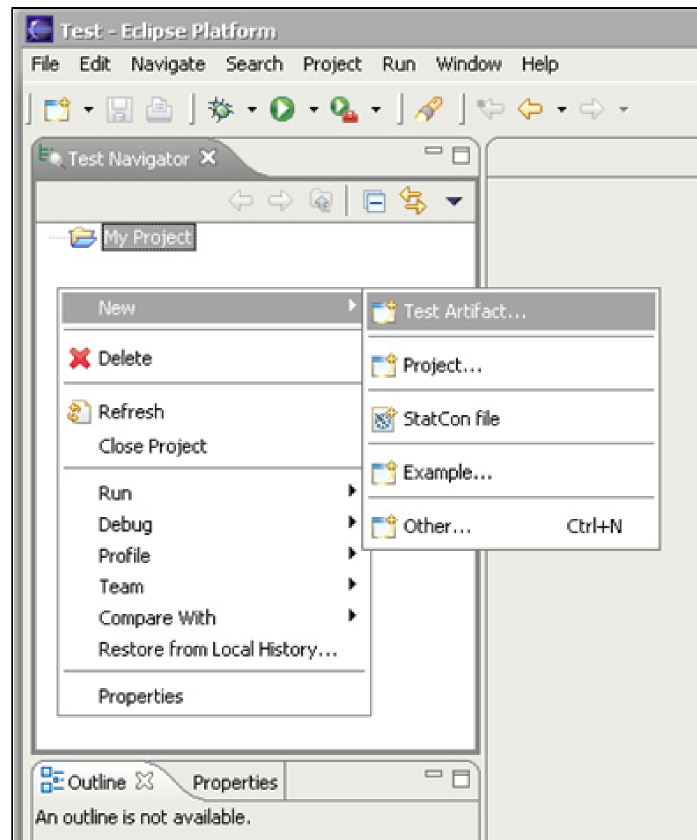


Figure 63: Creating a New Test Artifact

- Under the Wizard folder “Test”, select the folder “Test Suite” and then select “TPTP Expect Test Suite”, see Figure 64.
- Select the “Next” button.
- Enter a name for the Test Suite in the field “File Name”, select “My Project” as parent folder, see Figure 64.
- Select the “Finish” button.

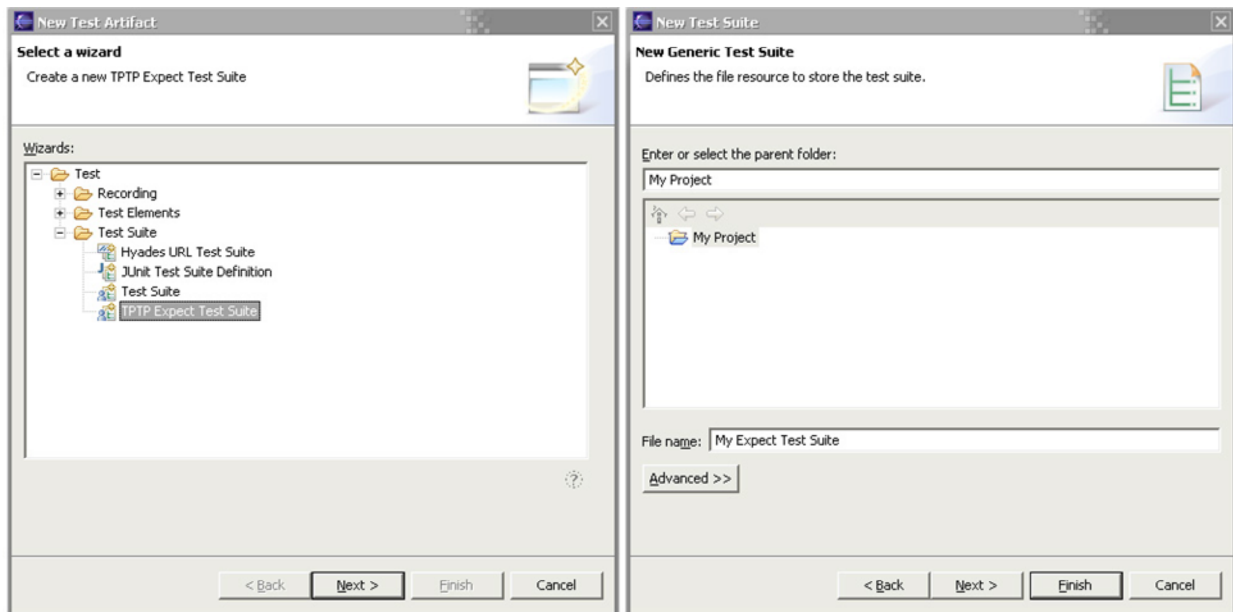


Figure 64: Creating a New TPTP Expect Test Suite

Before creating the next test artifact, the TPTP Expect Test Suite needs to be edited. After the TPTP Expect Test Suite has been created, its editor is opened in Eclipse, see Figure 65.

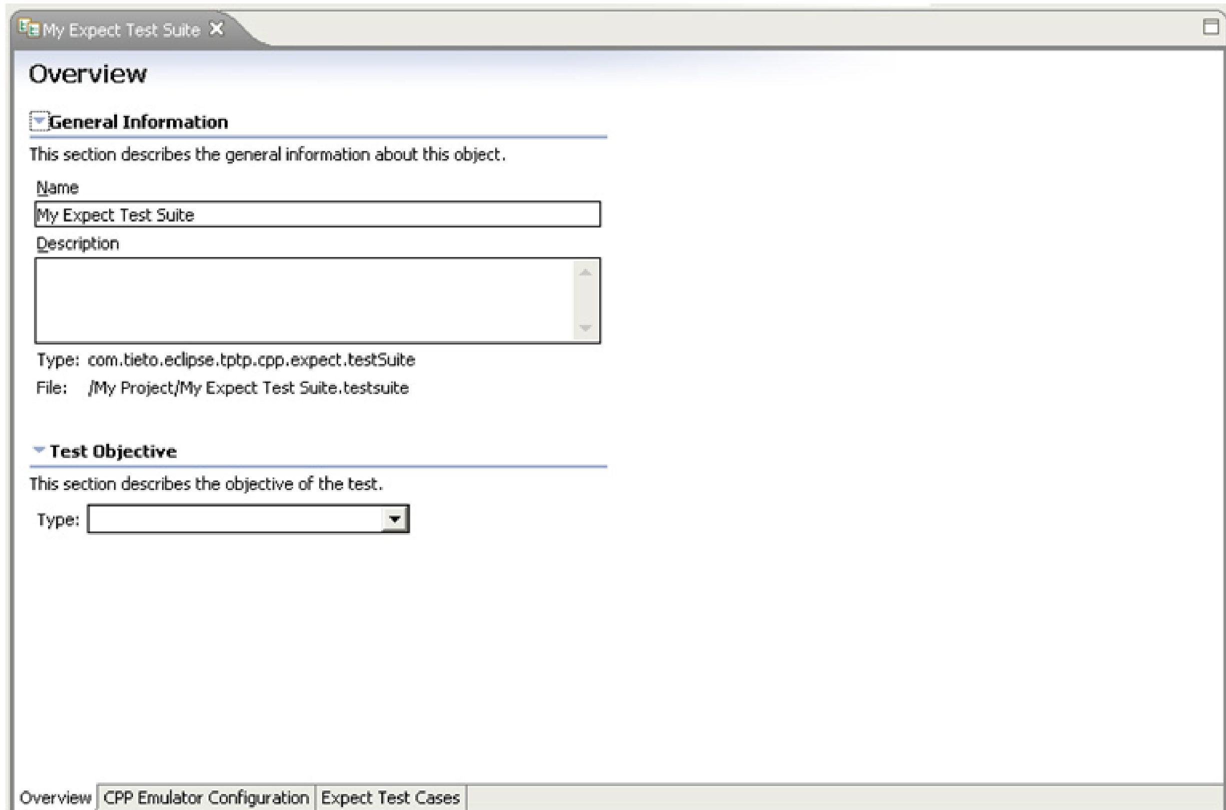


Figure 65: The TPTP Expect Test Suite Editor

The TPTP Expect Test Suite editor has three tabs:

- *Overview* – A default tab with general information and test objective.
- *CPP Emulator Configuration* – A tab with settings regarding the CPP Emulator
- *Expect Test Cases* – A tab for adding Expect test cases

The *CPP Emulator Configuration* tab, Figure 66, has the following fields:

- *Host Name* – The host name or IP address of the computer where the CPP emulator is installed. (In the prototype implementation, this has to be the same computer as running the RAC.)
- *Host Telnet Port* – The port number of the simulated CPP node.

- *ClearCase View* – The ClearCase view which can be used to access the current CPP Emulator (when writing this manual, CPP Emulator R2B is used).
- *.cppemu file* – The CPP Emulator configuration file
- *.persistent file* – The CPP Emulator persistent file
- *.checkpoint file* – The CPP Emulator checkpoint file

Enter the information in each field described above, either a:

- *.cppemu file* is given, or a…
- *.cppemu file* and a *.persistent file* are given, or a…
- *.checkpoint file* is given.

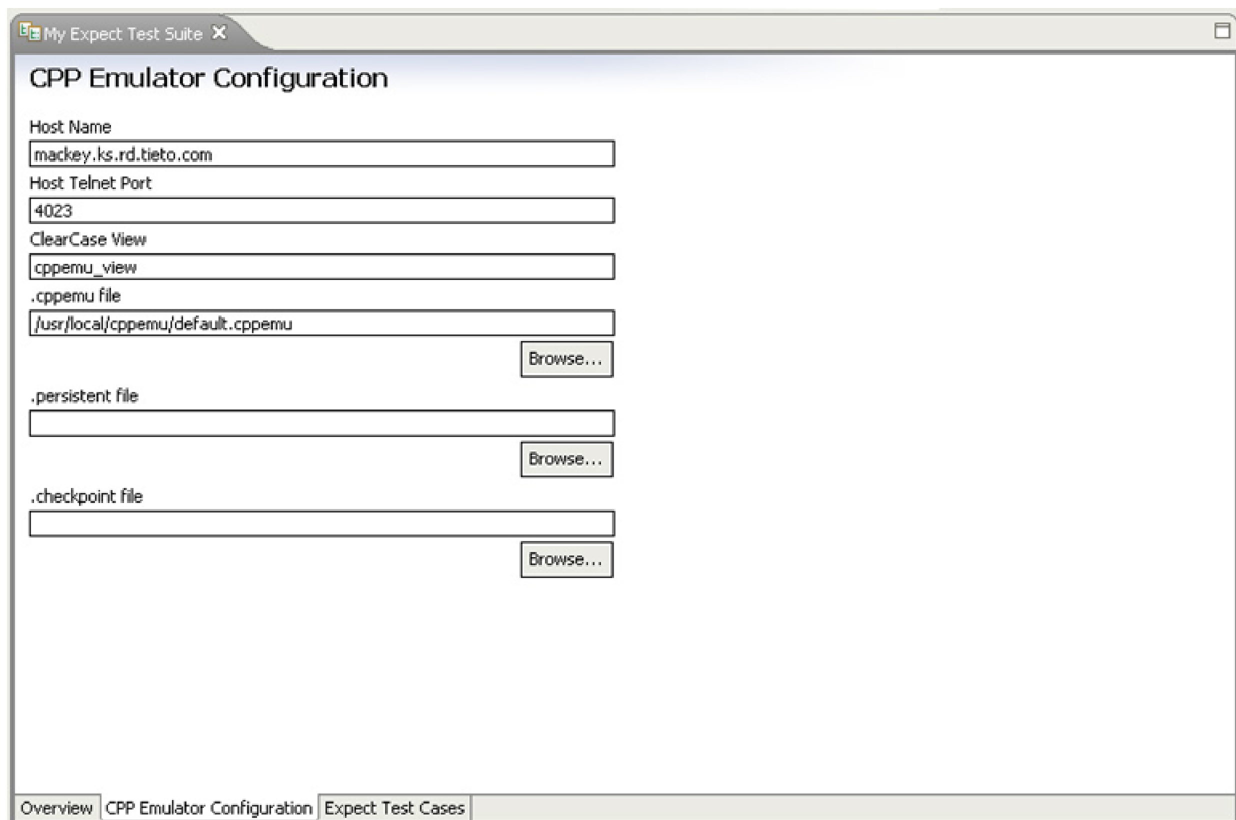


Figure 66: The CPP Emulator Configuration Tab

Figure 67 shows the *Expect Test Cases* tab. To add Expect test cases simply select the “Add” button, and a file browser dialog windows will appear. The added Expect test cases will not be transferred themselves, but only the path and filenames. Since only the path and filenames will be transferred, it is important that the remote machine (where the CPP Emulator and Expect executable is located) can access the Expect script files.

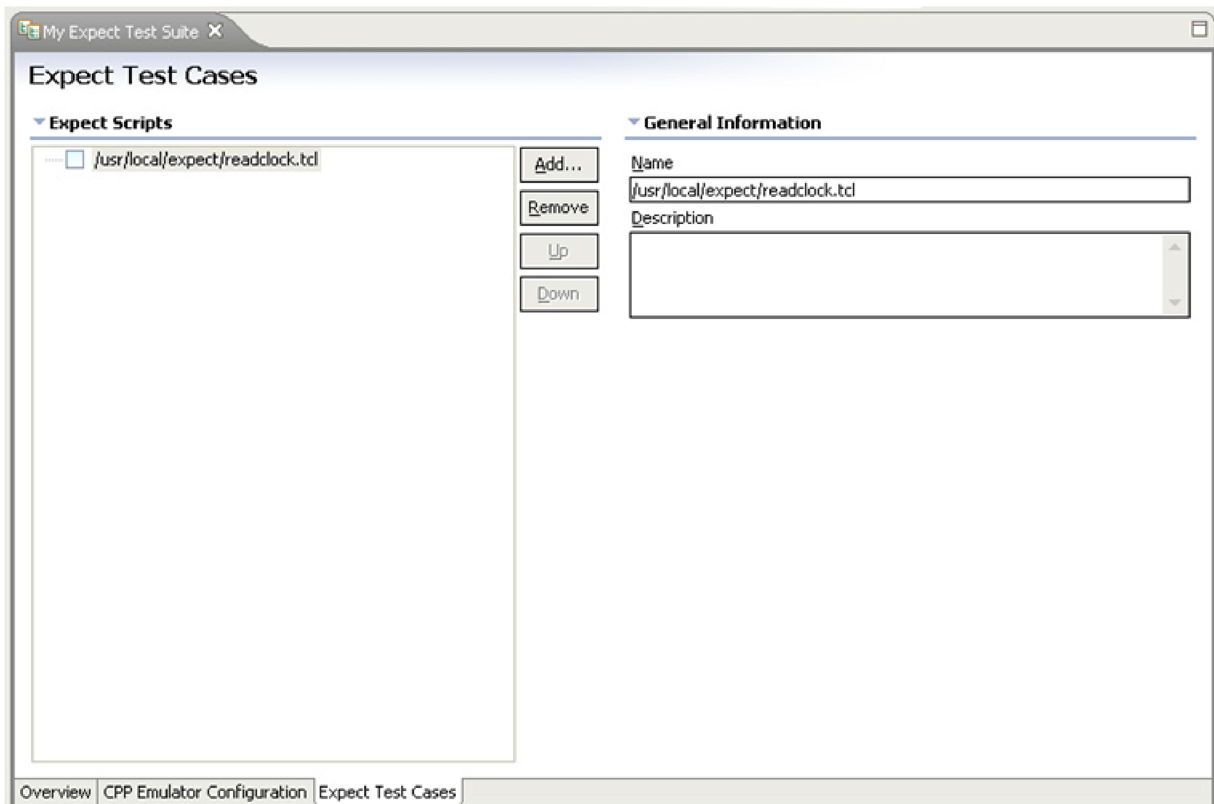


Figure 67: The Expect Test Cases Tab

G.4.4 Creating and Editing the Artifact Resource

The next step is to create the Artifact resource:

- Click with the right mouse button in the “Test Navigator” view and select “Test Artifact…” in the “New” menu, see Figure 63.
- Under the Wizard folder “Test”, select the folder “Test Elements” and then select “Artifact”, see Figure 68.

- Select the “Next” button.
- Enter a name for the Artifact in the field “File Name”, select “My Project” as parent folder, see Figure 68.
- Select the “Finish” button.

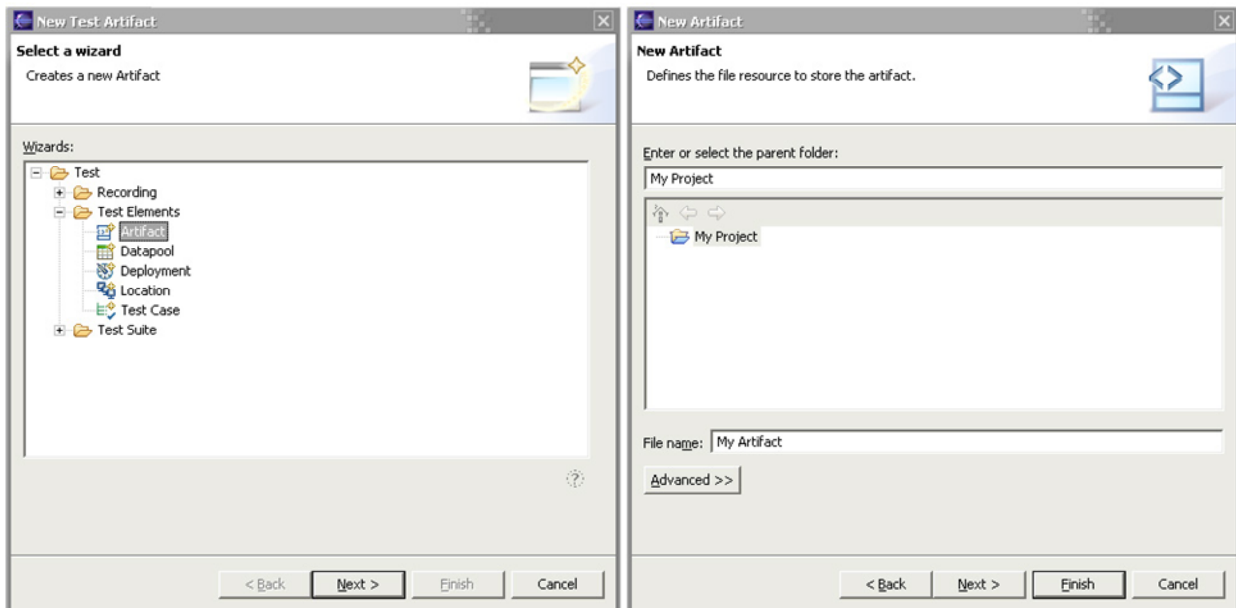


Figure 68: Creating a New Artifact

After the Artifact has been created, its editor is opened in Eclipse, see Figure 69.

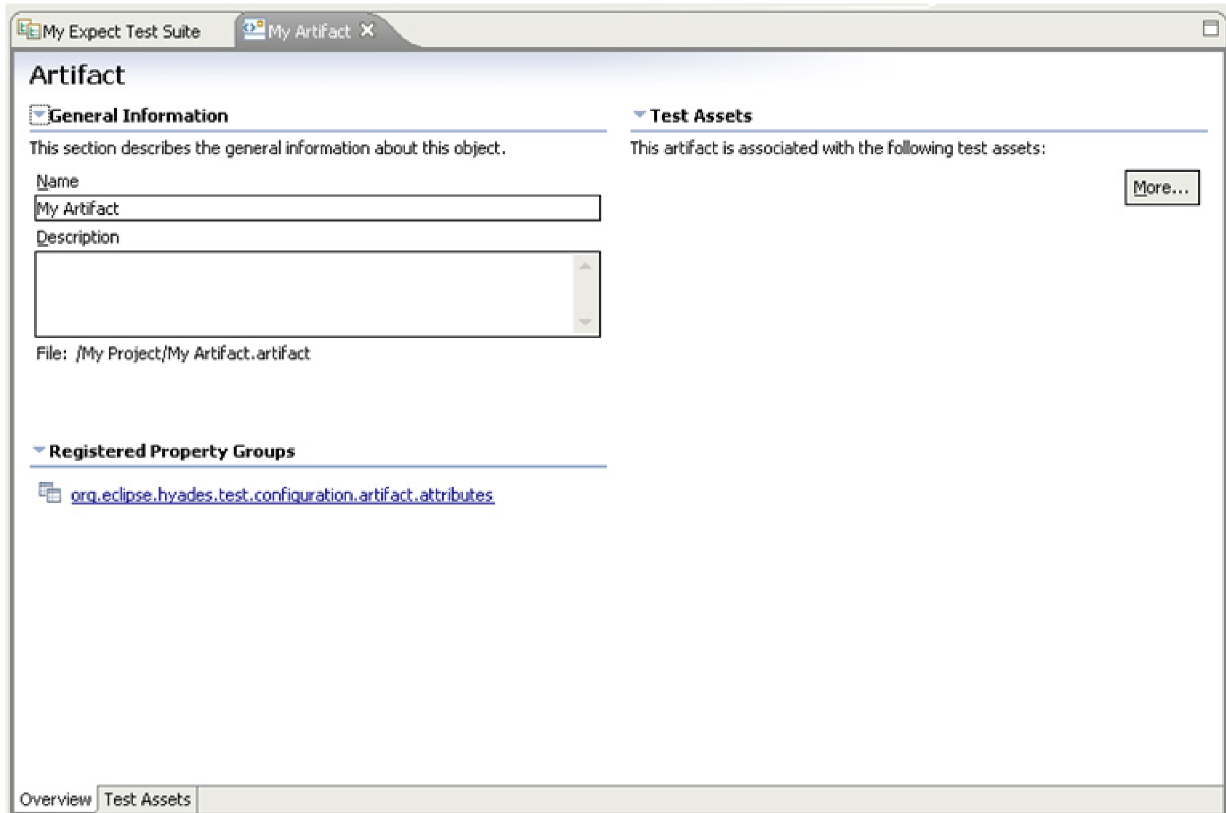


Figure 69: The Artifact Editor

The Artifact editor has two tabs:

- *Overview* – A default tab with general information.
- *Test Assets* – A tab for associating *Test Suites* with the *Artifact*.

To add the *My Expect Test Suite* to the *Test Assets* tab, simply select the “Add” button, see Figure 67, and the “Select Resource” dialog window will appear, see Figure 70. Mark the “My Expect Test Suite.testsuite” and select “OK”.

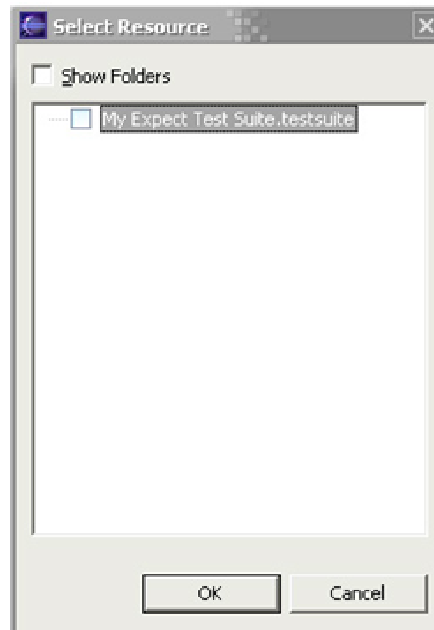


Figure 70: The Select Resource Dialog Window

After the TPTP Expect Test Suite has been selected, the *Test Assets* tab should look like Figure 71.

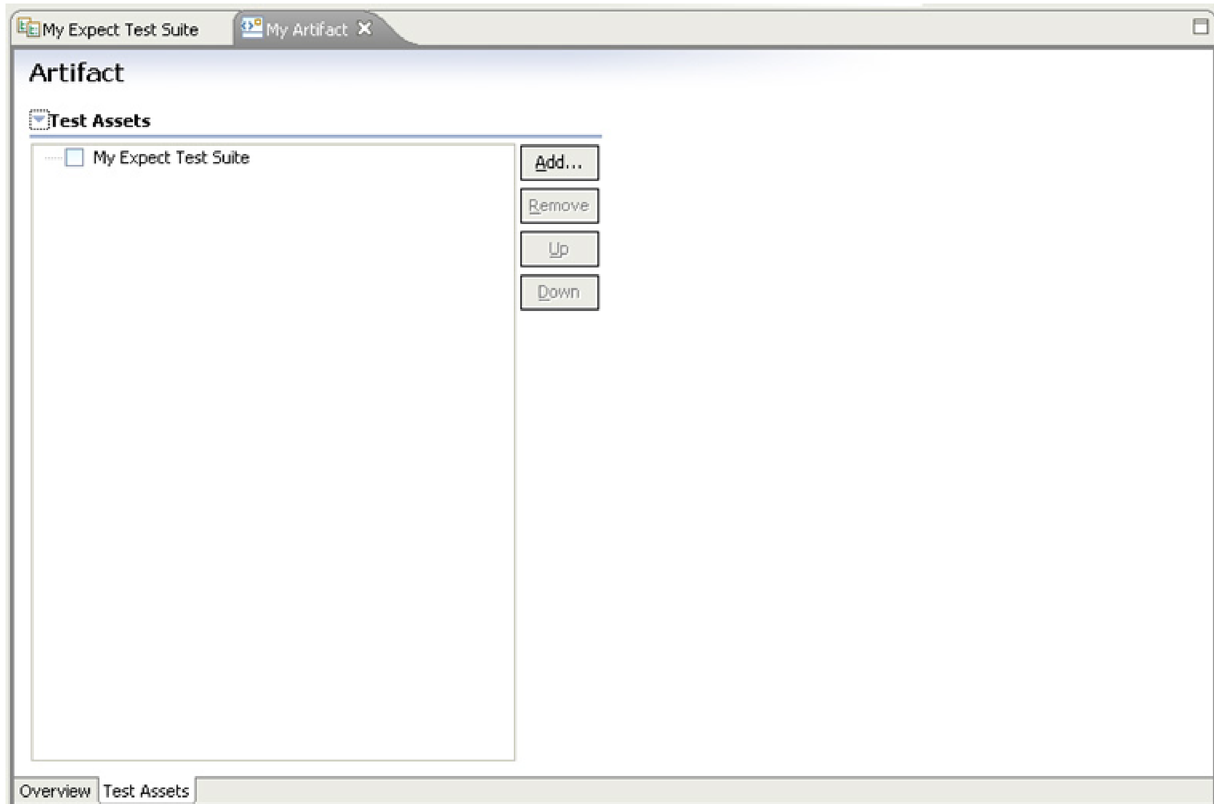


Figure 71: The Test Assets Tab

G.4.5 Creating and Editing the Location Resource

The next step is to create the Location resource:

- Click with the right mouse button in the "Test Navigator" view and select "Test Artifact…" in the "New" menu, see Figure 63.
- Under the Wizard folder "Test", select the folder "Test Elements" and then select "Location", see Figure 72.
- Select the "Next" button.
- Enter a name for the Location in the field "File Name", select "My Project" as parent folder, see Figure 72.
- Select the "Finish" button.

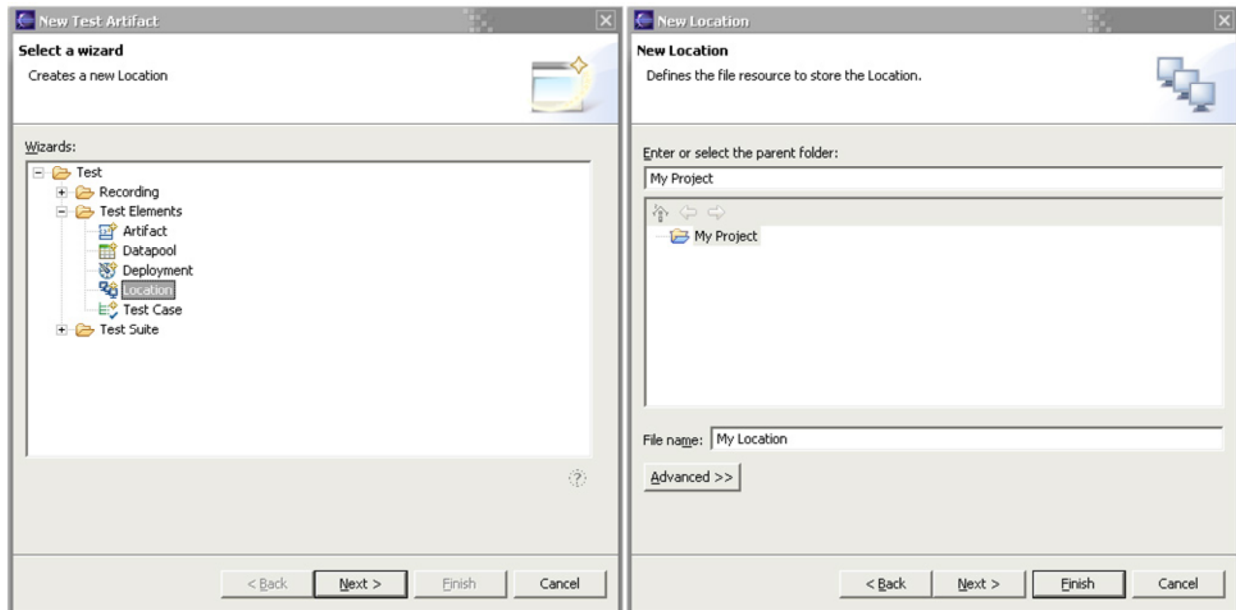


Figure 72: Creating a New Location

After the Location has been created, its editor is opened in Eclipse, see Figure 73.

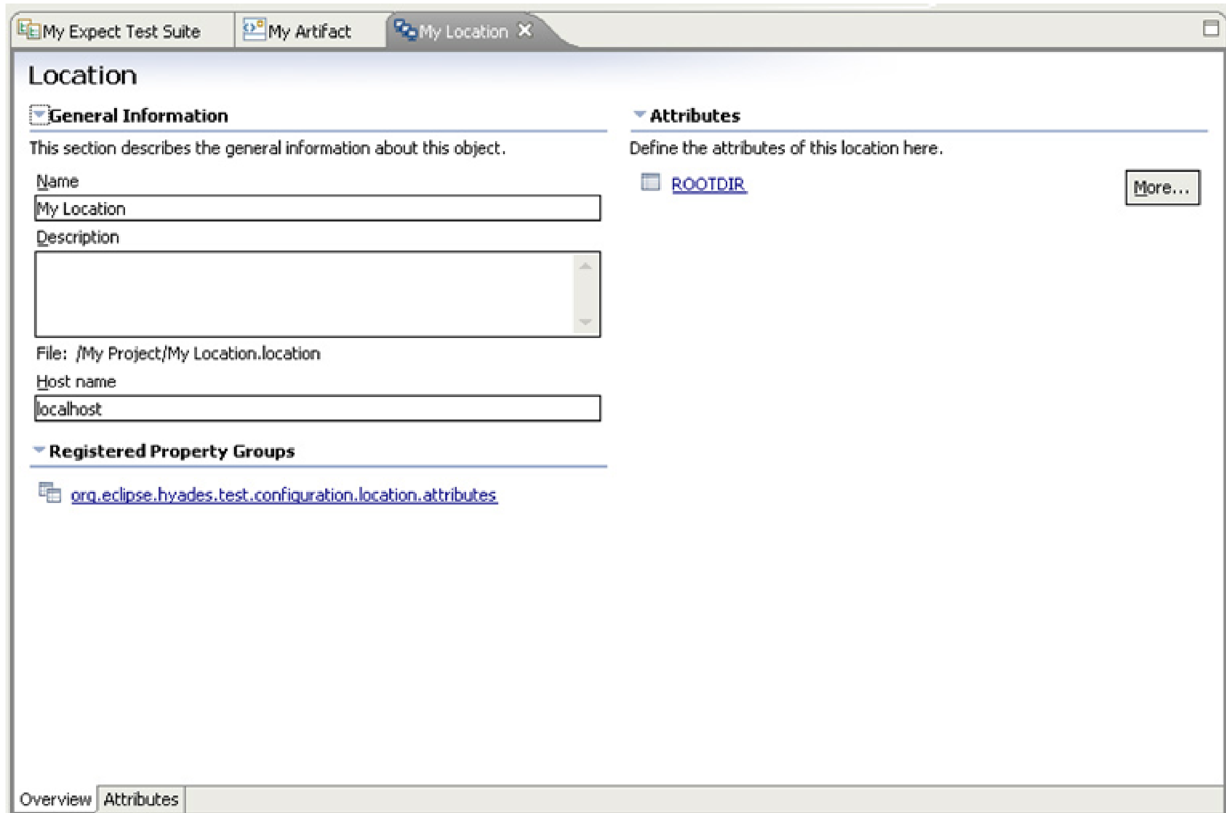


Figure 73: The Location Editor

The Location editor has two tabs:

- *Overview* – A default tab with general information.
- *Attributes* – A tab for setting attributes of the Location.

The only setting that has to be edited on the *Location* resource is the *Host name* field on the *Overview* tab. The *Host name* represents the machine running the RAC.

G.4.6 Creating and Editing the Deployment Resource

The next, and final step, is to create the Deployment resource:

- Click with the right mouse button in the “Test Navigator” view and select “Test Artifact…” in the “New” menu, see Figure 63.

- Under the Wizard folder "Test", select the folder "Test Elements" and then select "Deployment", see Figure 74.
- Select the "Next" button.
- Enter a name for the Deployment in the field "File Name", select "My Project" as parent folder, see Figure 74.
- Select the "Finish" button.

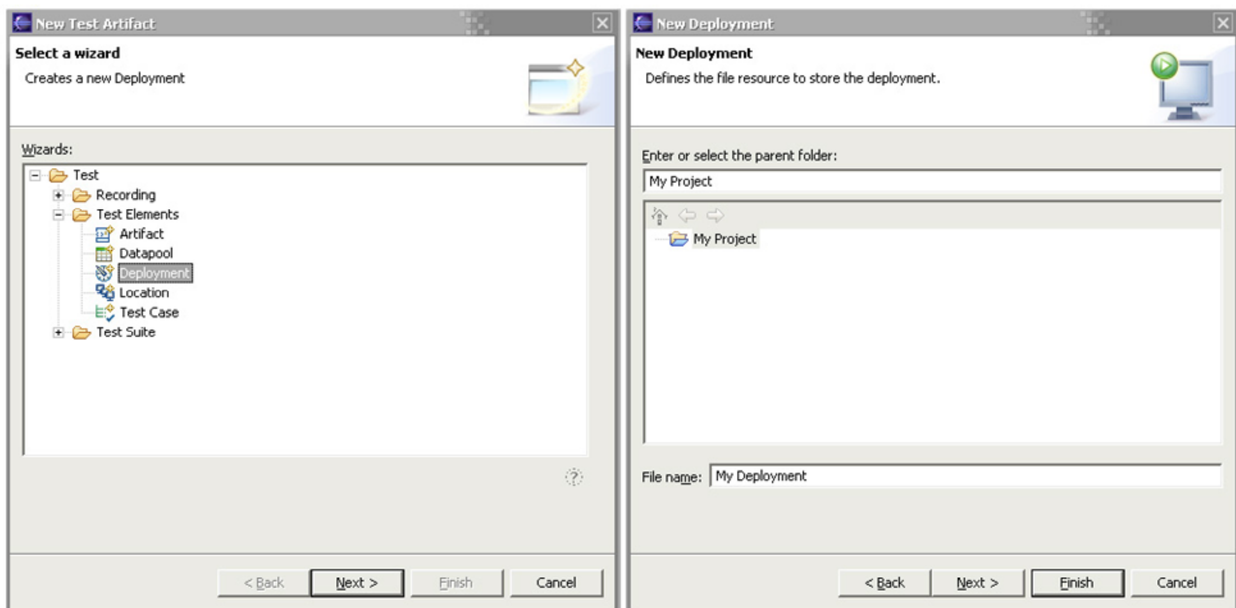


Figure 74: Creating a New Deployment

After the Deployment has been created, its editor is opened in Eclipse, see Figure 75.

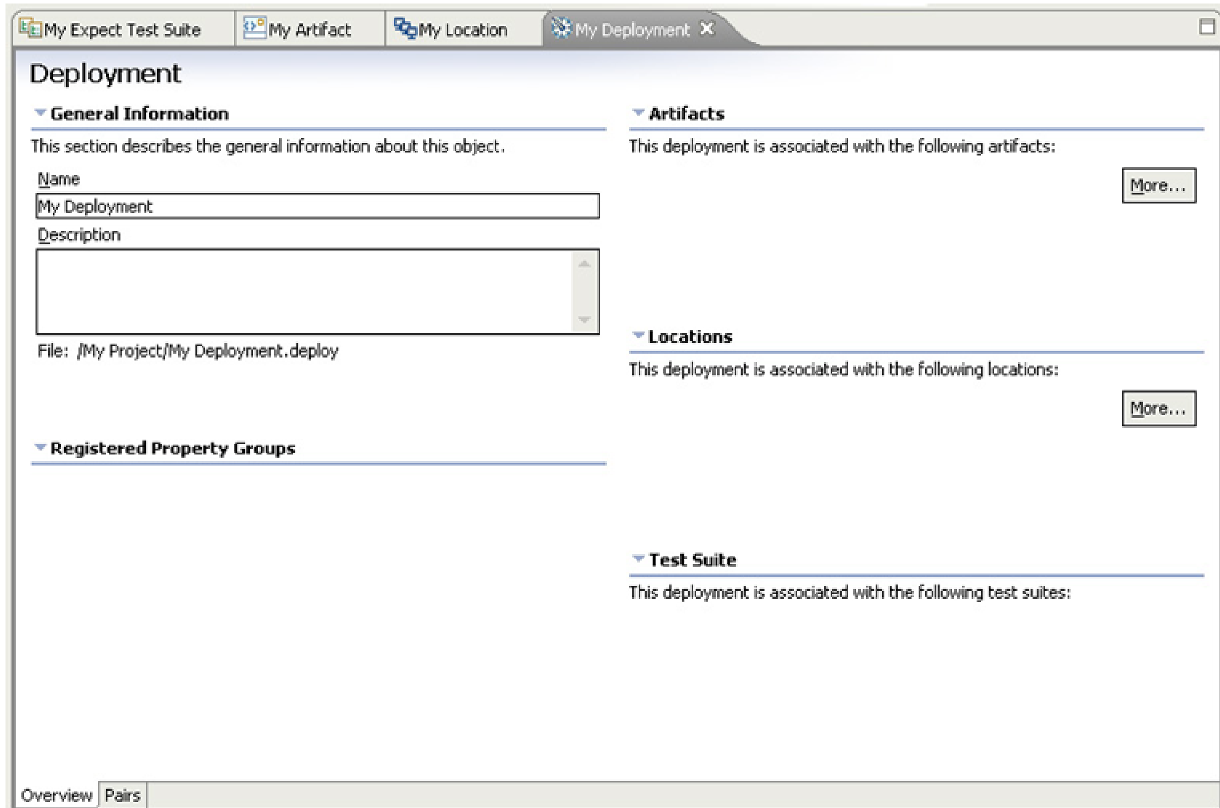


Figure 75: The Deployment Editor

The Location editor has two tabs:

- *Overview* – A default tab with general information.
- *Pairs* – A tab used to pair the *Artifact* and the *Location* together.

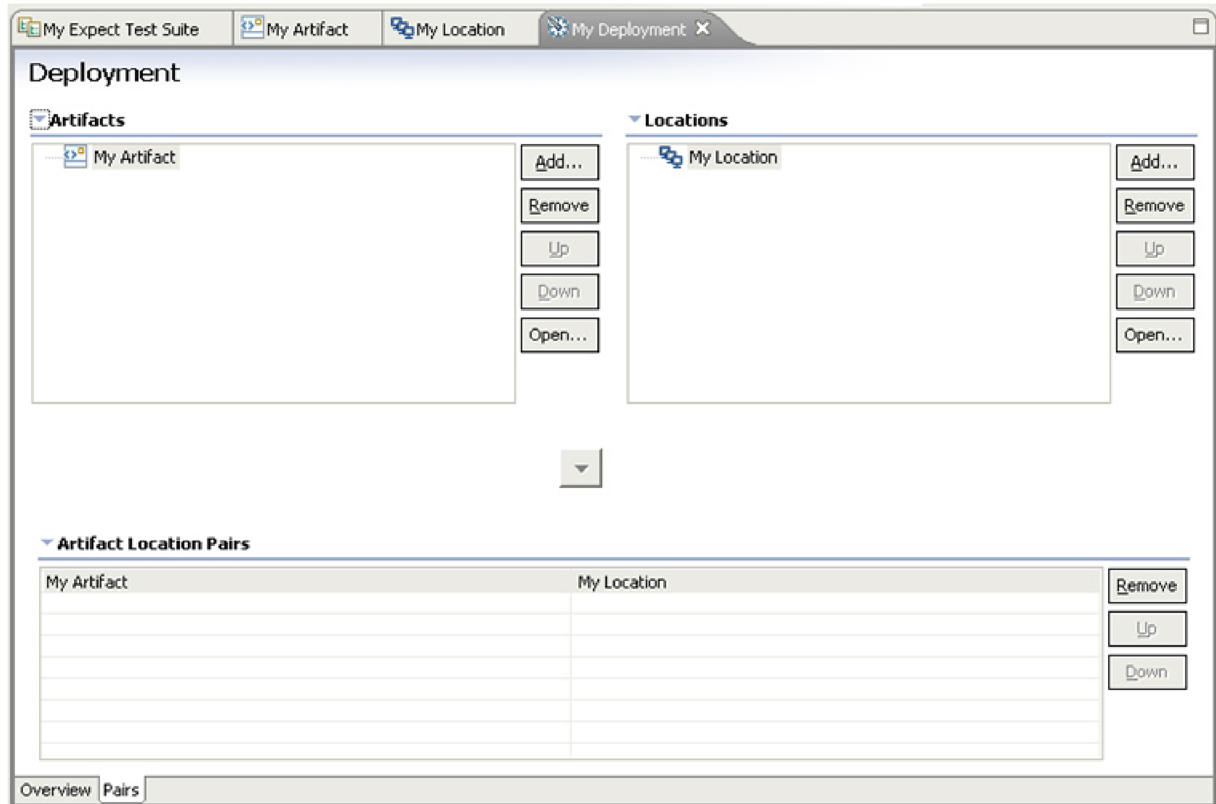


Figure 76: The Pairs Tab

To pair *My Artifact* and *My Location* together on the *Pairs* tab, see Figure 76:

- Select the "Add" button under the "Artifact" section. The "Add Artifact Association" dialog window will appear, see Figure 77.
- Confirm that "Use an existing resource" is marked, and select the "Browse…" button. The "Select resource" dialog window will appear.
- Mark the "My Artifact.artifact" and select "OK", see Figure 78.
- Select the "Finish" button.

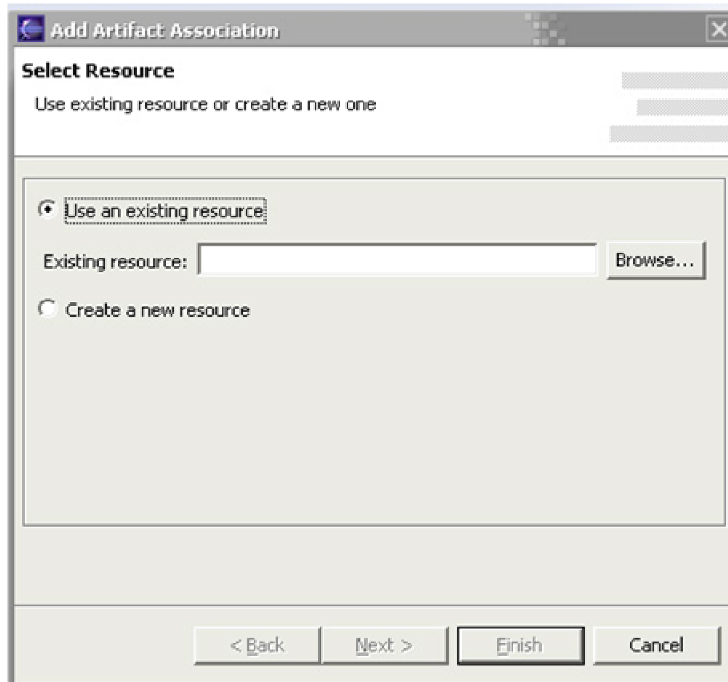


Figure 77: The Add Artifact Dialog Window

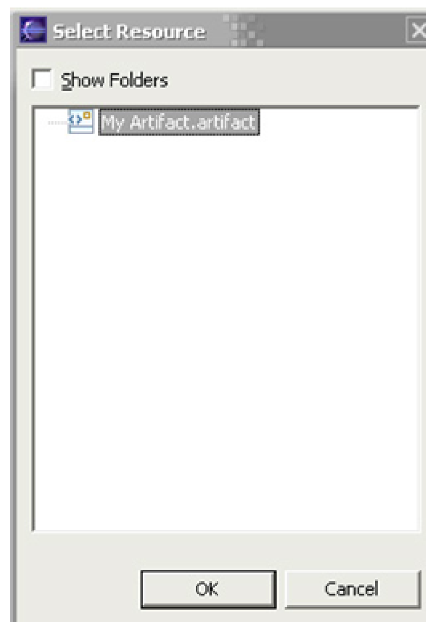


Figure 78: The Select Resource Dialog Window

- In a similar way as with the *Artifact*, select the "Add" button under the "Locations" section, see Figure 76. The "Add Location Association" dialog window will appear, see Figure 79.
- Confirm that "Use an existing resource" is marked, and select the "Browse…" button. The "Select resource" dialog window will appear.
- Mark the "My Location.location" and select "OK", see Figure 80.
- Select the "Finish" button.

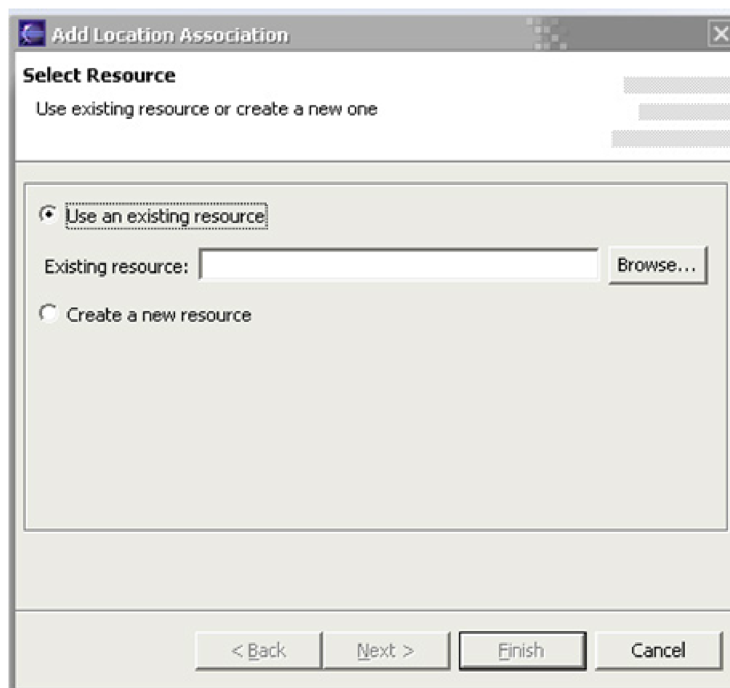


Figure 79: The Add Location Dialog Window

Figure 80: The Select Resource Dialog Window

Now that the *Deployment* has been associated with the *Artifact* and the *Location*, the *Artifact* and *Location* has to be paired. To pair the *Artifact* and *Location*, select the small button with and arrow pointing downwards on the *Pairs* tab. The *Pairs* tab should now look like Figure 81.

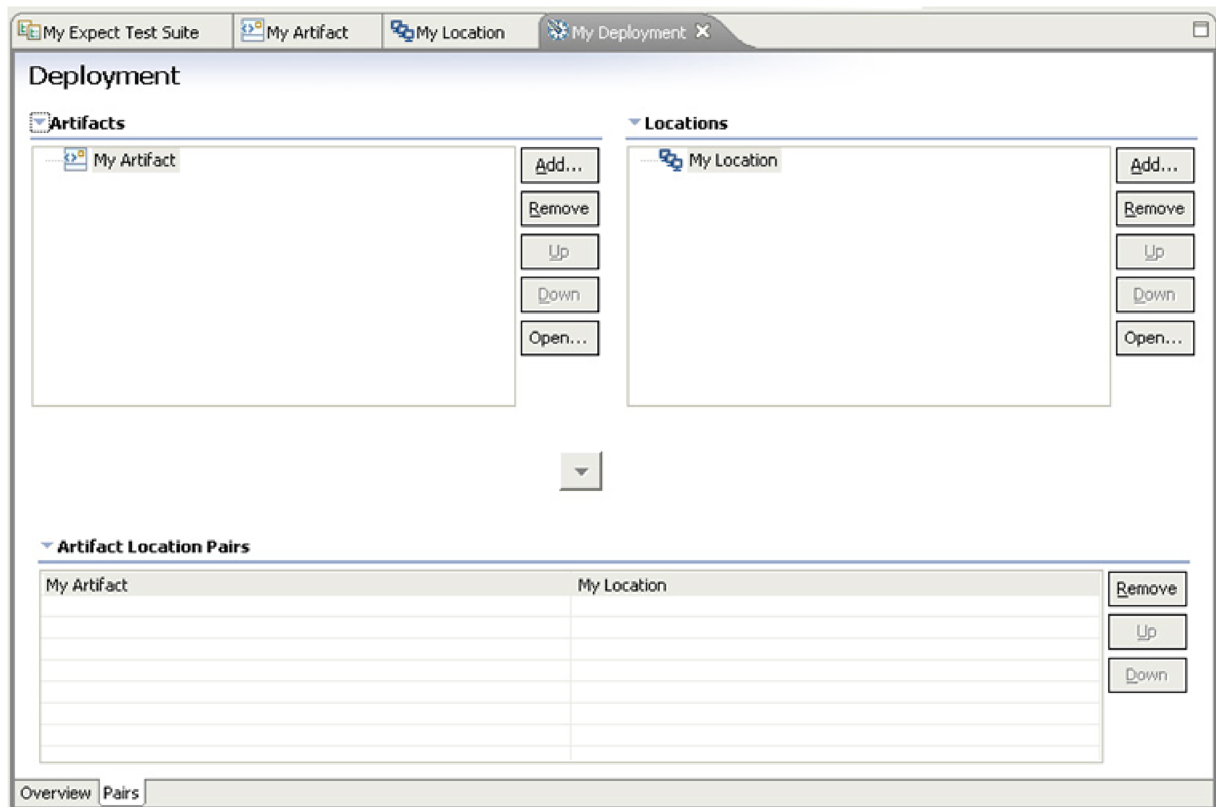


Figure 81: The Pairs Tab

The *Overview* tab should now look like Figure 82.

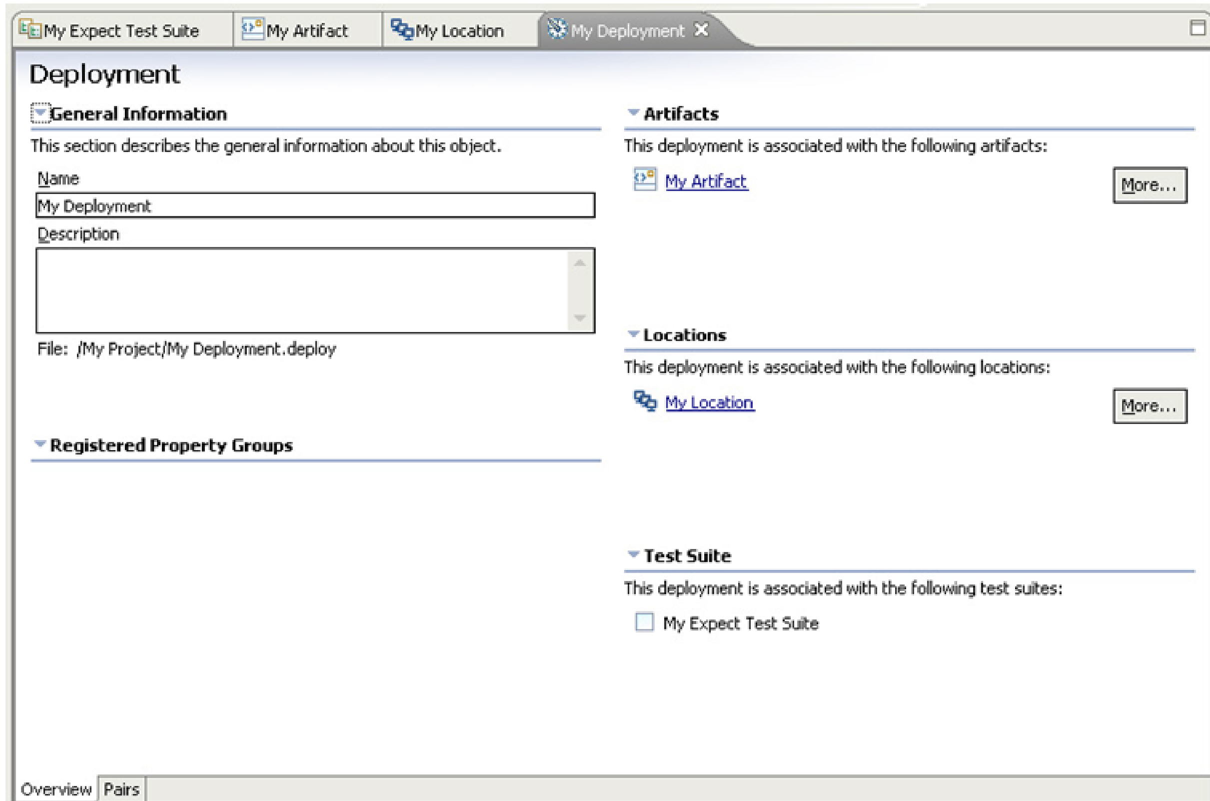


Figure 82: The Overview Tab

The Preparation of the test should now be completed. The "Test Navigator" window should now contain, see Figure 83:

- My Artifact
- My Deployment
- My Expect Test Suite
- My Location

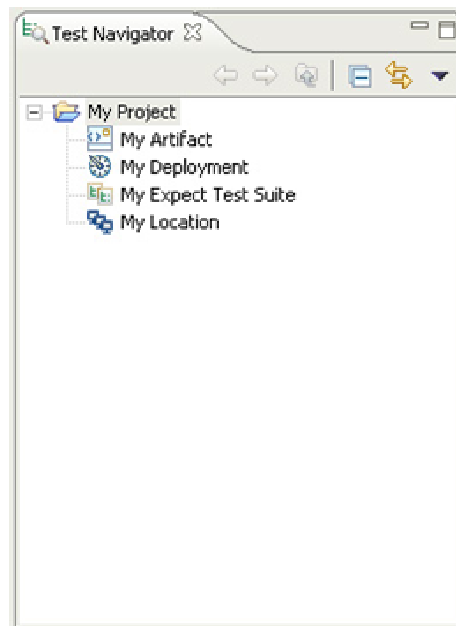


Figure 83: The Test Navigator after Test Preparation

G.5 Run Test

After the *Prepare Test* phase it is time to run the test. To run the test:

- Click with the right mouse button on “My Expect Test Suite” and select “Run…” in the “Run” menu, see Figure 84. The “Run” dialog window will appear.

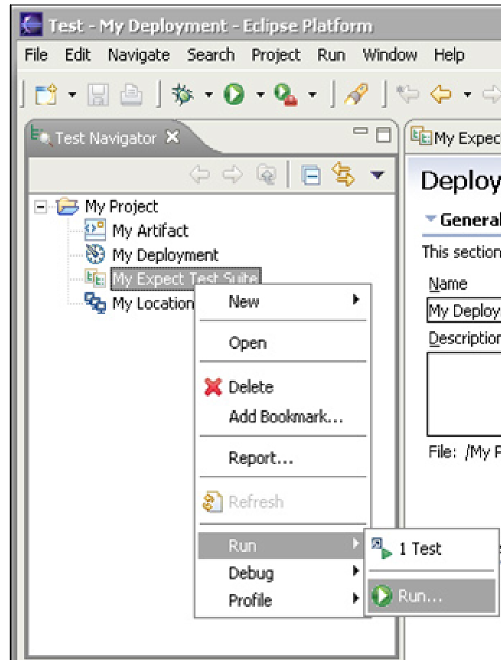


Figure 84: How to Open the Run Dialog Window

To run *My Expect Test Suite*, a new test configuration has to be created. To create a new test configuration:

- In the “Configurations” window in the “Run” dialog window, mark “Test”.
- Select the “New” button and a new test configuration will be created.
- Under the “Test” tab in the “Run” dialog window, mark “My Expect Test Suite”. The “My Deployment” deployment will appear.
- Mark “My Deployment” and select the “Apply” button.
- Finally, select the “Run” button and the test will start to run.

Figure 85 shows how the “Run” dialog window will look like after the described steps (just before selecting the “Run” button).

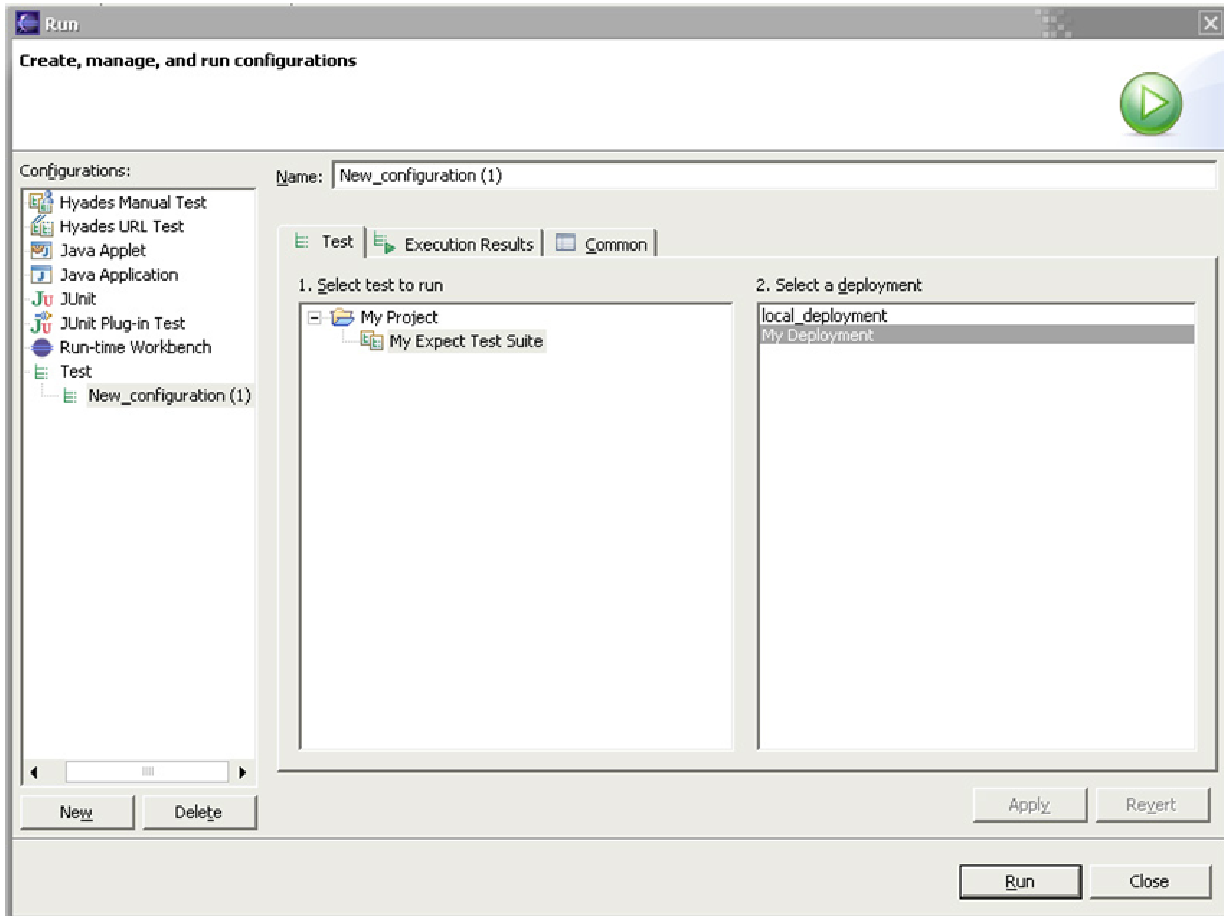


Figure 85: The Run Dialog Window

The *Run Test* phase consists of three parts:

- Launch of the CPP Emulator
- Running all Expect test scripts associated with the TPTP Expect test suite
- Teardown of the CPP Emulator

The test execution will take several minutes, during the test execution the test result will be sent back from the agent and RAC to the Eclipse Workbench.

G.6 Evaluate Test

After the *Run Test* phase is completed the test results can be evaluated. The test result is represented with a test execution resource, which will be created automatically during the *Run Test* phase. The test execution resource will automatically be named based on the name of the test suite, and will show itself in the *Test Navigator*, see Figure 86.

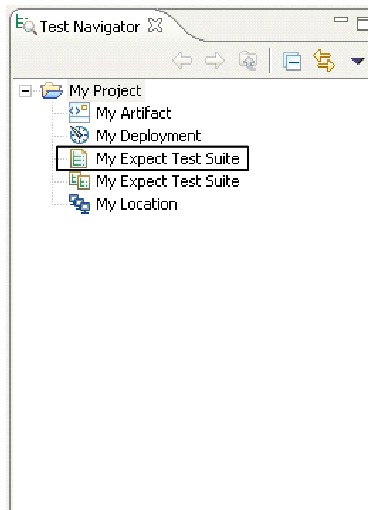


Figure 86: My Expect Test Suite Test Execution Resource

The test execution result includes four states (test verdicts) [33]:

- Pass
- Failed
- Error
- Inconclusive

Where *Pass* means that the test(s) could be executed successfully and passed. *Failed* means that the test(s) could be executed successfully, but did not pass. *Error* means that the test(s) could not be executed, meaning that something failed in the test bed. Finally, *inconclusive* corresponds to when the test(s) is/are still running.

To open the *My Expect Test Suite* test execution resource:

- Double click with the left mouse button on the *My Expect Test Suite* test execution resource in the *Test Navigator*, see Figure 86.
- The *My Expect Test Suite* test execution editor is opened in Eclipse, see Figure 87.

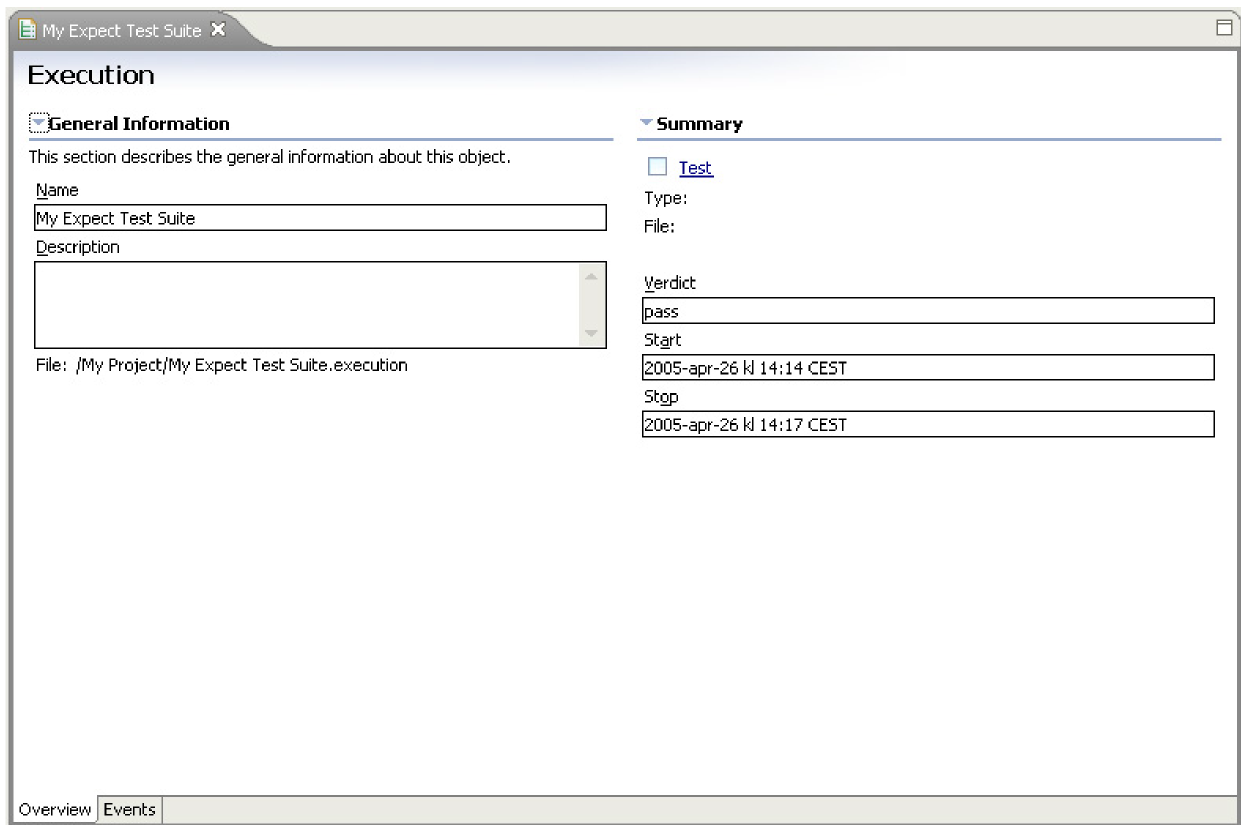


Figure 87: The My Expect Test Suite Test Execution Editor

The Location editor has two tabs:

- *Overview* – A default tab with general information.
- *Events* – A tab describing the test execution result.

The *Overview* shows the overall verdict, in Figure 87 pass. The *Events* tab, see Figure 88, contains a more detailed, graphical view of the test execution.

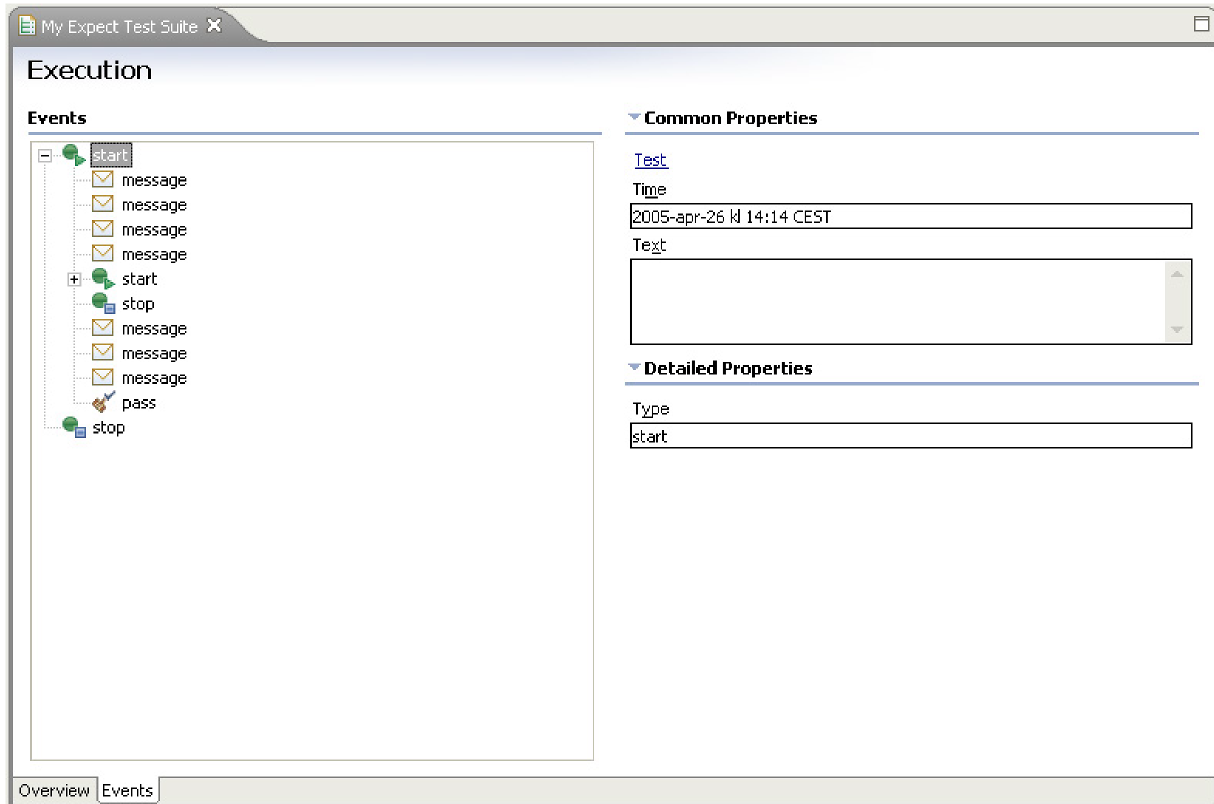


Figure 88: The Events Tab

Each element that has a small *plus sign* ('+') to the left, can be expanded to show more detailed information by clicking on the *plus sign* with the left mouse button. Figure 89 shows *My Expect Test Suite* expanded.

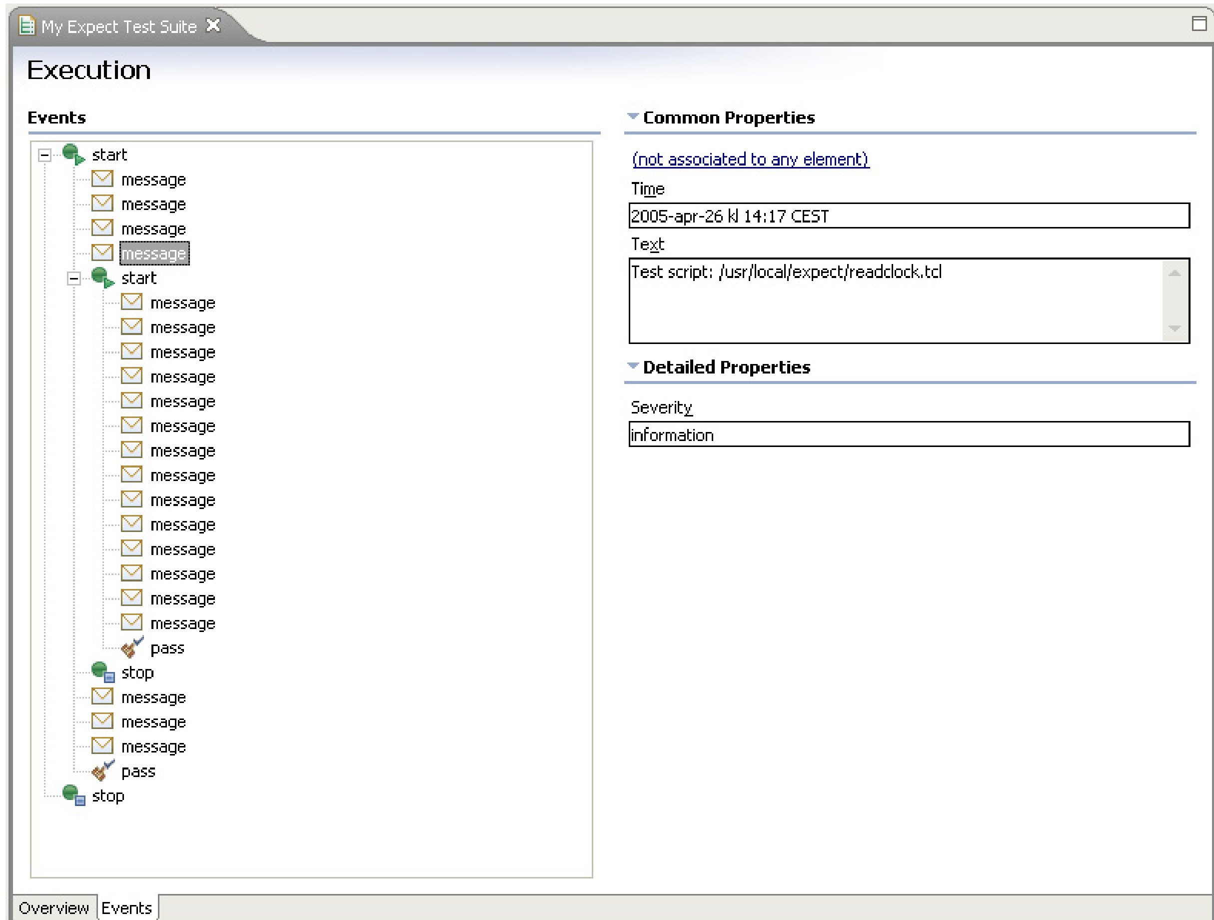


Figure 89: The Events Tab, with a Collapsed View

G.6.1 Test Execution Structure

Each test execution of the test suite created by the TPTP Expect Test Suite wizard has a similar structure of the test execution:

- A Start container for the test execution (expandable)
- The initial messages part (three messages)
    - A message describing which ClearCase view that will be started
    - A message indicating that the CPP Emulator is about to start
    - A message indicating that the execution of tests is about to start
- The test script part (which will be repeated for each test script in the test suite)

- A message describing the path and name of the test script that is about to run
- A start container for the test script (expandable)
- The messages generated by the test script
- A test verdict describing the result of the test script execution
- A stop container for the test script

- The last messages part (three messages)
  - A message indicating that the execution of tests is finished
  - A message indicating that the CPP Emulator is about to teardown
  - A message indicating that the started ClearCase view is about to be closed
- A summarized test verdict describing the total result of the test execution
- A Stop container for the test execution

G.6.2 Exporting the Test Execution Result

The test execution is saved in the Eclipse workspace. The project name is created as a folder (directory) in the file system in the Eclipse workspace folder. Under the project folder all project resources including the test execution can be found. The test execution resource has the file extension '.execution'. Eclipse stores the resources in Zip files. To view the test execution resource file:

- Rename the "My Expect Test Suite.execution" file to "My Expect Test Suite.execution.zip".
- Open the "My Expect Test Suite.execution.zip" file in WinZip or another Zip compliant program, see Figure 90. The "My Expect Test Suite.execution.zip" should contain a file named "ResourceContents".
- Unzip the "ResourceContents" file to a folder of your choice. The "ResourceContents" file contains XML fragments. A suitable text editor for viewing "ResourceContents" is an editor with XML high-lightning support, such as the freeware editor ConTEXT.
- Open the "ResourceContents" file with a text editor of your choice, see Figure 91.
- Do not forget to rename the "My Expect Test Suite.execution.zip" file back to "My Expect Test Suite.execution" after unzipping the "ResourceContents" file.

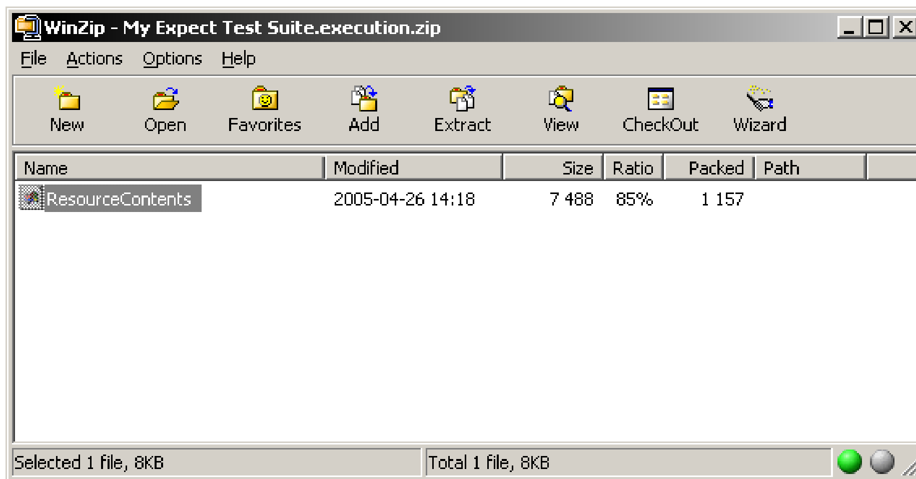


Figure 90: WinZip Showing the "My Expect Test Suite.execution.zip" File

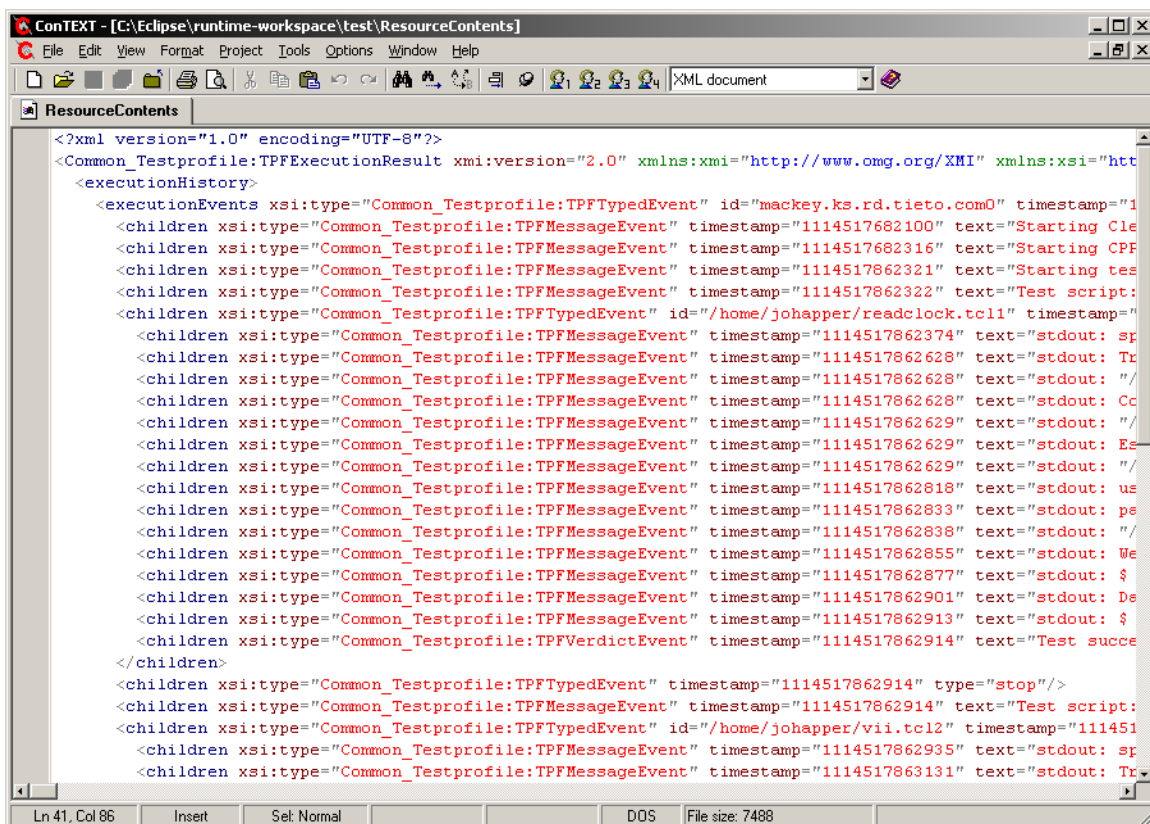


Figure 91: ConTEXT Showing the "ResourceContents" File

H Prototype – Installation Instruction

H.1 Introduction

The installation of the prototype includes two parts: installation of the plug-ins on the client side, the Eclipse Workbench, and installation of the plug-in on the remote side, the RAC. The distribution of the prototype consists of the following:

- Eclipse TPTP Client plug-ins
    - com.tieto.eclipse.tptp.cpp.expect.core_1.0.0
    - com.tieto.eclipse.tptp.cpp.expect.ui_1.0.0
- Eclipse TPTP RAC plug-in
    - com.tieto.eclipse.tptp.cpp.expect
- Eclipse TPTP RAC start/stop script

Figure 92 shows the Eclipse TPTP conceptual architecture and the prototype plug-ins.

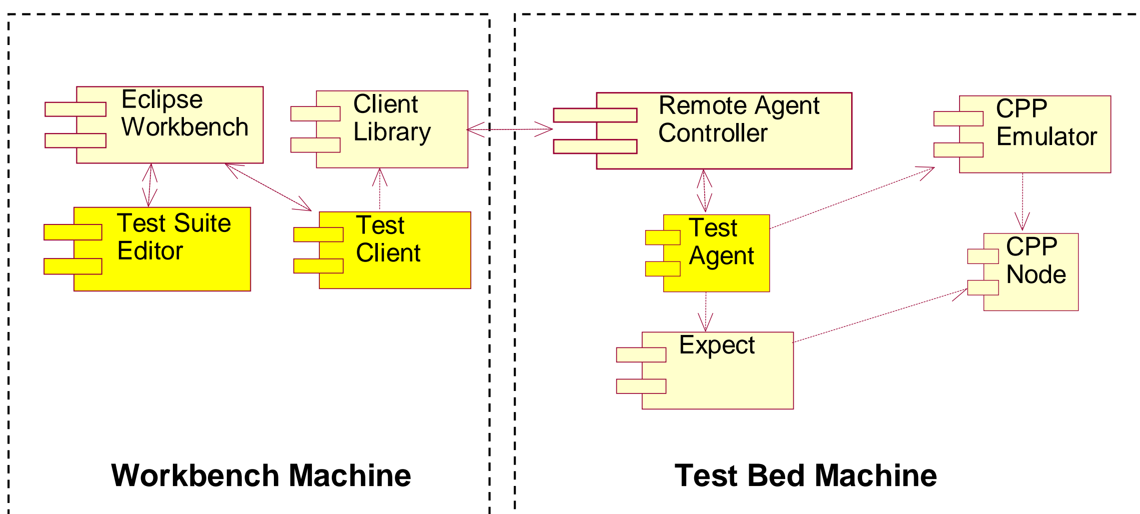


Figure 92: Plug-ins in the Eclipse TPTP Architecture

The Test Client plug-in (com.tieto.eclipse.tptp.cpp.expect.core) makes up the core functionality for launching and executing the Expect test suites. The Test Suite Editor plug-in (com.tieto.eclipse.tptp.cpp.expect.ui) incorporates the graphical components for creating and editing the TPTP Expect Test Suite in the Eclipse Workbench.

H.2 Requirements

The prototype requires the following software on the client side:

- Java Runtime Environment (JRE) or Java Development Kit (JDK) 1.4.2
- Eclipse SDK 3.0.2
- Eclipse Modeling Framework (EMF) SDK 2.0.2
- XML Schema Infoset Model (XSD) SDK 2.0.2
- TPTP 3.2A Runtime

During the development and testing of the prototype, the Eclipse Workbench for Microsoft Windows was used, but there should not be any problem using the Eclipse Workbench for Linux.

The server side requirements are:

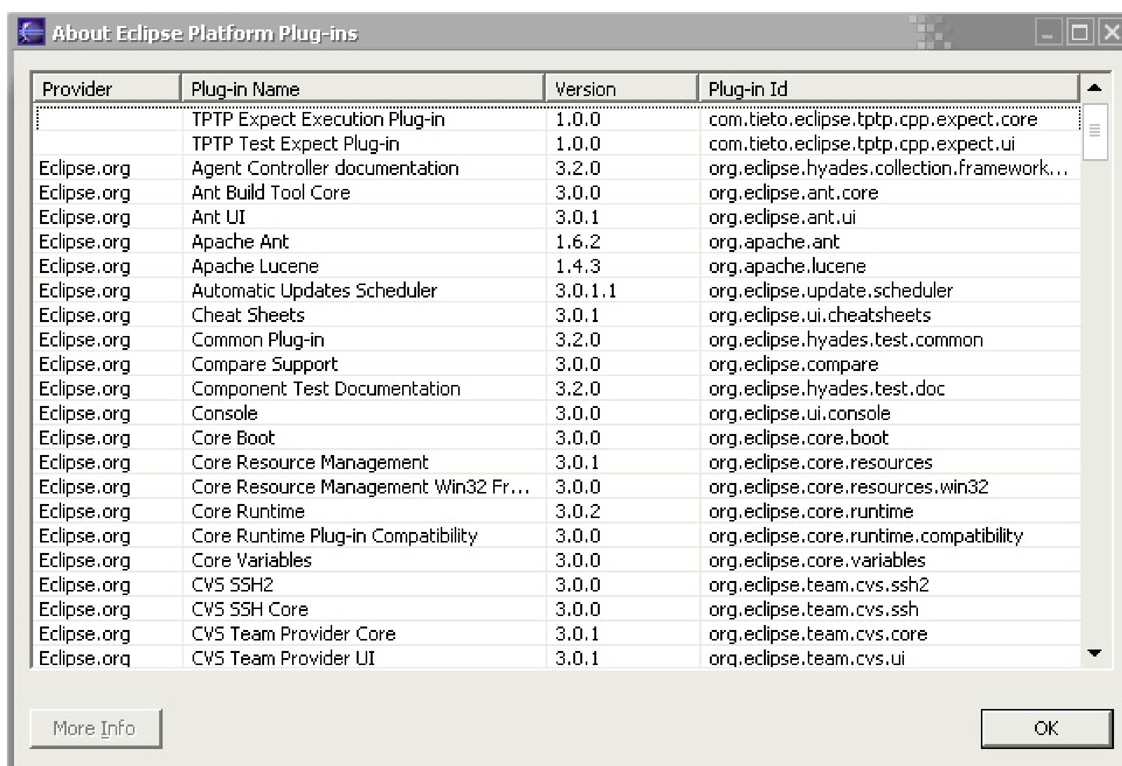
- Remote Agent Controller (shipped with Eclipse TPTP 3.2A), which is also known as Hyades Data Collection Engine
- CPP Emulator R2A or R2B (accessible via a valid ClearCase view)
- Expect

During the execution and testing of the prototype, the Linux and Solaris Sparc operating systems were used for running the Remote Agent Controller.

H.3 Installation of the Eclipse Plug-ins

To install the Eclipse plug-ins, simply unzip and copy the two folders found in the sub folder “client_eclipse_plugins” in the prototype distribution zip file into your “<Eclipse-root-dir>\plugins” folder. The plug-ins will automatically be detected during start up of Eclipse. To verify that the plug-ins have been correctly detected, do the following in Eclipse:

- In the “Help menu”, select “About Eclipse Platform”. The dialog window “About Eclipse Platform” will appear.
- Select the button “Plug-in Details”. Another dialog window “About Eclipse Platform Plug-ins” appears, see Figure 93.



Provider	Plug-in Name	Version	Plug-in Id
	TPTP Expect Execution Plug-in	1.0.0	com.tieto.eclipse.tptp.cpp.expect.core
	TPTP Test Expect Plug-in	1.0.0	com.tieto.eclipse.tptp.cpp.expect.ui
Eclipse.org	Agent Controller documentation	3.2.0	org.eclipse.hyades.collection.framework...
Eclipse.org	Ant Build Tool Core	3.0.0	org.eclipse.ant.core
Eclipse.org	Ant UI	3.0.1	org.eclipse.ant.ui
Eclipse.org	Apache Ant	1.6.2	org.apache.ant
Eclipse.org	Apache Lucene	1.4.3	org.apache.lucene
Eclipse.org	Automatic Updates Scheduler	3.0.1.1	org.eclipse.update.scheduler
Eclipse.org	Cheat Sheets	3.0.1	org.eclipse.ui.cheatsheets
Eclipse.org	Common Plug-in	3.2.0	org.eclipse.hyades.test.common
Eclipse.org	Compare Support	3.0.0	org.eclipse.compare
Eclipse.org	Component Test Documentation	3.2.0	org.eclipse.hyades.test.doc
Eclipse.org	Console	3.0.0	org.eclipse.ui.console
Eclipse.org	Core Boot	3.0.0	org.eclipse.core.boot
Eclipse.org	Core Resource Management	3.0.1	org.eclipse.core.resources
Eclipse.org	Core Resource Management Win32 Fr...	3.0.0	org.eclipse.core.resources.win32
Eclipse.org	Core Runtime	3.0.2	org.eclipse.core.runtime
Eclipse.org	Core Runtime Plug-in Compatibility	3.0.0	org.eclipse.core.runtime.compatibility
Eclipse.org	Core Variables	3.0.0	org.eclipse.core.variables
Eclipse.org	CVS SSH2	3.0.0	org.eclipse.team.cvs.ssh2
Eclipse.org	CVS SSH Core	3.0.0	org.eclipse.team.cvs.ssh
Eclipse.org	CVS Team Provider Core	3.0.1	org.eclipse.team.cvs.core
Eclipse.org	CVS Team Provider UI	3.0.1	org.eclipse.team.cvs.ui

Figure 93: The “About Eclipse Platform Plug-ins” Dialog Window

If the plug-ins have been successfully installed, “TPTP Expect Execution Plug-in” and “TPTP Test Expect Plug-in” should be listed as in Figure 93.

Another way to verify the installation of the plug-in is to confirm that the “TPTP Expect Test Suite” wizard can be found:

- In the “File” menu, select “New” and “Other…”. The “New” wizard dialog window appears.
- Verify that under the “Test” folder, the folder “Test Suite” contains the “TPTP Expect Test Suite” wizard, see Figure 94.

If any of these two steps can be confirmed, the installation of the Eclipse plug-ins is successful.

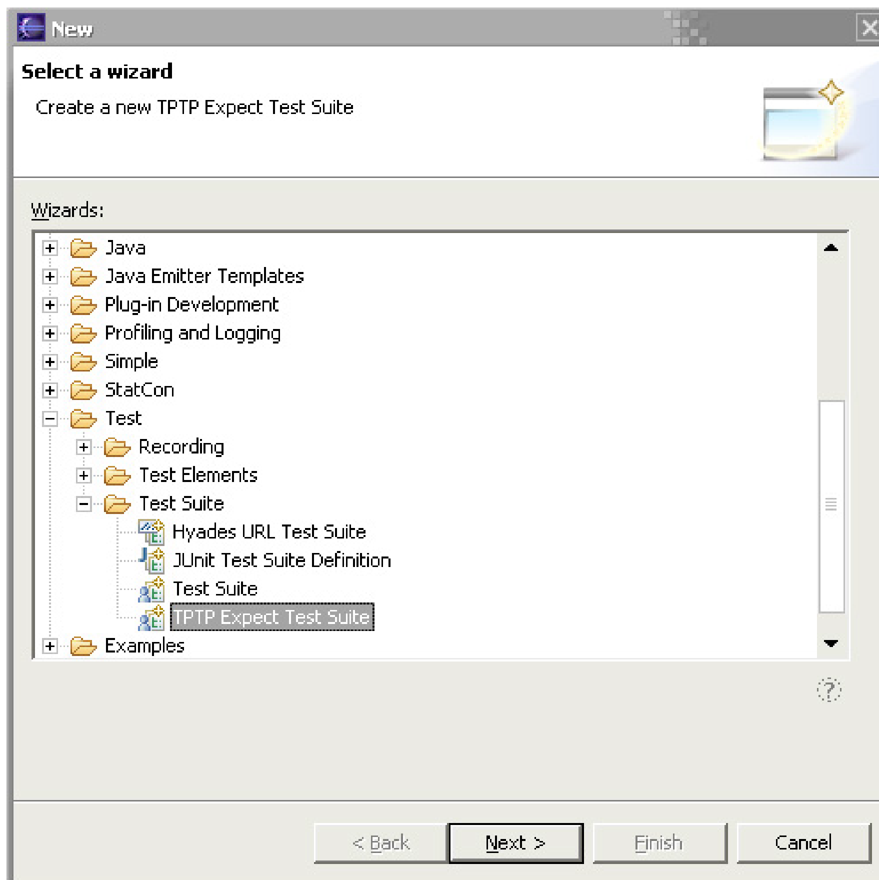


Figure 94: The “TPTP Expect Test Suite” Wizard

H.4 Installation of the RAC Plug-in

To install the RAC plug-in, simply unzip and copy the folder found in the sub folder "rac_plugin" in the prototype distribution zip file into your "<RAC-root-dir>/plugins" folder. The plug-in will automatically be detected during start up of the RAC. To verify that the plug-in has been correctly detected, do the following:

- Open the "servicelog.log" file, located in the "<RAC-root-dir>/config" folder, using a text editor.
- Confirm that sub-string "Successfully loaded plugin: com.tieto.eclipse.tptp.cpp.expect" can be found in the "servicelog.log" file.

The RAC plug-in also requires some environment variables to be set, in order to function correctly. The RAC needs to be started using a start script, "start.tcsh". The start script can be found in the sub folder "rac_start_stop_scripts" in the prototype distribution zip file along with another script file, "stop.tcsh". Use the "stop.tcsh" to terminate all processes started by the RAC. To use the scripts, simply unzip and copy the script files into your "<RAC-root-dir>/bin" folder. Before using the start script, it has to be edited:
Open the "start.tcsh" in an editor (NOTE: Edit the file using an editor in Linux/Unix or an Linux/Unix compatible editor under the Microsoft Windows Environment).
Change the "JAVA_HOME" variable to match the path where Java Runtime is installed.
Change the "RASERVER_HOME" variable to match the path where the RAC is installed.
Finally, change the "path" variable, so that the "Expect" executable is found within the path.