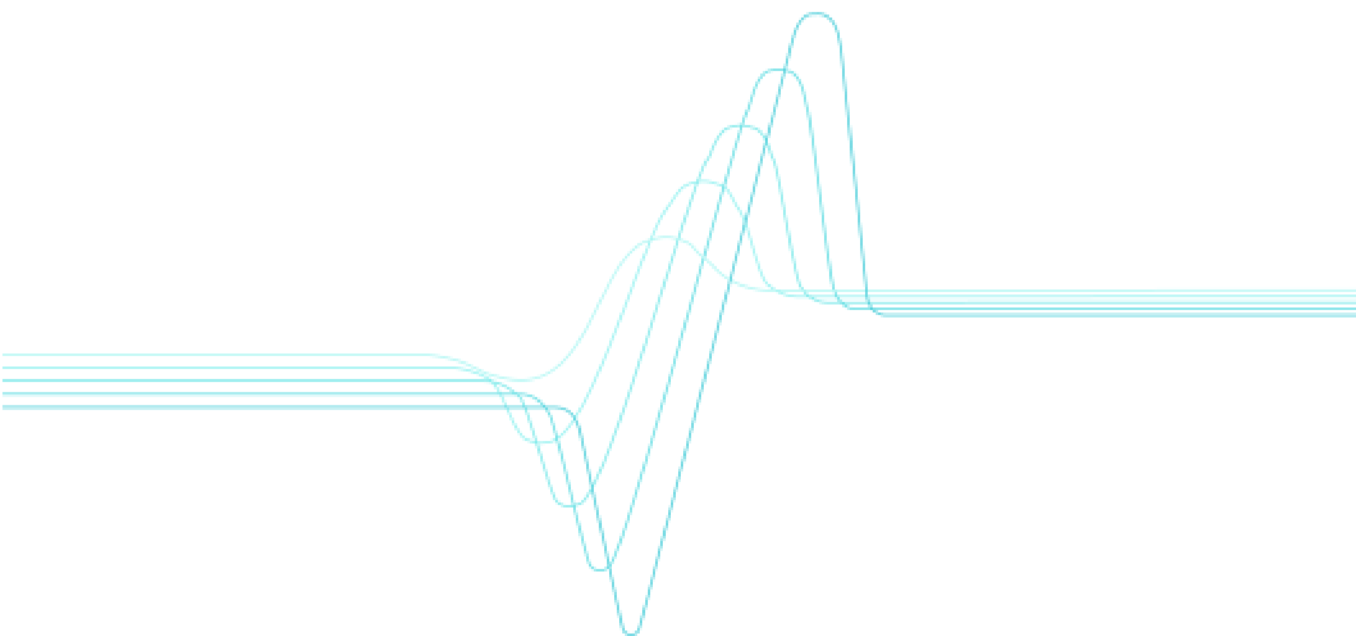
ESPOO 2004 VTT PUBLICATIONS 542

Pekka Pulkkinen

Mapping C++ Data Types into a Test Specification Language



VTT PUBLICATIONS 542

Mapping C++ Data Types into a Test Specification Language

Pekka Pulkkinen

VTT Electronics

ISBN 951–38–6402–2 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6403–0 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)

Copyright © VTT Technical Research Centre of Finland 2004

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Technical editing Marja Kettunen

Otamedia Oy, Espoo 2004

Pulkkinen, Pekka. Mapping C++ Data Types into a Test Specification Language [C++-tietotyyppien määrittely testienkuvauskielellä]. Espoo 2004. VTT Publications 542. 89 p. + app. 13 p.

Keywords Testing and Test Control Notation 3 (TTCN-3), software testing, software development

Abstract

Software testing is becoming a more and more important and challenging part of software development nowadays. Since the complexity and size of software is growing day by day, software developers must concentrate increasingly on testing, which costs both time and money. Therefore, different methods and tools have been developed to facilitate and precipitate software testing and also improve the quality of software.

One emerging new testing technology is TTCN-3 (Testing and Test Control Notation 3), which is a standardized test specification and implementation language. TTCN-3 provides a broad spectrum of testing abilities and is among others designed for testing software modules. It is also intended to be used for several applications with several data description languages. Even if C++ is one of the most popular programming languages nowadays, TTCN-3 cannot be yet efficiently utilize for testing C++ software. In order to take advantage of TTCN-3 in testing C++ modules, the interface of the tested component should be defined at the TTCN-3 language level. Therefore, C++ data types need to be mapped to TTCN-3.

The purpose of this thesis is to provide data type mappings from C++ to TTCN-3, and to implement a TTCN-3 based test system in order to test a C++ software module. Due to the differences between C++ and TTCN-3, such as lacking of object model in TTCN-3 and ambiguity of C++ pointers, several challenges are faced during this work. However, fairly comprehensive data type mapping is provided, which is finally verified in a real world-like situation by using TTCN-3 to test a C++ module. This example gives a clear insight of the usability and advantage of data type mappings and also valuable experience on the suitability of TTCN-3 in testing C++ software module is gained.

Pulkkinen, Pekka. Mapping C++ Data Types into a Test Specification Language [C++-tietotyyppien määrittely testienkuvauskielellä]. Espoo 2004. VTT Publications 542. 89 s. + liitt. 13 s.

Avainsanat Testing and Test Control Notation 3 (TTCN-3), software testing, software development

Tiivistelmä

Ohjelmistotestaus on yhä tärkeämpi ja haastavampi osa ohjelmistonkehitysprosessia. Ohjelmistojen koon ja kompleksisuuden kasvaessa testauksen merkitys korostuu. Tämän vuoksi ohjelmistotestauksen helpottamiseksi ja nopeuttamiseksi sekä ohjelmistojen laadun parantamiseksi onkin kehitelty erityisiä menetelmiä ja työkaluja.

Eräs testaukseen kehitetyistä uusista menetelmistä on TTCN-3 (Testing and Test Control Notation 3), joka on standardoitu testien kuvaus- ja toteutuskieli. TTCN-3 tarjoaa laajan valikoiman eri testausmenetelmiä ja sitä voidaan käyttää muun muassa ohjelmistomoduulien testaukseen. TTCN-3 on myös suunniteltu käytettäväksi yhdessä monien kuvauskielten kanssa erityyppisten sovellusten testaamisessa. Vaikka C++ on nykyään eräs suosituimmista ohjelmointikielistä, ei TTCN-3:a voida vielä tehokkaasti käyttää C++-ohjelmistojen testaamiseen. Käytettäessä TTCN-3:a C++-ohjelmistomoduulin testaukseen tulee testattavan komponentin rajapinta määritellä TTCN-3-kielellä. Tämän vuoksi tarvitaan määrittelysäännöt C++-tietotyyppien muuntamiseksi TTCN-3-kielelle.

Tässä diplomityössä määritellään C++-tietotyypit TTCN-3-kielellä sekä toteutetaan TTCN-3 testijärjestelmä C++ moduulin testaamiseksi. TTCN-3- ja C++-kielten välillä on suuria eroavaisuuksia, kuten olio-ohjelmointimallin puuttuminen TTCN-3:sta sekä C++-osoittimien moniselitteisyys, minkä vuoksi työn aikana kohdataan useita ongelmia. Tästä huolimatta työssä toteutetaan suhteellisen kattavat tyyppimäärittelyt, joita verifioidaan käyttämällä TTCN-3:a erään C++-moduulin testaukseen. Tämä esimerkki antaa selkeän kuvan tyyppimäärittelyjen käytettävyydestä ja hyödyllisyydestä. Lisäksi saadaan arvokasta kokemusta TTCN-3:n soveltuvuudesta C++-ohjelmistojen testauksessa.

Table of Contents

Abstract 3

Tiivistelmä 4

Table of Contents 5

Foreword 8

Acronyms and Abbreviations 9

1. Introduction 12

2. Software Testing 14

2.1. Testing Process 14
2.2. Testing Techniques 16
2.2.1. Functional Testing 17
2.2.2. Structural Testing 19
2.2.3. Static Testing 19
2.3. Types of Testing 20

3. Software Testing with TTCN-3 21

3.1. TTCN-3 Overview 21
3.1.1. TTCN-3 Test System 22
3.1.2. Basic Language Elements 24
3.1.3. TTCN-3 Control and Runtime Interfaces 26
3.1.4. TTCN-3 Presentation Formats 26
3.2. Implementing Tests with TTCN-3 27
3.3. Using TTCN-3 in Testing C++ Software Modules 31
3.3.1. Type Mappings 32
3.3.2. Runtime Behavior 33
3.4. TTCN-3 Related to Other Languages 34
3.4.1. ASN.1 34
3.4.2. IDL 35
3.4.3. XML 35

4. Mapping of C++ Fundamental Types to TTCN-3 37
4.1. Boolean Type 38
4.2. Characters 38
4.3. Integers 40
4.4. Floating Point Types 42
5. Mapping of C++ Compound Types to TTCN-3 43
5.1. User-defined Types 43
5.1.1. Class and Structure 43
5.1.2. Union 47
5.1.3. Enumerated Types 48
5.2. Pointers 49
5.2.1. Review of C++ Pointers 49
5.2.2. Pointer to Basic Types 51
5.2.3. Pointer to Class 55
5.2.4. Pointer to Pointer 56
5.2.5. Pointer to Other Types 57
5.3. References 58
5.4. Arrays 58
5.5. Type Definition 60
5.6. Templates 60
5.7. Conclusion 61
6. Case Study: Using TTCN-3 to Test a C++ Module 63
6.1. Testing Environment 63
6.2. Tested Module 66
6.3. TTCN-3 Test Software 68
6.3.1. Mappings for C++ Fundamental Types and Pointers 69
6.3.2. Mapping for the CFile Class 70
6.3.3. Other Data Types 71
6.3.4. Test Cases 72
6.4. Runtime Implementation 75
6.5. Test Runs and Results 76
7. Discussion 78
7.1. General Evaluation 78
7.1.1. Evaluation of Type Mappings 78

7.1.2. Evaluation of the Case Study ... 80
7.2. Problems and Solutions ... 81
7.3. Usability and Advantage of Type Mappings ... 82
7.4. Conclusion ... 83

8. Summary ... 84

References ... 86

APPENDIX 1 TTCN-3 Test Script: Mapping of C++ Basic Data Types and Pointers

APPENDIX 2 TTCN-3 Test Script: Mapping of the Interface of CFile Class

APPENDIX 3 TTCN-3 Test Script: Mapping of Other Necessary Types

APPENDIX 4 TTCN-3 Test Script: Implementation of the Test Cases

Foreword

The research work for this thesis has been carried out at VTT Technical Research Centre of Finland, in the group of Software Platforms during the fall of 2003 and spring 2004. The foundation of this research was achieved from the TT-Medal project (Tests & Testing Methodologies with Advanced Languages), which is a consortium of eight industrial partners and four research partners from Finland, Germany and Netherlands.

I would like to thank all of those people who have supported me during this work. Especially I wish to thank my supervisor Mr. Matti Kärki for the several great discussions, his guidance and support for this thesis. I would also like to thank my agreeable colleagues at VTT for many interesting discussions and useful working knowledge. Additionally, I appreciate the comments and proposals for improvement that I have received from my supervisor at the University of Oulu, Prof. Jukka Riekki, and the work's 2nd reviewer, Prof. Junzhao Sun.

Oulu, April 18, 2004

Pekka Pulkkinen

Acronyms and Abbreviations

API	Application Programming Interface, an interface that is used for accessing an application or a service from a program.
ASCII	American Standard Code for Information Interchange, the standardized set of 128 characters including letters, numbers, punctuation, and control codes.
ASN.1	Abstract Syntax Notation One, a language used for defining data types.
ATS	Abstract Test Suite, the test specification document.
CD	Coder/Decoder, that part of a TTCN-3 test system, which is responsible for the encoding and decoding of test data.
CH	Component Handler, an entity of TTCN-3 test system, which is responsible for distributing parallel test components.
CORBA	Common Object Request Broker Architecture, the OMG's standard that provides a set of common interfaces through which object-oriented software can communicate, regardless of computer platform.
ETS	Executable Test Suite, an implementation of the abstract test suite (ATS) that runs on a given test platform.
ETSI	European Telecommunications Standards Institute
GFT	The Graphical Presentation format of TTCN-3 behavior definitions.
HCI	Human-Computer Interaction, the study of how people work with the computers and how computers can be designed to help people effectively use them.
IDL	Interface Definition Language, a simple language for describing software interfaces.
ISO	International Standardization Organization

ITU	International Telecommunication Union
ITU-T	Telecommunication Standardization Sector of ITU
MFC	Microsoft Foundation Class library for Microsoft Visual C++.
MSB	Most Significant Bit, the binary digit in a binary number that represent the most significant value, typically the leftmost bit.
MSC	Message Sequence Chart, a graphical means for describing the behavior of systems runs (traces) within communication systems.
MSDN	The Microsoft Developer Network, a set of services for developers to help write applications using Microsoft products and technologies.
MTC	Main Test Component
MTS	Methods for Testing and Specification, a group of ETSI.
OMG	Object Management Group, a consortium of software vendors, developers, and users that promotes the use of object-oriented technology in software applications.
PA	Platform Adapter, an adapter in a TTCN-3 test system, in which external functions and timers are implemented.
PTC	Parallel Test Component
SA	System Adapter, an adapter in a TTCN-3 test system, which adapts message and procedure based communication to the particular execution platform.
SUT	System Under Test
TCI	TTCN-3 Control Interface, a standardized interface that specifies the interaction between Test Management (TM) and TTCN-3 Executable (TE) in a test system.

TE	TTCN-3 Executable, the part of a test system that deals with interpretation or execution of a TTCN-3 executable test suite (ETS).
TFT	The Tabular Presentation Format for TTCN-3, a graphical format that is similar in appearance and functionality to earlier versions of TTCN.
TM	Test Management, an entity that provides a user interface and administers the TTCN-3 test system.
TRI	TTCN-3 Runtime Interface, a standardized interface that defines the interaction of the TTCN-3 Executable (TE) with the SUT adapter (SA) and Platform Adapter (PA).
TTCN	Tree and Tabular Combined Notation, the first version of TTCN.
TTCN-2	Tree and Tabular Combined Notation second edition, former version of TTCN-3.
TTCN-3	Testing and Test Control Notation Version 3.
UML	Unified Modeling Language, a programming language that is used for object-oriented software development.
XML	Extensible Markup Language, a metalanguage for creating new markup languages. XML is a flexible way to create information formats, and share both data and meta-data with other applications and users.

1. Introduction

Software testing is becoming a more and more important and challenging part of software development nowadays. The complexity and size of software is growing day by day and thus more failure possibilities are expected and software becomes more unreliable. That is why software developers must concentrate increasingly on testing, which costs both time and money. Therefore, different methods and tools have been developed to facilitate and precipitate software testing and also improve the quality of software.

Software testing can be thought to be the process of uncovering evidence of defects in software systems [1]. A defect can occur in any phase of development and it is the result of bugs, misunderstandings, omissions and other this kind of mistakes. Testing is not tracking and fixing the bugs, but its purpose is substantially to ensure that a software application does what it is supposed to do, and does not do what it is not supposed to do.

TTCN-3 (Testing and Test Control Notation 3) is a standardized test specification and implementation language for all kind of black-box testing. TTCN-3 can be used in variety applications in the telecommunication area, for example protocol testing, service testing, module testing, testing of CORBA based platforms and testing of APIs. This language is very powerful and flexible and it supports a broad spectrum of testing types. In addition the possibility for different presentation formats and interfaces to different data description languages are provided. [2]

Since TTCN-3 is a test specification language, its main purpose is to implement test cases at an abstract level. That is, TTCN-3 test script concentrates on the purpose of test cases and the test system details are hidden. However, in order to be able to implement tests, tested component should be represented at the TTCN-3 language level. Thus, the interface of the tested component including the data types and the entry points must be mapped to TTCN-3. Furthermore, in order to fully implement tests the adaptation between the TTCN-3 test system and tested component is also required. This adaptation can be realized through standardized TTCN-3 execution interfaces. [3]

At this moment, C++ is a very popular language in software development. That is why there will be additional interest to use TTCN-3 in testing C++ based systems in the future. Even if TTCN-3 can be used for several applications with several data description languages, C++ is not directly supported to be used with it. TTCN-3 cannot be widely utilized for testing C++ software before C++ has been mapped to TTCN-3. However there are not even mapping rules between C++ and TTCN-3 to manually implement the conversion. [4]

TTCN-3 is an effective and flexible language and its syntax is similar to C++ or other high level programming languages but, however, there are considerabe differences in the concepts they are based on. One major difference is that C++ is an object-oriented language, whereas TTCN-3 is a pure procedural language. Another difference is that in C++ pointers play the major role, but TTCN-3 does not provide pointers at all. Due to these differences several challenges will be faced, when C++ features are being implemented at the TTCN-3 language level [4].

The purpose of this thesis is to gain experience in using TTCN-3 in testing C++ software. The main goal is to provide data type mapping from C++ to TTCN-3 and verify the mapping in practice by testing a C++ software module. The mapping of the following C++ data types will be dealt with:

- basic types (e.g. int, double, char),
- user-defined types (e.g. class, union),
- pointers,
- references,
- arrays, and
- templates.

Some of these types can be mapped to TTCN-3 pretty straightforwardly, but some of them are more complicated and perhaps can not be even solved at all. However, at least some kind of solutions will be provided.

As an introduction to the topic, Chapter 2 gives a general description of the software testing process and introduces common software testing techniques. Chapter 3 will provide an overview of TTCN-3 test system and give an insight of TTCN-3 relation to C++ as well as other languages, in which research has been made so far. Then the primary goal for this thesis work, mapping of C++ data types to TTCN-3, is divided into two groups in Chapters 4 and 5, which provide data type mappings for C++ fundamental types and compound types, respectively. These type mappings are verified in Chapter 6, in which a case study is implemented for using TTCN-3 to test a C++ software module. Chapter 7 gives a general discussion about the work performed in this thesis. Also some problems with solutions are presented, and advantages of the type mappings are discussed when TTCN-3 is used for testing C++ software. Finally, Chapter 8 gives a brief summary and presents the results that were achieved in this thesis.

2. Software Testing

Software testing has always been a challenging and laborious part of software development. Especially nowadays, when the complexity of software is growing increasingly and development times are getting shorter, testing is taking a more important role in the development process.

The two main reasons for testing are to make a judgment about the quality or acceptability of software and, the other hand, to discover problems [5]. These are, however, fairly extensive concepts; we do not necessarily know how to determine a good quality measure for software, or when the problems have been discovered.

This chapter gives a general insight into the software testing process and describes different software testing techniques. Also a brief summary of different types of testing is provided.

2.1. Testing Process

Testing is usually thought to be the last activity in the software development process after implementation. However, testing is such a type of activity that is performed all the time during the development phase and even after development, not just at the end of coding. Even if a testing process is separated from development process, they are intimately related to each other [1]. This can be observed in the software development life cycle, which is often described by the V-model shown in Figure 1. This model emphasizes the correspondence between software testing and development activities that are shown on the right side and on the left side of the model, respectively [5].

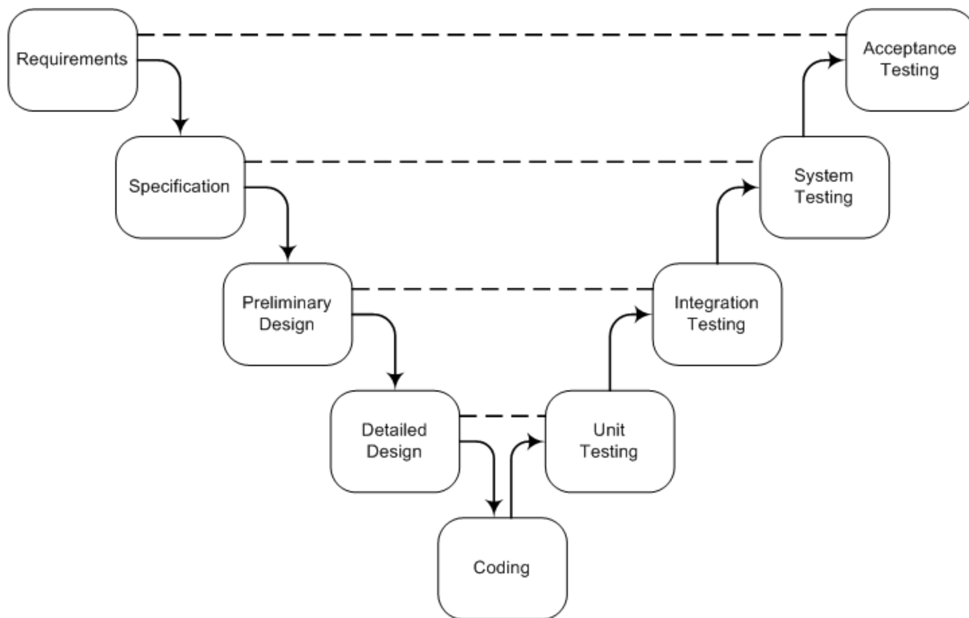


Figure 1. Abstraction levels of the software and testing development in the V-model.

The testing process has different goals and measures of success from the development process. That is, the testing process is for specifying if the product behaves as it is supposed to do, whereas the development process is for trying to build a product that meets the requirements. Normally these two processes are assigned to different people, which is very important from a system test perspective. The testers should write the test code independently from the developers, because different people may interpret the meaning of requirements differently. This ensures that the resulting system really corresponds with the requirements rather than that the system does what the developer has imagined the requirements to mean.

An interesting aspect of the testing process is how the occurred faults or errors are located. Sometimes failures can be caused by the test cases themselves or the drivers that execute them, not because of the tested software. In the real world, test software may actually contain more bugs than the software under test. That is why sometimes also the test case need to be tested.

The feedback loop in Figure 2, which often operates between the testing and development processes, will help to identify defective test cases [1]. The testing process feeds test results back to the development process. After revising the designs and implementations,

the development process feeds them to the testing process. By testing development products again a tester may determine that a failure occurred due to problems with the test system itself.

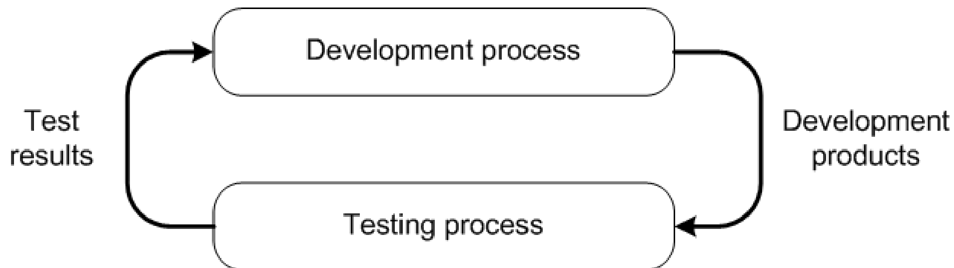


Figure 2. A feedback loop of the testing and development processes.

The testing process can be divided into three phases: design, construction and execution and evaluation. The test design part also includes several steps; first the system to be tested is examined and analyzed to identify its features and responsibilities. Then test cases are designed according to these results. Finally the expected results for each test case are developed. In the test construction part the artifacts that are needed for testing are constructed. For example, test cases are translated into programming language. The test execution and evaluation phase is normally the quickest part, and also the most visible part of the testing. In this phase earlier constructed test cases are executed and the results of each test case are evaluated as pass or fail. [1] [6]

These test steps can be performed at each level illustrated in Figure 1, that is, unit testing as well as integration, system and acceptance testing consist of design, construction, and execution and evaluation steps. Usually tests are designed and constructed on the left side of the V-model and the execution and evaluation are performed on the right side of the V-model. That is, test design and construction can be done before actual software has been written. [7]

2.2. Testing Techniques

Identifying test cases can be roughly divided into two groups; functional testing, which is also called specification-based or black box testing, and structural testing, which is also referred to as implementation-based or white box testing [1]. Both of these approaches include several distinct identification methods, such as robustness testing, path testing and data flow testing.

Another way to split testing techniques in half is to use the terms *static testing* and *dynamic testing*. Dynamic testing is the usual way of testing, that is, testing software by running and using it, whereas static testing is something that is not running, it is performed by examining and reviewing the code or specification.

2.2.1. Functional Testing

The software specifications are an essential part of functional testing, since the test cases are developed based on the specifications. Thus, it is not important how the actual software is implemented. Therefore, test cases can be implemented concurrently with, or even before the coding. [1]

Functional testing can be thought of as a mathematical function, in which the inputs are mapped to its outputs. There are several approaches to the functional testing [5]: boundary value analysis, robustness testing, worst case analysis, special value testing, equivalence class testing, and decision table based testing. The first four of these techniques are so called boundary-based approaches and they are the most common functional testing techniques. In the two other techniques different methods are used to find a comprehensive set of input value combinations that will verify the tested module as well as possible.

Boundary value analysis, as the name indicates, focuses on the boundary of the input space. Basically, when the test cases are being constructed, the input values are chosen in the way that they are at their minimum and just above the minimum, a normal value, and values at their maximum and just below the maximum. This model is based on the fact that the errors usually occur near the extreme values. [5]

The second model of boundary-based techniques is called robustness testing, which is actually a simple extension of the boundary value analysis. In robustness testing, in addition to the extreme values used in the boundary value analysis, also values outside the accepted range are used, that is the values slightly greater and slightly less than maximum and minimum, respectively. The most important aspect in the robustness testing is the expected output with the prohibited input. That is why the main attention is paid to exception handling. [5]

The boundary-based techniques described so far are used in the situations in which only the single fault assumption is made, that is, an extreme value is given only to a single

variable at a time. In the worst case testing what happens when several values have an extreme value is tested. This model needs much more effort than other boundary-based techniques, since a much greater number of test cases is needed for it, but then it is clearly more exhaustive. [5]

Special value testing is probably the most common, the most comfortable and sometimes very useful form of functional testing. This model is based on the tester's knowledge and earlier experience of weaknesses of the program while devising test cases. There are no guidelines for input values and thus it is very dependent on the tester's abilities. However, it often reveals faults more efficiently than other boundary-based techniques. [5]

The term "equivalence class" means a set of input values that produce the same result as another similar set. That is, if one test case that use the equivalence class works correctly, the remaining similar test cases can be assumed to work correctly as well, even if they are not tested. The idea of equivalence class testing is to use one element from each equivalence class when identifying test cases. This model is used when complete testing with small redundancy is wanted. [5]

Decision table based testing is useful in the situations in which varying sets of conditions produce numbers of combinations of actions. Hence, this technique uses tables, which show a set of conditions and actions resulting from them. These tables are then used to identify test cases. Decision table based testing is very useful for applications in which lots of decisions have to be made or where several logical relationships exist among input variables, but for some applications it is somewhat worthless. [5]

All of the functional testing techniques described above exhibit the tested program as a mathematical function that maps its inputs into outputs. When one is considering the technique that would be the best, it is important to pay attention to that part or those properties of the program that are most likely to be defective. However, these techniques can be thought to test only for a single function. There are also testing techniques that can be categorized into functional testing techniques, such as state based testing. These are not testing only for a single function but rather function interactions. State-based testing exercises an implementation for different test case and input sequences and the system under test (SUT) is modeled as a state machine. [6]

2.2.2. Structural Testing

In contrast to the functional testing, which is based on the specification of software, structural testing is based on software itself. When constructing the test cases in structural testing, input values can be determined by examining the source code of the tested system, and thus generate various execution paths by using different inputs. However, outputs of test cases must fulfill the requirements of the specification. The main advantage of structural testing compared to functional testing is that it improves the coverage, i.e. more comprehensive test results can be achieved. [1]

The two most common forms of structural testing are path testing and data flow testing. Path testing is based on the control flow of the tested program, from which test cases are generated, whereas data flow testing focuses on the links at which data objects are being used [8].

Path testing is the oldest technique in structural testing, and the most common basic test technique for programmers, but it should be common also for the testers. Several path testing techniques have been invented during past decades, such as DD path testing (decision-to-decision), basis path testing and branch coverage testing. DD path is a sequence of statements that begins from the decision statement and ends with another decision statement. In this technique, the test cases are constructed according to the DD paths of a program. When every DD path has been executed, we know that each predicted outcome has been examined. However, a simple program may have billions of different alternatives, thus comprehensive path testing is impossible. Normally testers do not construct paths by themselves, but they use a tool for it. [8] [5] [9]

Data flow testing includes the test strategies that select paths of control flow in order to construct and examine the sequences of events that are related to the status of a data object. The aim might be, for example, to ensure that all data objects, which have been initialized, are used somewhere. Data flow testing is often thought of as one form of path testing. [5] [8]

2.2.3. Static Testing

The process in which the source code, software architecture and design are reviewed without executing the program, is called structural analysis or static testing [9]. This type of testing is performed early in the development cycle and its purpose is to reveal bugs that are difficult to find with functional testing techniques. This kind of testing is sometimes pretty time-consuming and not productive enough and therefore it is not always performed. [9]

2.3. Types of Testing

The previous sub-chapter introduced several testing techniques which are normally used to test an abstract model of system capabilities and they do not require any specific implementation. However all capabilities of the system under test should be exercised when system testing is performed [6]. Over the course of time, several different types of testing have been developed. In most cases, these tests are performed after functional testing and they are usually closely tied to an implementation. The following list gives a brief summary of some implementation-specific types of testing [6].

- **Conformance testing** is for determining to what extent an implementation conforms the specification on which it is based. This is usually applied to a formal standard.
- **Interoperability testing**, to determine that two or more implementations are able to work with each other (i.e. interworking).
- **Configuration testing** is for identifying legal combinations of the environments for which a system under test fails.
- **Stress testing** is conducted to evaluate a system or component when the load is maximum or higher compared to its requirements.
- In **volume testing** an attempt to crash a system is tried by using the largest possible input (i.e. maximum amounts of data).
- **Performance testing** is conducted to evaluate a system's or component's actual performance against performance requirements.
- **Usability testing** evaluates the ergonomics of an HCI (Human-Computer Interaction) design.
- **Security testing** evaluates the ability to break into the security of the system by criminals or by accident.
- **Restart/recovery testing** evaluates automatic or manual recovery facilities of the system under test.
- **Maintainability testing** evaluates if the system will be maintainable.
- **Regression testing** retests a system or component to detect side effects of modifications.
- **Acceptance testing** determines whether or not a system satisfies its acceptance criteria, i.e. the system must fulfill its specification, be usable in practice and be reliable.

3. Software Testing with TTCN-3

3.1. TTCN-3 Overview

TTCN-3 (Testing and Test Control Notation 3) is a standardized test specification and implementation language for all kind of black-box testing. The TTCN-3 standard has been produced by the ETSI Technical Committee Methods for Testing and Specification (MTS) [10][11][12][13][14][15] and afterwards the International Telecommunication Union (ITU-T) has made some refinements to it and produced their own standards [16][17][18]. TTCN-3 can be used in a variety of applications in the telecommunication area, for example protocol testing, service testing, module testing, testing of CORBA based platforms and testing of APIs. This language is very powerful and flexible and it supports a broad spectrum of testing types. In addition, different presentation formats, and interfaces to different data description languages are provided. [2]

The previous version of TTCN was TTCN-2 (Tree and Tabular Combined Notation, Edition 2) which was restricted only to conformance testing. Even if TTCN-3 has evolved from TTCN-2, it is very different both syntactically and functionally and provides a much wider range of testing including interoperability, robustness, regression, system and integration testing. However, some basic functionality of TTCN-2 has been retained.

The following list shows some of the new features that TTCN-3 provides [16]:

- the ability to specify dynamic concurrent testing configurations,
- operations for procedure-based and message-based communication,
- the ability to specify encoding information and other attributes,
- the ability to specify data and signature templates with powerful matching mechanisms,
- type and value parameterization,
- the assignment and handling of test verdicts,
- test suite parameterization and test case selection mechanisms,
- combined use of TTCN-3 with ASN.1 and potentially with other languages, and
- well-defined syntax, interchange format, and static semantics.

3.1.1. TTCN-3 Test System

TTCN-3 test system is basically a set of interacting entities, in which each entity implements a particular functionality needed to construct the entire test system. A central entity of a test system is TTCN-3 executable (TE), which implements the execution of a TTCN-3 test suite. The TE interacts with the user and system under test (SUT) through the standardized interfaces, TTCN-3 runtime interface (TRI) and TTCN-3 control interface (TCI). These interfaces unify the way of realizing the TTCN-3 test system and provide adaptation for communication, management, component handling, external data and logging [2]. Overall view of a TTCN-3 test system is illustrated in Figure 3 [15].

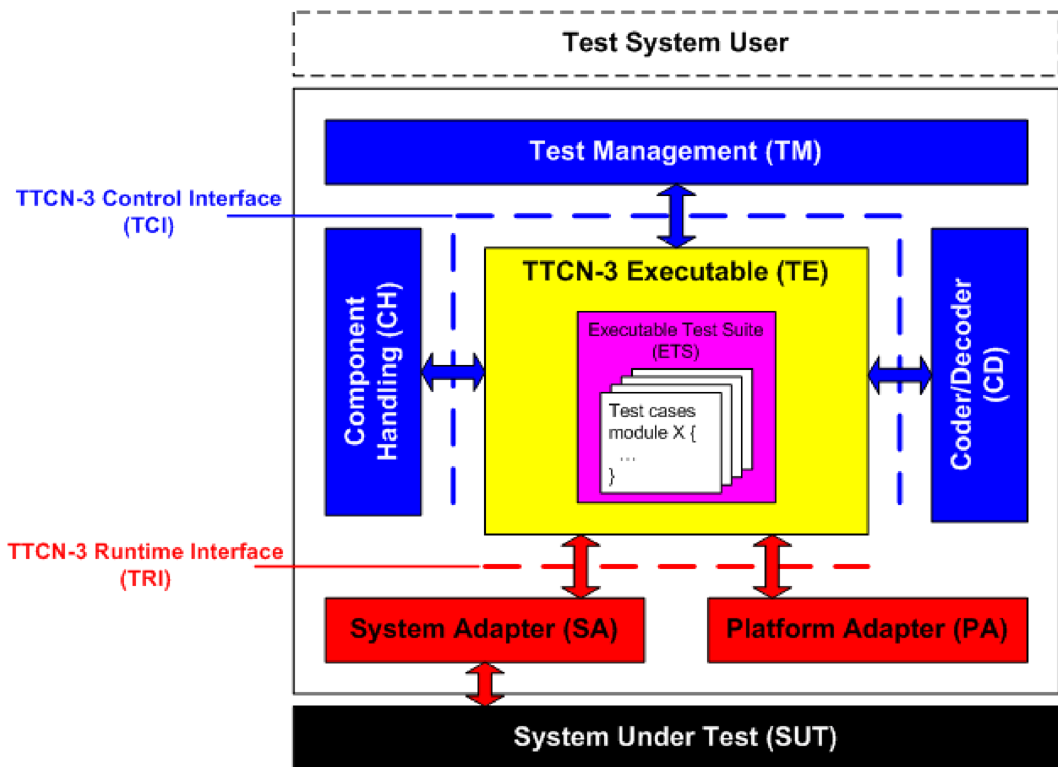


Figure 3. Overview of a TTCN-3 test system.

The TE is an abstract implementation of a TTCN-3 module and other entities make these abstract concepts concrete [2]. Test management (TM) is for overall management of the test execution, component handling (CH) is used to administer test components, and Coder/Decoder (CD) is for handling of types and values. The system adapter (SA) im-

plements the communication with the SUT and platform adapter (PA) realizes timers and external functions. [15]

The communication of the TTCN-3 test system is based on the interconnected test components and an explicit test system interface, which are connected to each other by well-defined communication ports. Every test system contains exactly one main test component (MTC), which is generated automatically at the beginning of each test case execution. In addition one or more parallel test components (PTCs) can be created dynamically during the execution. Each component has a set of communication ports. These ports are modeled as an infinite FIFO queue and they can have either in-, out-, or inout-direction. A conceptual view of a TTCN-3 test system communication is shown in Figure 4.

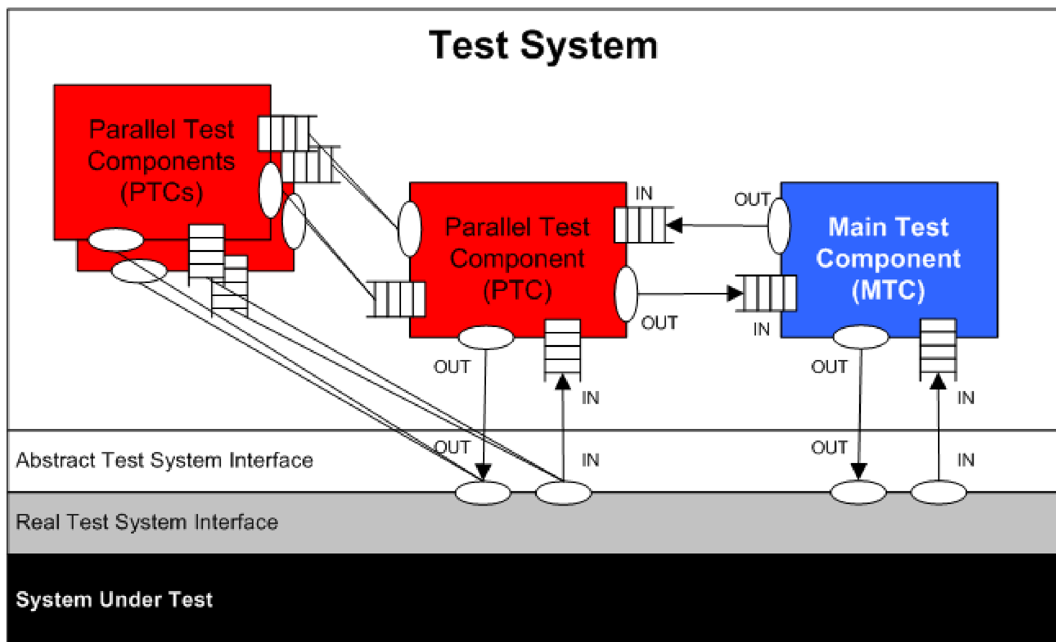


Figure 4. TTCN-3 test system communication. The elliptical circles describe the local ports and the in-directions of the ports are modeled as infinite FIFO queues.

3.1.2. Basic Language Elements

The top-level unit of TTCN-3 is a module which is an entity including all other elements. The module is usually divided into two parts: definition part and optional control part. Modules cannot be nested, i.e. a module cannot include other modules, but they can import definitions from other modules. Figure 5 gives a general overview of TTCN-3 language elements.

The module definitions part consists of data type definitions, test data, test system architecture, and test system behavior. Data type definitions are either simple predefined types (i.e. integer, float, charstring) or user defined types, such as arrays or other structured types (i.e. record, set).

Test data is fed to or received from the tested function or module. It can be constructed from constants, variables or message templates. Templates are used to either transmit a set of distinct values or to test whether a set of received values matches the template specification. Templates provide several possibilities, such as organize and reuse test data including a simple form of inheritance.

The architecture definitions describe messages, signatures, test components, and communication ports. The test components define the interface to the system under test, and they communicate through the communication ports with the system and other test components. Procedure signatures and messages are the entry points of the test components.
Test behavior definitions describe the modular structure of the tests by taking advantage of altsteps, functions and test cases. A TTCN-3 alt statement, which consists of several altsteps, provides a specific feature of the TTCN-3 semantics. Altstep defines an ordered set of alternatives, and it is a scope unit similar to function. They are used to specify default behavior or to structure the alternatives of the tests.

The module definitions part specifies the top level definitions of the module and these definitions are global in the entire module. The module control part can be thought as the main program of the TTCN-3 module. Its main purpose is to execute test cases in a certain order, but the local declarations, such as timers and variables, may be defined in the control part, as well. [10]

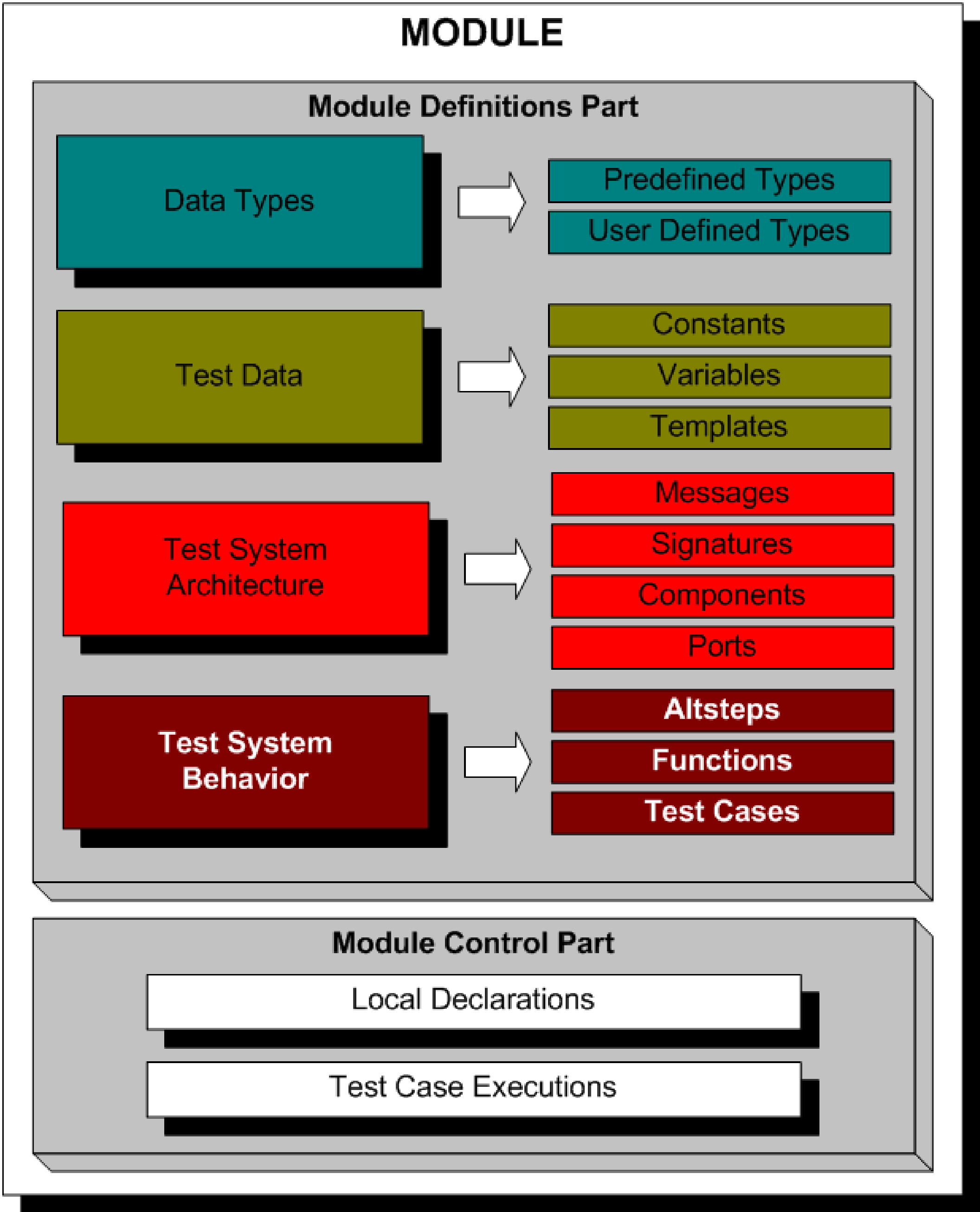


Figure 5. TTCN-3 language elements.

3.1.3. TTCN-3 Control and Runtime Interfaces

As mentioned earlier, standardized adaptation of the test system is defined by the TTCN-3 execution interfaces, TRI (runtime interface) [14] and TCI (control interface) [15]. The TCI define entities, interfaces, types and operations, which prepare the way for flexible management of TTCN-3 based systems. The TCI complements and completes the TRI, which provides a test and platform specific adaptation layer. These well-defined interfaces include sets of operations, and are independent of the implementation language, SUT (system under test) and processing platform. That is, the code produced by any TTCN-3 compiler or interpreter and any platform or device are compatible if they all support these interfaces. [19] [20]

The TRI consists of two parts, TRI SA (SUT Adapter) and TRI PA (Platform Adapter), as it was illustrated in Figure 3. The TRI SA realizes message and procedure based communication between the SUT and TTCN-3 test system. It is responsible for the mapping of communication ports to test system interface ports and transmits requests and responses through these ports. The TRI PA realizes the external functions and timers. That is, it enables external function invocations and timer operations, i.e. starting, reading, stopping and inquiring the status of timer. [2]

3.1.4. TTCN-3 Presentation Formats

In addition to fundamental textual format, TTCN-3 provides the tabular presentation format (TFT) and graphical presentation format (GFT), standardized by [11] and [12], respectively. TTCN-3 core language is represented with normal text-based syntax and it can be used independently, but TFT and GFT are based on the core language. Different alternatives to display the core language and the interfaces to different languages provided are shown in Figure 6. [2]

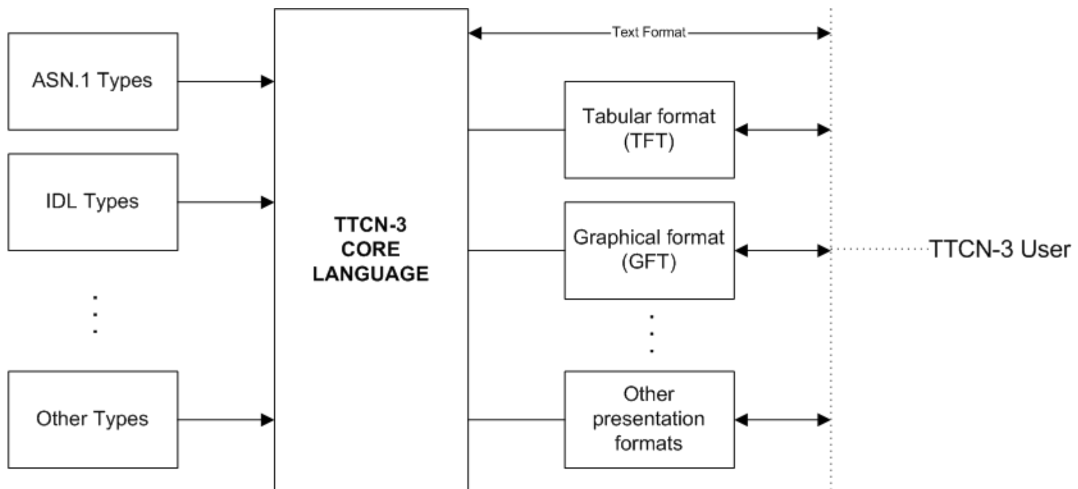


Figure 6. Different presentation formats and types provided by TTCN-3.

A TTCN-3 textual core notation can be written using any text editor and the syntax looks similar to a typical programming language, such as C++ or Java. The tabular presentation format (TFT) is similar in appearance and functionality of earlier versions of TTCN and is aimed at the developers that are familiar with the TTCN style of writing test suites. Basically, the TFT is collection of tables that represent a TTCN-3 module. The graphical presentation format (GFT) provides the visualization of test behavior by taking advantage of the message sequence charts (MSC). The GFT provides graphical presentation for each TTCN-3 behavior description, that is, diagrams for the following behaviors are provided: the control part of a TTCN-3 module, TTCN-3 test case, TTCN-3 function, and TTCN-3 altstep (i.e. alt statement). [11] [12] [2]

3.2. Implementing Tests with TTCN-3

When tests are being implemented using TTCN-3, the implementation process can be divided into the following steps:

- defining data types and procedure signatures,
- defining test data,
- defining ports and components,
- defining test behavior, and
- execution of test cases.

These are the steps that can be clearly distinguished from each other, but however, the order is not always the same. That is, for example, ports and component can be defined at the beginning of a module or test data can be constructed in parallel with test behavior.

The first step, i.e. definitions of the data types and procedure signatures, may sometimes be very challenging, since the data types of the system under test are to be defined at the TTCN-3 language level in order to implement rest definitions (i.e. procedure signatures, test data, test behavior, etc.). This is because the data types of the SUT may differ considerably from TTCN-3 types. Therefore the mapping from the target language to TTCN-3 must be available. Consequently, this thesis concentrates on specifying data type mappings from C++ to TTCN-3.

TTCN-3 provides two different mechanisms for information exchange between the test components and system under test (SUT); message-based communication is related to data types and procedure-based communication is related to procedure signatures [2]. TTCN-3 does not have any special message type, but a message can be any value of a certain TTCN-3 type. A procedure signature consists of the following fields: its name, a list of parameters, a return value, and a list of exceptions. Parameters, return value and exceptions are optional. An example of the procedure signature:

```
signature MySignature (in integer myPar1, inout float myPar2)
    return boolean exception (reasonType);
```

Test data, as mentioned earlier, are constructed from constants, variables and templates. The templates mechanism provides a very comfortable way to specify, organize and structure test data [2]. In fact, test data is sometimes constructed within the test cases, i.e. in a test behavior step. However, once test data has been produced, it can be fed to the SUT as an input or received from the SUT as an output. A signature template *myData* for the earlier defined signature *MySignature()* could be as follows:

```
template MySignature myData := {
    myPar1 := 100,
    myPar2 := 1.23
}
```

Components and ports are specified by creating a port type or component type, respectively. A port can be message-based, procedure-based or both (i.e. mixed) depending on

whether it is used for message exchange or with the procedure calls. In addition ports can have three different directions (*in*, *out* and *inout*) for allowed messages or procedures. A port type definition for *MySignature()* would be for example:

```
type port MySignaturePort procedure {
    out MySignature
}
```

In addition to port type definition, also component type definitions must be specified, which can include constants, variables, timers and ports. Each test component is an instance of a corresponding component type. An example of the component type definition, which contains *MySignaturePort* could be:

```
type component MyTester {
    port MySignaturePort port1
}
```

Because the abstract test system interface can be thought of as a test component, i.e. a collection of ports, another component type definition is needed. Thus, the following component type definition is for the abstract test system interface:

```
type component MyTestSystemInterface {
    port MySignaturePort SUT_Interface
}
```

The abstract test system interface and MTC (Main Test Component) are created automatically, when the execution of a test case starts and they can be referenced by predefined keywords *system* and *mtc*, respectively.

Test system behavior consists of altsteps, functions and test cases, as depicted earlier in Figure 5. The main entities in the definition of test system behavior are test cases, which are usually for the description of MTC's behavior. Altsteps and functions are used to modularize of the test case structure, for example by collecting common definitions or describing the behavior of PTCs (Parallel Test Components) in the functions and altsteps, which then can be called within a test case. An example of test case definition, which takes advantage of an altstep and a function, is shown in Figure 7 below.

```
// functions and altsteps used within the test cases
function InitTestCase() runs on MyTester {
    map(mtc:port1, system:SUT_Interface);
    activate(DefaultAltSet());
}
altstep DefaultAltSet() runs on MyTester {
    [] any port.getreply {
        // . . .
    }
    [] any port.catch {
        // . . .
    }
}


// test case implementation
testcase tc_01() runs on MyTester system MyTestSystemInterface {
    InitTestCase();
    // call the function under test
    port1.call(MySignature:myData, nowait);
    alt {
        // check the expected result
        [] port1.getreply(MySignature:{-, 1.0} value true) {
            if(/*...*/) { . . . }
        }
    }
    // . . .
    setverdict(pass);
    unmap(mtc:port1, system:SUT_Interface);
    stop;
}
```

Figure 7. An example of function, altstep and test case definitions.

All the definitions described so far are defined in the module definitions part. Finally, the execution, i.e. main program, is implemented in the module control part, in which among others the test cases are executed. An example of the control part:

```
control {
    var verdicttype myVerdict := none;
    if (aVerdict == none) {
        aVerdict := execute(tc_01());
    }
    if(myVerdict == pass) {
        // . . .
    }
}
```

After the entire test script has been completed, the tests are executed as follows: The execution starts from the module control part, in which test cases are executed in a certain order, one after the other. Functions and alt statements specified in the test cases are performed in the same way as in any other procedural programming language. This causes the function of the SUT to be called through the corresponding communication port. During this call TTCN-3 data types (i.e. parameters) are transferred into the SUT's data types. This conversion is performed by the CD (Coder/Decoder) that is one entity of a standardized TCI (i.e. TTCN-3 Control Interface) as described earlier in section 3.1.1. The return values are correspondingly decoded from the target language back into TTCN-3. The implementation of the encoder and decoder as well as signature calls and external functions must be done using a target language, for example C++ or Java. Finally, as a result of test a case execution, a test verdict (i.e. pass, fail, inconc, none, error) is returned.

3.3. Using TTCN-3 in Testing C++ Software Modules

At this moment, C++ is a very popular language for software development, which certainly will attract additionally interests to use TTCN-3 in testing C++ software. TTCN-3 is designed for dynamic testing, so it can be used for various different types of testing at different levels. Hence, TTCN-3 can be also used to test software modules. [10]

The role of TTCN-3 in testing software modules is to define test procedures and specification of test suites. That is, test software written with TTCN-3 is attached to tested software through the interface of the system under test (SUT). However, TTCN-3 is designed for the test case implementation at an abstract level and the test system details are hidden. Thus, TTCN-3 test system needs the adaptation to the SUT in order to fully implement tests. [3]

Practical experience has proven that using TTCN-3 for testing C++ software is laborious. This is caused by several issues: on the one hand, test data is difficult to define within the TTCN-3 modules, and on the other hand, in order to implement an executable system the run time behavior presents additional requirements. However, these issues are not insuperable, but currently excessive effort is needed in order to implement test cases.

3.3.1. Type Mappings

A typical test system architecture for software testing contains the entities that are used to call the functions of the SUT with certain parameters, i.e. test driver, and to simulate the software components that are not yet implemented, i.e. test stubs [6]. In order to call or simulate these functions or components within a TTCN-3 module, the interfaces of the SUT must be implemented at the TTCN-3 language level. In the case of C++, the interfaces are defined in the header files, which contain data type definitions, data declarations and data definitions, and also function declarations. Thus, these elements should be mapped from C++ to TTCN-3. Figure 8 shows an example of how the mapping may look. The details presented in this figure will be explained later in chapters 4 and 5.

Even if TTCN-3 is an effective and flexible language and its syntax is similar to C++, there are significant differences between these languages. These differences are mainly due to the lacking of an object model in TTCN-3, but also other features, such as pointers, are outside the scope of TTCN-3. During the mapping process depicted in Figure 8, the differences between these languages become significant.

Currently there are no specific rules how C++ types and their properties can be implemented by TTCN-3. This thesis concentrates on this issue by providing data type mappings from C++ to TTCN-3. Chapter 4 provides mappings for C++ fundamental types and Chapter 5 for C++ compound types. Mapping of member functions are not included in this work, since this provides mapping only for the C++ data types, not entire C++ language mapping. However, a case study in Chapter 6 provides a mapping for the entire interface of a C++ module including member functions.

```
class CFile : public CObject {
public:
// Flag values
    enum OpenFlags {
        modeRead = 0x0000,
        modeWrite = 0x0001,
    };
// Constructors
    CFile();
    CFile(int hFile);
// Operations
    virtual UINT Read(void* lpBuf, UINT nCount);
    static BOOL PASCAL GetStatus (
        LPCTSTR lpszFileName, CFileStatus& rStatus);
    // ...
// destructor
    virtual ~CFile();

private:
    UINT m_hFile;
    BOOL m_bCloseOnDelete;
    CString m_strFileName;
};
```

```
module T3_CFile {
    import from T3_CObject all;

// type definitions for the components and ports
    type component CFileTSI {
        port CFilePort aCFilePort
    }
    type port CFilePort procedure {
        out CFile_no_parameters,
        CFile_one_parameter,
        Read,
        GetStatus,
        D_Cfile
    }

// Flag values
    const CppInt modeRead := hex2int('0000'H);
    const CppInt modeWrite := hex2int('0001'H);
// Constructors
    signature CFile_no_parameters() return CppPtr;
    signature CFile_one_parameter(in CppInt hFile)
        return CppPtr;
// Operations
    signature Read(in CppPtr this, in CppPtr lpBuf,
        in UINT nCount) return UINT;
    signature GetStatus(in LPCTSTR lpszFileName,
        in CppPtr rStatus) return BOOL;
// destructor
    signature D_CFile(in CppPtr this);

    type record CFileType {
        UINT m_hFile optional,
        BOOL m_bCloseOnDelete optional,
        CStringType m_strFileName optional
    }
} // end of module T3_CFile
```

Figure 8. An example of mapping entire C++ software interface to TTCN-3.

3.3.2. Runtime Behavior

In order to implement an executable test system, a TTCN-3 abstract test suite (ATS) must be compiled or interpreted to a target language, and also runtime behavior must be implemented [21]. Thus, in our case, we want to translate TTCN-3 code into C++ and implement the runtime behavior with C++ as well. Roughly speaking, runtime behavior consists of the following operations:

- encode parameters (in/inout directions) from TTCN-3 into the target language,
- decode parameters (out/inout directions) and return values from the target language into TTCN-3,
- implement the procedure-based communication (i.e. call, getcall, reply, getreply),
- implement message-based communication (i.e. send and receive operations),

- handle exception, and
- implement the external functions.

For the moment, these operations must be implemented by hand, which considerably increases the effort for implementing an executable test system. This means in practice that lots of code lines are needed even for a simple test case. Especially encoding and decoding require quite much work, and if these codecs could be generated automatically, testing would be considerably facilitated.

3.4. TTCN-3 Related to Other Languages

In addition to using TTCN-3's own notation, ASN.1 (Abstract Syntax Notation One) is the only language that is supported by the standard to be used with TTCN-3. However, research has been extensively made for the extensions of the OMG IDL [22] [23] [24] and XML [25]. Hence, according to these studies, ETSI has lately published a technical specification of IDL to TTCN-3 mapping [26] and they have XML to TTCN-3 mapping under work.

Because TTCN-3 was designed to be used in a broad spectrum of testing types, these kinds of mapping are extremely useful. Roughly speaking, ASN.1 data definitions are used for protocol stacks, IDL definitions for CORBA based systems and XML definitions for Web services. The following sub-chapters give a quick view of the relation of TTCN-3 to these three languages.

3.4.1. ASN.1

ASN.1 is standardized data representation format produced by the International Standards Organization (ISO). It is used for the definition of simple data types, and also a notation for referencing these types and for specifying the values of these types. Its main purpose is to achieve interoperability between platforms. [27]

TTCN-3 standard supports using ASN.1 together with TTCN-3. Therefore it is possible to reference to ASN.1 definitions from within a TTCN-3 module. Also specifying the encoding rules for imported ASN.1 definitions is allowed. Simply speaking, TTCN-3 allows importing and using the ASN.1 objects within a TTCN-3 module. [2]

However, some difficulties exist when ASN.1 is used with TTCN-3. The ASN.1 objects, which are imported into a TTCN-3 module, are called *foreign objects* and they can be used only if they have *a TTCN-3 view*. This means that part of the information of an object which is needed when the object is used in TTCN-3. This TTCN-3 view can be full, zero or something between them. Because ASN.1 has richer types and it provides a wider range of type construction mechanisms, the use of ASN.1 definitions within a TTCN-3 module is more complex than just importing them. More speculation about these problems is provided in [2].

3.4.2. IDL

The IDL (Interface Definition Language) is mostly used to describe the object interfaces for CORBA (Common Object Request Broker Architecture) based systems. It supports the most basic data types like C++ or other high level languages. CORBA is a middleware for Internet based distributed systems, which uses an object-oriented concept. It is standardized by the Object Management Group (OMG).

A mapping of OMG IDL to TTCN-3 has been studied in several papers [22] [23] [24]. Based on these studies, ETSI has produced a technical specification that provides the mapping of CORBA IDL specifications into TTCN-3 [26]. These mapping rules are also intended for other interface specification languages, not only for CORBA IDL. The purpose of IDL mapping is to provide a definition for the use the TTCN-3 core language with IDL [26]. This can be also observed from Figure 6. Chapters 4 and 5 in this thesis, which provide data type mapping from C++ to TTCN-3, take an advantage of IDL to TTCN-3 mapping documents [22] [23] [24] [26]. However, C++ and IDL differ considerably from each other, and thus IDL mapping documents cannot be fully utilized, even though they can be used as a guideline during the C++ to TTCN-3 mapping process.

3.4.3. XML

The Extensible Markup Language (XML), which is derived from the Standard Generalized Markup Language (SGML), is a simple and flexible text format and it is independent of platform, software and hardware [28]. XML is currently widely used, among others, on the Web for exchanging a wide variety of data. The purpose of XML is for marking up data in the documents. That is, special portions of text are wrapped in the tags that makes it easier to manipulate a document. XML has been used for example, for

making the configuration files for software, or for telecommunication applications for transferring control or application data. The main advantage of XML is its flexibility in representing structural information.

For instance, in [25] TTCN-3 is used for providing a test framework for Web services which are developed in the form of XML schemas. The key element of this test framework is to translate XML data to TTCN-3. That is, when test data structures are being generated, the mapping of XML to TTCN-3 is required. Some kind of a mapping is provided in [25], and according to "The TTCN-3 User Conference" that will be held in May 2004 [29], the mapping of XML to TTCN-3 is being currently worked with.

4. Mapping of C++ Fundamental Types to TTCN-3

This and the following sections present the work performed in this thesis. The following two chapters (Chapters 4 and 5) contain data type mappings from C++ to TTCN-3 and then Chapter 6 provides a case study, in which these mappings are utilized. As described earlier in sections 3.2 and 3.3, the first step in test implementation with TTCN-3 is to define the data types that are used, for example, while constructing the test data used in the test cases. Since here TTCN-3 is going to be used in testing C++ software, the interfaces of tested module must be implemented at the TTCN-3 language level. Therefore, the C++ to TTCN-3 type mappings provided here are required.

C++ types are divided into two groups in the C++ standard [30]: fundamental types and compound types. Compound types are constructed from the fundamental types. Thus, the same approach for dividing data type mapping in two parts has been also used in this work: this chapter describes the mapping of fundamental types and mapping of compound types is described in Chapter 5.

Fundamental types in C++ consist of integral types, floating point types and the *void* type. Integral types contain character types (plain, signed and unsigned), integer types (signed and unsigned), wide character (i.e. *wchar_t*) and boolean type (i.e. *bool*). Floating point types include *float*, *double* and *long double*. The *void* type has an empty set of values; i.e. it is "an incomplete type that cannot be completed" [30].

In contrast, there can be infinite number of types in TTCN-3. This is, because it is possible to limitlessly define new types by using the *type* command. However, TTCN-3 supports only a few basic types, which are mostly same as those in C++, but there are also some TTCN-3 specific ones [16]. There are quite big differences between the basic types of these languages and this is especially emphasized when they are used in practice.

Some aspects of C++ types are implementation-defined [31]. For example the size of *int* depends on the system that is being used. That is why, in order to provide comprehensive type mapping, the hardware characteristic of the used system must be known. In this work, Visual C++ Version 6.0 was used and, for example, sizes of variables are based on the characteristics of this system.

The representation of integral types in C++ is based on a pure binary numeration system, for example two's complement, one's complement, or signed magnitude representation

[30]. On the contrary, TTCN-3 uses a different numeration system in which types must be compatible. That is, at assignments, instantiations and comparisons the values must have the same root type and they must not violate subtyping [10]. For example, C++ accepts a negative value to be placed in an unsigned integer, but since the MSB (Most Significant Bit) is set, this value is interpreted as large positive number. In addition, C++ provides standard conversions (e.g. *int* to *double*), but they are not provided by TTCN-3. This kind of problem can be however solved only by making the value conversions by hand.

4.1. Boolean Type

A boolean type is almost the same in both C++ and TTCN-3. However, in C++ true is handled as an integer number 1 and false as a zero, whereas TTCN-3 accepts only values *true* and *false*. C++ accepts both true and false and also any integer value, in which all values except 0 are truncated to 1 (corresponding *true*) [31] [16]. Table 1 shows a type mapping for the boolean type and some examples of boolean expressions. Thus, when an interface of a C++ component contains a boolean type, the corresponding representation in TTCN-3 can be seen from Table 1. In order to enhance readability and to provide clear distinction, all mapped types get the prefix Cpp (i.e. abbreviation of C++).

Table 1. Mapping of boolean type from C++ to TTCN-3.

C++	TTCN-3
`bool`	`type boolean CppBool;`

`bool is_true = true;`
`bool is_true = 10;`
`bool is_false = 0;`
	`var CppBool is_true := true;`
	`var CppBool is_true := true;`
	`var CppBool is_false := false;`

4.2. Characters

TTCN-3 characters are based on ISO/IEC 646 [32], which uses a 7-bit coded character set, whereas C++ uses in most cases 8-bit characters, which is typically a variant of ISO-646 [31]. An example of 8-bit character set is standard ECMA-128 [33], which contains among others the Scandinavian characters ‘ä’ and ‘ö’. However, the set of characters in

C++ is only partially standardized, which might cause problems if the same application is used in different environments.

C++ characters consist of three different types: *char, signed char* and *unsigned char.* Plain and signed char can hold values from –128 to 127, whereas unsigned char can hold values between 0 and 255. TTCN-3 accepts only characters corresponding to the values 0 to 127. In conclusion, if the basic ASCII character set is used, char in TTCN-3 is the same as char in C++. [31] [16]

In addition, C++ includes a type *wchar_t*, which is a distinct implementation-defined type and is large enough to hold the largest extended character set specified among the supported locales. The nearest corresponding type in TTCN-3 is *universal char*, which is a 32-bit value based on ISO/IEC 10646, which altogether contains 2^{32} characters including all kinds of character sets, such as kanji, katakana, hiragana, romaji, etc. *universal char* can also be used when normal chars outside the ASCII range are used. A mapping of character is provided in the Table 2 below. [31] [16]

Table 2. Mapping of characters from C++ to TTCN-3.

C++	TTCN-3
`char`	`type char CppChar;`
`signed char`	`type char CppSignedChar;`
`unsigned char`	`type char CppUnsignedChar;`
`wchar_t`	`type universal char CppWchar_t;`

C++ accepts also integer values to be put into *char*, which is commonly used for example in a function call. This is not allowed in TTCN-3. However, TTCN-3 standard [10] provides predefined functions, such as *int2char, int2unichar, char2int* and *unichar2int*, which can be used if an integer value is used in the place of character or vice versa. These functions make the conversions based on ISO/IEC 646. There are also some special characters, such as horizontal tab (‘\t’) and new line (‘\n’) which cannot be directly mapped to TTCN-3, but by using the conversion functions they can also be specified.

However, the earlier provided mapping is not perfect. The problems arise, for example, when a negative value is placed into a *char* or values outside the ASCII range are used.

This is solved by providing distinct mapping, in which integers are used instead of char. This alternative way to map C++ char type is provided in Table 3. Since characters in C++ are actually integer types [30], the mapping in Table 3 works in every situation. However, this mapping is not very descriptive and thus the mapping in Table 2 is recommended for use, when possible.

Table 3. Alternative mapping for characters from C++ to TTCN-3.

C++	TTCN-3
`char`	`type integer CppChar(0..255);`
`signed char`	`type CppChar CppSignedChar;`
`unsigned char`	`type integer CppUnsignedChar(-128..127);`
`whcar_t`	`type integer CppWchar_t(0..65535);`

4.3. Integers

There are four distinct integer types, which are *signed char*, *short int*, (plain) *int* and *long int,* in ascending order by their size, respectively. In addition, there exists a corresponding unsigned integer type for each signed type, and unsigned types occupy the same amount of storage as the corresponding signed types. [30].

The sizes of integer types are implementation-defined, but all of them are a multiple of the size of *char* which is normally 8 bits [31]. In contrast, TTCN-3 accepts all possible integer values, basically from negative infinity to positive infinity. Thus, when mapping C++ integers to TTCN-3, their sizes must be restricted according to the properties of the system used. Mapping of integers is shown in Table 4 below.

Table 4. Mapping of integer types from C++ to TTCN-3.

C++	TTCN-3
`short`	`type integer CppShort(-32768..32767);`
`signed short`	`type CppShort CppSignedShort;`
`short int`	`type CppShort CppShortInt;`
`unsigned short`	`type integer CppUnsignedShort(0..65535);`
`int`	`type integer CppInt(-2147483648..2147483647);`
`unsigned int`	`type integer CppUnsignedInt(0..4294967295);`
`long`	`type integer CppLong(-2147483648..2147483647);`
`unsigned long`	`type integer CppUnsignedLong(0..4294967295);`
`// etc...`	`// etc...`

In addition, values for C++ integers can be assigned in one of the integer literal form: decimal, octal, hexadecimal. TTCN-3 provides conversion functions to make transformations between integer literals. Table 5 below gives some examples of using literals.

Table 5. Examples of using integer literals in TTCN-3.

C++	TTCN-3
`int i = 1;`	`var CppInt i := 1;`
`i = 0xFF;`	`i := hex2int('FF'H);`
`i = 077;`	`i := oct2int('77'O);`

However, conversion functions return a positive integer, which means that they cannot be used for negative numbers. Similarly, C++ *unsigned int* accepts negative values, but *CppUnsignedInt* (declared in Table 4) does not. In these kinds of situations, conversion must be done by hand.

For example, for a C++ variable

```
unsigned int UINT = -1;
```

the corresponding TTCN-3 representation would be

```
var CppUnsignedInt UINT := hex2int('FFFFFFFF'H);
```

Similary, for a C++ variable

```
int minusOne = 0xFFFFFFFF;
```

the corresponding TTCN-3 representation is

```
var CppInt minusOne := -1;
```

4.4. Floating Point Types

There are three different floating point types in C++, which are *float*, *double* and *long double*. These types differ from each other by size, which determines the precision of a value. That is, the set of values of float is a subset of values of double, which is again a subset of the values of long double. Moreover, the value representation is implementation-defined. That is, for example *double* and *long double* could be exactly the same in some systems. [30]

In TTCN-3, a key word *float* describes floating point numbers, which can be expressed in two different ways. First is the normal dot notation and second is by using two numbers, mantissa and exponent, which are separated by E [16]. The precision is not restricted in TTCN-3, that is why all C++ types *(float, double, long double*) can be mapped to TTCN-3 in the same way. However, the ranges should be declared according to the properties of the used system. Table 6 provides a mapping for floating point types from C++ to TTCN-3.

Table 6. Mapping of floating point types from C++ toTTCN-3.

C++	TTCN-3
float	**type float** CppFloat (-3.402823466E38 .. 3.402823466E38);
double	**type float** CppDouble (-1.7976931348623158E308 .. 1.7976931348623158E308);
long double	**type float** CppLongDouble (-1.7976931348623158E308 .. 1.7976931348623158E308);

5. Mapping of C++ Compound Types to TTCN-3

As mentioned earlier, types are divided into two groups in C++: fundamental types and compound types. Compound types are typically constructed from fundamental types as follows [30]:

- array, i.e. an indexed sequence of value elements of a given type,
- function, which may have parameters and a return value of given types,
- pointer to any fundamental or compound type,
- reference to any fundamental or compound type,
- class (or struct), i.e. a data structure that can contain member data, member functions, member constants, and member types,
- union, i.e. a class in which all members are allocated at the same offset within an object,
- enumeration, i.e. a set of named constant values, or
- pointer to non-static class member.

The mapping of compound types in this chapter is divided into the following parts: user-defined types (i.e. *class, struct, union, and enum*), pointers, references, arrays, type definition, and template. Even though in [30] functions are included in compound types, the mapping of a function is excluded from data type mapping in this work, because functions are not actual data types. User-defined types can be mapped from C++ to TTCN-3 fairly straightforwardly but, for example, mapping of pointers due to their ambiguity has proven to be very complicated.

5.1. User-defined Types

5.1.1. Class and Structure

C++ structural types *class* and *struct* are almost similar, the only difference is that all members in a struct are public by default, whereas in class they are private. Since this difference does not affect the mapping, this chapter deals only with the mapping of class and the structure can be mapped in exactly the same way.

Class is the most common user defined type in C++. The purpose of classes is to provide a tool which can be used to create new types that can be used in the same way as the fundamental types. However, classes are created for a special purpose and they may contain more data [31]. There are several benefits in using classes, for example, inheritance. However, these benefits are based on the object-oriented features, which are not supported in TTCN-3.

In addition, class provides access specifiers, which are *public*, *protected* and *private*. These specifiers are used to restrict the access of data members of a class. In short, private members can be used by member functions or friends of a class; protected members can be used as private members and also by member functions and friend derived from this class; public members can be used by any function. In class, all members are private by default. [31]

TTCN-3 does not support this kind of specifiers. Therefore, there is no way to restrict the access of data elements in TTCN-3. However, it is important to see that when we are implementing the test cases, we are not interested in who is able to access data within test cases. On the contrary, sometimes we might be interested in how to manipulate protected data.

All in all, in this case we are mapping C++ data types and they should be thought of, as just data structures. These kinds of basic classes can be mapped to TTCN-3, by using TTCN-3 structured type *record*. The meaning of basic class in this case is that the member functions are not dealt with. Table 7 below shows a simple example of how C++ class can be mapped to TTCN-3.

Table 7. Mapping of basic C++ class to TTCN-3 [1].

<table>
<tr><th>C++</th><th>TTCN-3</th></tr>
<tr><td><pre>class Person {
private:
    char* name;
    char* addr;
    int age;
};</pre></td><td><pre>type record Person {
    CppString name,
    CppString addr,
    CppInt age
}</pre></td></tr>
</table>

[1] Mapping of type *char** will be considered more in chapter 5.2, which deals with the pointers.

However, C++ classes may inherit data from the other classes, and this complicates the mapping. Table 8 below shows one solution of how inheritance can be mapped from C++ to TTCN-3. In this example *MyClass2* is said to be derived from its base class *MyClass1*, or in other words *MyClass2* is subclass and *MyClass1* is its superclass.

Table 8. Mapping of inheritance from C++ to TTCN-3.

<table>
<tr><th>C++</th><th>TTCN-3</th></tr>
<tr><td><pre>class MyClass1 {
public:
    int number;
    char character;
};

class MyClass2 : public MyClass1 {
    char* string;
};</pre></td><td><pre>module MyClass1 {
    type record MyClass1Type {
        CppInt number,
        CppChar character
    }
}

module MyClass2 {
    import from MyClass1 all;
    type record MyClass2Type {
        MyClass1Type super,
        CppString string
    }
}</pre></td></tr>
</table>

Multiple inheritance can be mapped in the same way as single inheritance, i.e. super classes are declared as fields in the subclass type. However, if there is a class that inherits two classes that are both derived from the same super class, are things more complicated, see Figure 9 below.

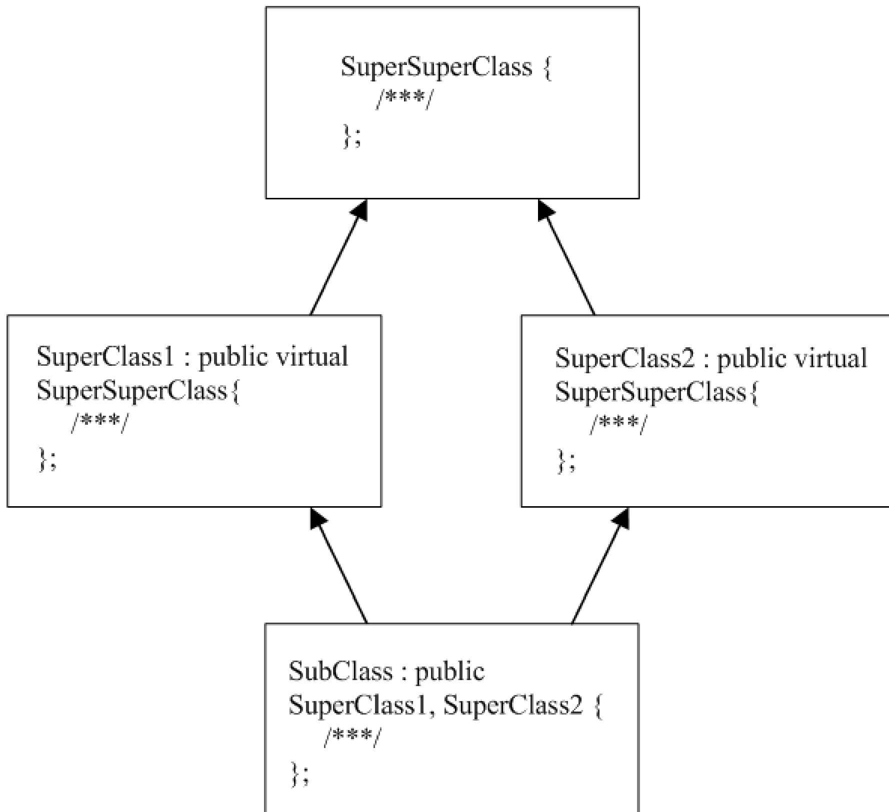


Figure 9. Class diagram for the multiple inheritance.

In this figure *SuperClass1* and *SuperClass2* use virtual inheritance, that is, an object of *SubClass* contains exactly one instance of *SuperSuperClass*. This kind of situation is impossible to depict in TTCN-3, because if we declare both *SuperClass1* and *SuperClass2* (which both contain the field *SuperSuperClass*) in a module of *SubClass*, we end up with two instances of *SuperSuperClass*. This is a very unusual situation but if it happens, the programmer must use the same data values for both instances that are derived from *SuperSuperClass*.

In addition, in C++, data can be defined as constant, static, or static constant within a class. Constant members should be mapped as normal member data, i.e. within a record. The programmer must however notice that constant members must be used as constants. Static data should be accessed and modified by using TTCN-3 *external function*s. Constant static members can simply be mapped to external constants. Examples of mapping static and constant member data are shown in Table 9 below.

Table 9. Mapping of constant and static member data from C++ to TTCN-3.

<table>
<tr><th>C++</th><th>TTCN-3</th></tr>
<tr><td><pre>class MyClass {
    const int constInt;
    static int staticInt;
    static const int constStaticInt;
};</pre></td><td><pre>module MyClass {
    type record MyClassType {
        CppInt constInt
    }
    external function Set_staticInt (
        in CppInt staticInt);
    external function Get_staticInt( )
        return CppInt;
    external constant CppInt constStaticInt;
}</pre></td></tr>
</table>

5.1.2. Union

A union is similar to class, but it can hold only a value of one member at a time. Therefore, the memory space allocated for a union is only as much as the size of its largest member. Union is almost the same both in C++ and TTCN-3. The only difference is that C++ provides anonymous unions, which is a union without a name. This kind of union is not a type itself, but normally it is declared within a class. It is also noticeable that either language does not keep track of which member's value is held by a union, but this is left up to the programmer. Table 10 gives examples of union mappings. [31][10]

Table 10. Mapping of union from C++ to TTCN-3.

<table>
<tr><th>C++</th><th>TTCN-3</th></tr>
<tr><td><pre>union U {
    int i;
    char c;
};</pre></td><td><pre>type union U {
    CppInt i,
    CppChar c
}</pre></td></tr>
<tr><td><pre>class Person {
public:
    char* name;
    int age;
    union {
        char* addr;
        int phone;
    };
};</pre></td><td><pre>type union Anonymous {
    CppString addr,
    CppInt phone
}
type record Person {
    CppString name,
    CppInt age,
    Anonymous contact
}</pre></td></tr>
</table>

5.1.3. Enumerated Types

Enumeration, specified by the keyword *enum* in C++, is a type that can hold a distinct named set of values specified by the user [31]. Thus, every item in this set has a unique name and corresponding integer value. These values are started increasing from zero by default, but they can also be initialized with other values. The elements in an *enum* can be thought as constants, which can be used later in a program to distinguish them from each other.

TTCN-3 supports enumerated types as well, but they differ from C++ *enum* considerably. For example, C++ accepts *enum* value to be used as an integer value, thus an enum variable can used in a place of an integer to be passed as an argument in a function call. In contrast, TTCN-3 enumerators can be used only by their identifiers. In addition, C++ enumerators also accept negative values, which are prevented in TTCN-3 and might cause problems in some situations. Moreover, in C++ it is possible to define an enumeration without a name, in which case members of enumeration are named integer constants.

From all appearances, C++ *enum* should be mapped to constant integers in TTCN-3. By doing so, all of the difference stated above could be avoided. Table 11 below shows the mapping of enumeration.

Table 11. Mapping of enumeration from C++ and TTCN-3.

C++	TTCN-3

`enum Days {`
`    Monday = 1,`
`    Tuesday = 3,`
`    Wednesday`
`};`
	`const CppInt Monday := 1;`
	`const CppInt Tuesday := 3;`
	`const CppInt Wednesday := 4;`

`enum Days { Sunday = -5 };`	`const CppInt Sunday := -5;`
`enum { Mo, Tu, We };`	`const CppInt Mo := 0, Tu := 1, We := 2;`

5.2. Pointers

5.2.1. Review of C++ Pointers

The meaning of a pointer is to point to an address, in which some data is stored. In other words, a pointer specifies the address of a variable. When a variable is being created, the system allocates memory space from a concrete location of a memory using some allocation strategy. The programmer cannot determine where in the memory a variable is being placed, but this is done by a compiler and the operating system. However, the programmer might be interested in knowing the address of this location.

Pointer is defined by using asterisk character (*) in C++. This is called a unary * operator and it performs indirection, i.e. the expression of a pointer to a variable [30]. This * operator can be read as “value pointed to by” or “pointer to”. For example, if there is a type T, T* can hold the address of an object, which has the type of T. [31]

Pointers are very useful and it is one of the great advantages in C++. The most common use of pointers is to pass them to a function as an argument or to reference to the complex data types such as class. The other very essential purpose of pointers is achieved when

dynamic data is being used. That is, one can allocate dynamic arrays by using pointers, for example.

Pointer can point to any types. It is however important to specify a correct type for a pointer, because the amount of storage that is to be reserved for the value pointed to by a pointer is based on its type. In addition arithmetic operations are performed according to the type of pointer. There is also a special type pointer, *void* pointer, which can point to any type. Void pointer is a kind of exception, and for example arithmetic operations are not allowed with them. Figure 10 below shows different types of pointers. [30]

```
struct T {
    /* ... */
};
T* p_to_T;                    // pointer to type T

char c;
char* p_to_c = &c;            // pointer to char

double d;
double* p_to_d = &d;          // pointer to d
double** pp_to_d = &p_to_d;   // pointer to pointer to d

int* p_arr[5];                // array of 5 pointers to int: p[0]...p[4]
int (*p_to_arr)[5]            // pointer to array of 5 int
void* vp;                     // pointer to any type
```

Figure 10. Different types of pointers.

As described earlier, pointer can also point to a function. Therefore, functions can be used in two ways; call them or take its address. Pointer to function may be useful, since one can pass a function as an argument to another function. Pointer to function is declared in the same way as a normal function, except that the name must be in parentheses and before the name is an asterisk (*). An example of how pointer to function can be used is depicted in Figure 11.

```
// This function takes a pointer to function as an argument
void MyFunction (void (*pToFunc) (char*) ) {
     /*...*/
     pToFunc ("Hello World!");
     /*...*/
}
void AnotherFunction (char* string) {
     /*...*/
     cout << string << endl;
     /*...*/
}
void (*myPtr) (char*);         // pointer to function
int main () {
      myPtr = AnotherFunction; // myPtr points to AnotherFunction
      myPtr (“Hello World!”);  // call MyFunction through myPtr
      MyFunction (myPtr);      // call MyFunction and pass myPtr as a parameter
}
```

Figure 11. Pointer to function example.

Pointers and arrays are pretty much the same in C++. The name of an array is actually the same as the pointer to its first element [31]. Therefore, in order to implement a dynamic array, one can simply define a pointer of a certain type and use it as an array. Similarly, dynamic two-dimensional array would be a double pointer (i.e. pointer to pointer). In fact, a two-dimensional array is not exactly the same as a double pointer, because of the different memory allocation strategy. However, consecutive pointers can be used as multidimensional arrays, even though it is not recommended because such structures are vulnerable to errors.

In addition, in previous versions of C and C++, a string literal was defined as *char**, and it is still allowed to be used [31]. That is, *char** can be used as a string, for example within print commands (e.g. printf or cout).

5.2.2. Pointer to Basic Types

As mentioned earlier, pointers cannot be created within a TTCN-3 module. That is why some kind of model of pointers should be put into practice. However, bringing the func-

tionality of pointers to TTCN-3 has proven pretty challenging, because of the wide range of facilities of C++ pointers.

The pointer problem have been solved in this work by creating four external functions for each C++ type, which can be used to manipulate pointers in TTCN-3. First an external function can be used to create a pointer, second to get a value of the pointer, third to set a new value, and fourth is for deleting the pointer. Since every single C++ type needs at least four unique functions, there will be quite many functions altogether. However, these functions can be implemented in the separate library module, which can be imported to the other modules. As an example, external function declarations that are needed for the mapping of pointer to integer are shown in Figure 12 below.

```
type integer CppPtr;              // type definition for the pointer
type record of CppInt CppIntList;  // type definition for the array of C++ integers

external function New_CppInt (in CppIntList aList) return CppPtr;
external function Del_CppInt (in CppPtr ptr);
external function Get_CppInt (in CppPtr ptr, in integer index) return CppInt;
external function Set_CppInt (in CppPtr ptr, in integer index, in CppInt aValue);
```

Figure 12. External functions for the mapping of pointer to integer.

First of all, when we want to create a new pointer, we have to allocate memory for the value of the pointer. This is done by calling the appropriate external function, for example *New_CppInt()*, which takes one variable as a parameter and returns the pointer (i.e. address) of this value. As it was described earlier in section 5.2.1, pointers can be used in two different ways: as a pointer to a certain value or as a dynamic array. That is why, in order to achieve the efficient use of pointers, parameters (e.g. *CppIntList*) have the type *record of* (i.e. list of values), and so this system can be used either way. In the case it is used as an array, the argument is a list of values to be stored in the array and the return value is a pointer to its first element, just as in C++. On the contrary, when pointers are used in a normal way, i.e. pointer to a certain value, the argument list contains only one value.

Because the pointers are hidden from the programmer in TTCN-3, values of the pointers must be hidden behind the TRI/TCI interfaces. Consequently, when the function *New_CppInt()* is executed, the memory space for the values in *CppIntList* is allocated, and the pointer (i.e. return value) is the address of this memory location.

Once the memory space for the value pointed to by a pointer has been reserved, the garbage collection must be also taken care of. That is why appropriate delete functions for each type have been implemented (e.g. *Del_CppInt()* in Figure 12). Thus, every time when a pointer is created by using one of the creation functions (e.g. *New_CppInt()*), they must be destroyed before the end of test cases by using the appropriate delete function.

The third type of external function (e.g. *Get_CppInt()* in Figure 12) is used to fetch a value that is pointed to by a pointer. Hence, the pointer is passed as an argument to the function and the return value is the value of a certain type, which the pointer points to. These external functions take also another argument, which in the case of a dynamic array describes the index of the desired value. In a normal case, the second argument must be zero.

In addition, there must be also some functions that can be used to set a new value to a dynamic array or to change the value pointed to by the pointer (e.g. *Set_CppInt()* in Figure 12). It is also important to see that in the case of a dynamic array, by using these functions, only one value can be assigned at a time.

Mapping of pointer to any fundamental type can be done in exactly the same way as it is done for the integer. However, C++ type *char** is a special case; it can be thought as a pointer to char or as a string. If we want to use *char** as a string, the mechanism provided earlier does not work well, since every letter should be separated by a comma. That is why unique external functions for mapping of strings from C++ to TTCN-3 have been implemented. Declarations of these external functions are described in Figure 13 below. In these functions *char** is thought to be a string, and thus it is mapped to the *charstring* type.

```
type integer CppPtr;       // type definition for the pointer
type charstring CppString; // type definition for the C++ value char*

external function New_CppString (in CppString string) return CppPtr;
external function Get_CppString (in CppPtr ptr) return CppString;
external function Set_CppString (in CppPtr ptr, in integer index, in CppString
aValue);
```

Figure 13. External functions for the mapping of C++ type char to TTCN-3.*

Now we have aids which can be used to map C++ pointers to TTCN-3. Thus, when a user writes TTCN-3 test software, type mappings from C++ to TTCN-3 can be done according to these rules. Table 12 below provides a few examples of how mapping of pointer can be performed, and also clarifies how earlier described external functions are intended to be used.

Table 12. Examples of mapping of pointers from C++ to TTCN-3.

C++	TTCN-3

`double d = 1.2345;`
`double* ptr_to_d = &d;`
`double d2 = *ptr_to_d;`
`d = 5.4321;`
	`var CppDouble d := 1.2345;`
	`var CppPtr ptr_to_d := New_CppDouble ({d});`
	`var CppDouble d2 := Get_CppDouble (ptr_to_d, 0);`
	`d := 5.4321;`
	`Set_CppDouble (ptr_to_d, 0, d);`
	`//...`
	`Del_CppDouble(ptr_to_d);`

`int array[] = {1, 2, 3, 4, 5};`
`int* dyn_array = array;`
`int second = dyn_array[1];`
`dyn_array[1] = 0;`
	`var CppIntList array := {1, 2, 3, 4, 5};`
	`var CppPtr dyn_array := New_CppInt (array);`
	`var CppInt second := Get_CppInt(dyn_array, 1);`
	`Set_CppInt (dyn_array, 1, 0);`
	`//...`
	`Del_CppInt(dyn_array);`

`char* string;`
`string = "Hello World!";`
	`var CppPtr string;`
	`string := New_CppString ("Hello World!");`
	`// ...`
	`Del_CppString(string);`

The first example in Table 12 illustrates the normal usage of a pointer. It is important to see that in the function *New_CppDouble()*, argument *d* must be put in curly brackets, because the type of this argument is *record of.* In this case, there is only one item in that set. In the function *Get_CppDouble()*, the index value is zero, which means that very first element is being fetched.

The second example illustrates how the pointer can be used as a dynamic array. In this example, variable *CppIntList* has the type *record of* and that is why variable *array* in the function *New_CppInt()* is not put in curly brackets. Furthermore, the second element from the array is fetched in this example, and after that it is changed to zero. The third example simply shows how C++ type *char** can be mapped to TTCN-3, when it is used as a string.

As one can see, it is complicated to bring all the functionality of C++ pointers to TTCN-3 and that is why this mapping needs lots of effort.

5.2.3. Pointer to Class

Basically, pointer to class (or struct or union) can be implemented in the same way as a pointer to basic type that was described in the previous sub-chapter. However, every time pointer to class is created, unique external functions must be implemented. These external functions cannot be imported from the library module, since they are user-defined. Another difference is that the pointer to class type is not treated as a dynamic array. Instead, a pointer to a class always points to a single value, not array of objects. As an example, there is a simple class mapped from C++ to TTCN-3 in Table 13 below.

Table 13. An example of mapping C++ struct to TTCN-3.

C++	TTCN-3

`struct MyClass {`
`    int i;`
`    char* str;`
`};`
	`type record MyClass {`
	`    CppInt i,`
	`    CppString str,`
	`}`

In order to create a pointer to *MyClass*, we need to implement the appropriate external functions, which are declared in Figure 14 below.

```
// to create a pointer to MyClass
external function New_MyClass (in MyClass class) return CppPtr;
// to get the values of MyClass by using its pointer
external function Get_MyClass (in CppPtr ptr) return MyClass;
// to set the new values for MyClass
external function Set_MyClass (in CppPtr ptr, in MyClass newValue);
// to release the memory of MyClass
external function Del_MyClass (in CppPtr ptr);
```

Figure 14. Needed external function declarations for a pointer to MyClass.

5.2.4. Pointer to Pointer

Pointer to pointer can be simply mapped by creating a pointer to some variable and then create a new pointer to the earlier created pointer. However we need to implement additional external functions, which can be used to create pointer to pointer. Declarations of these external functions are shown in Figure 15.

```
// type definition for an array of pointers
type record of CppPtr CppPtrList;
external function New_CppPtr (in CppPtrList aList) return CppPtr;
external function Get_CppPtr (in CppPtr ptr, in CppInt index) return CppPtr;
external function Del_CppPtr (in CppPtr ptr);
```

Figure 15. External function declarations for the pointer to pointer.

By using this mechanism one can create a limitless consecutive pointer (e.g. pointer to pointer to … pointer to int). An example of mapping of a double pointer is shown in Table 14.

Table 14. Mapping of pointer to pointer.

C++	TTCN-3
`int i1 = 99;`	`var CppInt i1 := 99;`
`int* firstPtr = &i1;`	`var CppPtr firstPtr := New_CppInt({i1});`
`int** doublePtr = &firstPtr;`	`var CppPtr doublePtr := New_CppPtr({firstPtr});`
`int i2 = **doublePtr;`	`var CppPtr temp := Get_CppPtr(doublePtr, 0);`
	`var CppInt i2 := Get_CppPtr(temp, 0);`
	`// ...`
	`Del_CppPtr(firstPtr);`
	`Del_CppInt(doublePtr);`

5.2.5. Pointer to Other Types

Pointer to function or pointer to non-static member is a special case of pointers, which are used very rarely. That is why they can be thought to be case-specific, and therefore every time when, for example, pointer to function is created, unique external functions must be implemented. Similarly pointer to non-static member would need unique external functions for each event. Table 15 below gives a simple example of how pointer to function can be mapped.

Table 15. An example of mapping of pointer to function from C++ to TTCN-3.

C++	TTCN-3
`// Hello-function declaration`	`// external function declaration`
`void Hello (char* str);`	`external function Get_PtrToHello() return CppPtr;`
`// pointer to Hello-function`	`// This returns pointer to Hello-function`
`void (*pToFunc) (char*);`	`var CppPtr pToFunc := Get_PtrToHello();`
`pToFunc = Hello;`	

In fact, when pointer to function is being created, we do not need to allocate memory space for the function, because this is already done by the C++ compiler at compile time. In other words, *New()* and *Delete()* external functions are not needed.

5.3. References

In [31] a reference is declared as “an alternative name for an object”. The main purposes of C++ references are to specify arguments or return values for functions in general, and to use them with the overloaded operators in particular.

Reference is declared by using the ampersand (&) in C++ and it can be read as “reference to” or “address of”. It is important to note that reference is not a variable. Thus for example you can not change it to refer to another variable after initialization. [31]

In fact, reference is based on the same concept as pointer, i.e. both are referring to an object by its address. In the case of testing, references are only needed in the situations in which function arguments or return values are being specified. Thus, mapping of C++ references can be done in the same way as the mapping of C++ pointers described in Chapter 5.2.

5.4. Arrays

An array is a collection of the value elements of a certain type. Values in the array are not named but they are accessed by their position. Arrays can be constructed from almost every kind of type in C++, basically from any fundamental or compound type except *void* [30]. For a type T, the array is declared as T[n] that is the type “array of n elements of type T”. 'n' declares the number of elements in the array, which is also called the array bound. This is a constant expression, which is indexed from 0 to n-1. [31]

Chapter 5.2 deals with the pointers and there is also described how pointers are used as dynamic arrays. Consequently, when arrays are being mapped from C++ to TTCN-3, they should always be mapped to dynamic arrays, even though TTCN-3 core language [10] provides a mechanism to declare array in the same way as is done in C++. This is because in C++ any type of array is actually a pointer to its first element. Thus, the arrays can be mapped using external functions declared for the pointers in chapter 5.2. Table 16 below shows some examples of how arrays can be mapped from C++ to TTCN-3.

Table 16. Examples of mapping of arrays from C++ to TTCN-3.

C++	TTCN-3

`int intArr[] = {1,2,3,4,5};`
 	
`int last = intArr[4];`
	`var CppPtr intArr := New_CppInt({1,2,3,4,5});`
	
	`var CppInt last := Get_CppInt(intArr, 4);`

`double doubleArr[3] = {1.1,`
 	
`1.2, 1.3};`
	`var CppPtr doubleArr := New_CppDouble({1.1,`
	`1.2, 1.3});`

`char charArr[5] = {'H','i','!'};`	`var CppPtr charArr :=` `New_CppChar({"H","i","!"," "," "});`

`// Example of two-dimensional array`
 	
`int twoDimArr[3][5] = {{1,2,3,4,5},`
`{6,7,8,9,10}, {11,12,13,14,15}};`
 	
`int x = 1, y = 2;`
 	
`int middle = twoDimArr[x][y];`
	`// two-dimensional array with TTCN-3`
	
	`var CppIntList x1 := {1,2,3,4,5}`
	
	`var CppIntList x2 := {6,7,8,9,10}`
	
	`var CppIntList x3 := {11,12,13,14,15}`
	
	`var CppPtr firstPtr := New_CppInt(x1);`
	
	`var CppPtr secondPtr := New_CppInt(x2);`
	
	`var CppPtr thirdPtr := New_CppInt(x3);`
	
	`var CppPtr twoDimArray :=`
	`New_CppPtr({firstPtr, secondPtr, thirdPtr});`
	
	`var CppPtr x := Get_CppPtr(twoDimArray, 1);`
	
	`var CppPtr y := 2;`
	
	`var CppInt middle := Get_CppInt(x, y);`
	
	`Del_CppPtr(twoDimArray);`
	
	`Del_CppInt(firstPtr);`
	
	`Del_CppInt(secondPtr);`
	
	`Del_CppInt(thirdPtr);`

The first three examples in Table 16 are rather simple. The only matter worth nothing is that if the programmer initializes only part of the array elements, the rest are initialized automatically by space characters (or by zeros).

Mapping of a two-dimensional array (fourth example in Table 16) is much more complicated. In C++ a two-dimensional array can be thought of as an array of arrays. For example, the expression *matrix[i][j]* is interpreted as **(*(matrix + i) + j)* in C++ [31]. An example in Table 16 is implemented according to this rule, that is, *twoDimArray* is a pointer to the first element of an array that contains pointers to the other arrays (cf. pointer to pointer in 5.2.4).

5.5. Type Definition

Type definition, which is defined by a keyword *typedef* in C++, declares a new name for the given type. TTCN-3 provides exactly the same mechanism to declare new types, but its corresponding keyword is *type*. Table 17 below gives some examples of using type definitions in both languages.

Table 17. Mapping of typedef.

C++	TTCN-3
`typedef int int32;`	`type CppInt int32;`
`typedef short int16;`	`type CppShort int16;`
`typedef unsigned char uchar;`	`type CppUnsignedChar uchar;`

5.6. Templates

The template mechanism in C++ provides direct support for generic programming, that is, types can be used as parameters. For example, in the definition of class or function a type can be given as a parameter, in which case the exact types are defined at compile time.

TTCN-3 does not accept types as parameters and thus, type mapping must be done in another way. Consequently, the mapping of a template can be done by implementing several TTCN-3 types for each C++ template, i.e. every type combination for which a template is tested needs a unique TTCN-3 type. For example, in Table 18 a template called *String* that contains two data type fields is implemented. Corresponding TTCN-3 representation requires several separated records. In this case *String* template is used only for the types *int* and *char*, which can be observed from the definitions for the functions *SortInt()* and *SortChar()* in Table 18. Hence, two distinct records are needed. In fact, normally we probably do not know which type definitions are going to be used, but a good way is to implement distinct records for each combinations tested.

Table 18. Template data type mapping from C++ to TTCN-3.

C++	TTCN-3

template <**class** C> **class** String {
private:
    C* mString;
    C aValue;
public:
    String();
    virtual ~String();
};
	type record String_IntType {
	    CppPtr mString,
	    CppInt aValue
	}
	type record String_CharType {
	    CppPtr mString,
	    CppChar aValue
	}
	// etc...

void SortInt (String<**int**> intArr, **int** len); **void** SortChar (String<**char**> string, **int** len);	**signature** SortInt (**in** String_IntType intArr, **in** CppInt len); **signature** SortChar (**in** String_CharType string, **in** CppInt len);

5.7. Conclusion

Mapping of C++ data types to TTCN-3 was divided into two groups; mapping of C++ fundamental types was described in Chapter 4 and mapping of C++ compound types was provided here in Chapter 5.

Mapping of fundamental types could be solved pretty straightforwardly, even though there are some differences between C++ fundamental type and the corresponding TTCN-3 types. Due to the following distinctions some problem were faced:

- Some aspects of C++ types are implementation-defined.
- C++ is based on a pure binary numeration system (e.g. two's complement), whereas TTCN-3 uses another numeration system.
- C++ allows different literal forms (e.g. *hex*, *oct*) for integers, whereas TTCN-3 treats literals as distinct types.
- TTCN-3 uses a 7 bit coded character set, and C++ usually uses 8-bit characters.

– Characters in C++ are based on the binary numeration system, whereas characters are just letters in TTCN-3.

Mapping of compound types due to the object-oriented features of C++ and ambiguity of C++ pointers was much more complicated. Basic classes (and structures) could be mapped easily, but some difficulties were faced because of the special properties of C++ classes, such as access specifiers, inheritance, multiple inheritance, virtual inheritance, as well as static, const and const static member data.

Pointers in C++ are very special and ambiguous. Thus, bringing out the feature of pointers in all of their variety was, to say the least, challenging. The mapping of pointers was carried out by means of TTCN-3 external functions and a very useful outcome was achieved.

All in all, rather uniform data type mapping rules were found.

6. Case Study: Using TTCN-3 to Test a C++ Module

This chapter presents the implementation of a TTCN-3 based test system for a C++ class. The goal of these tests is to verify the data type mappings provided earlier in chapters 4 and 5 and also evaluate the usability of TTCN-3 for module testing. Thus, tests are not used for verifying the correctness of the C++ software, but the test cases are chosen in a way that they would be utilizing type mappings for all they are worth. Therefore, this case study will be demonstrating the usability of data type mappings in a real world-like situation. In addition to test cases, also implementations for TRI (TTCN-3 Runtime Interface) and TCI (TTCN-3 Control Interface) are produced in this exercise.

6.1. Testing Environment

The following tools have been taken advantage of in this exercise:

- Telelogic Tau 2.2,
- Visual C++ 6.0, and
- NuMega BoundsChecker 6.50.

Telelogic Tau 2.2 [34] is a tool family, which can be used to develop real-time and other advanced software systems. It includes two tools: Telelogic Tau that is a model driven tool based on UML 2.0 (Unified Modeling Language) and Telelogic Tau/Tester that is for designing, creating, and executing TTCN-3 test suites. In this exercise only the latter has been used. Visual C++ is used to build the entire test system, link all parts together, compile and run entire test software. NuMega BoundsChecker is used to guarantee that memory leaks do not occur.

TTCN-3 execution interfaces TRI (TTCN-3 Runtime Interface) and TCI (TTCN-3 Control Interface), described in [14] and [15] respectively, are the main parts of TTCN-3 based test system (see Chapter 3.1.3). These interfaces define a standardized adaptation for the test system communication. However, the test system that is used in this exercise is not equivalent to a standardized TTCN-3 test system, because Telelogic Tau/Tester does not exactly fulfill the requirements of a standardized TCI interface. However, Telelogic's implementation for the CD (Coding and Decoding handling) interface pro-

vides similar functionality to standardized TCI, but other interfaces (i.e. TM – Test execution Management and CH – Component Handling) are not provided. Hence, this system corresponds well to standardized adaptation. In Figure 16 below is a simple block diagram, which describes the test system used in this exercise.

To begin with, TTCN-3 code (*MyCode.ttcn* in Figure 16) is written using some text editor. This TTCN-3 script includes among others the interface of the tested C++ module at the TTCN-3 language level, which is produced by following the type mapping rules created in chapters 4 and 5. Entire TTCN-3 code is compiled using Telelogic Tau/Tester 2.2, which produces the TTCN-3 executable (TE). The TE basically implements the concrete TTCN-3 test suite, which together with the other entities (i.e system adapter (SA), platform adapter (PA), encoder and decoder) produces a concrete test system. The PA is responsible for the realization of external function and timers and the SA is responsible for the communication with the SUT (System Under Test). SA and PA are defined by the TRI (TTCN-3 Runtime Interface) [14], which is drawn in blue in Figure 16. The role of the encoder is to convert TTCN-3 data (e.g. parameters passed by call or external functions) to a C++ form. The decoder realizes the opposite operation, which is decoding C++ data (e.g. return values) back to TTCN-3. In other words, the *encode()* and *decode()* functions implement the conversions of the data types used (i.e. test data) between TTCN-3 and C++ by utilizing type mappings. Encoder and decoder belong to the TCI (TTCN-3 Control Interface) [15] which is the red part in Figure 16.

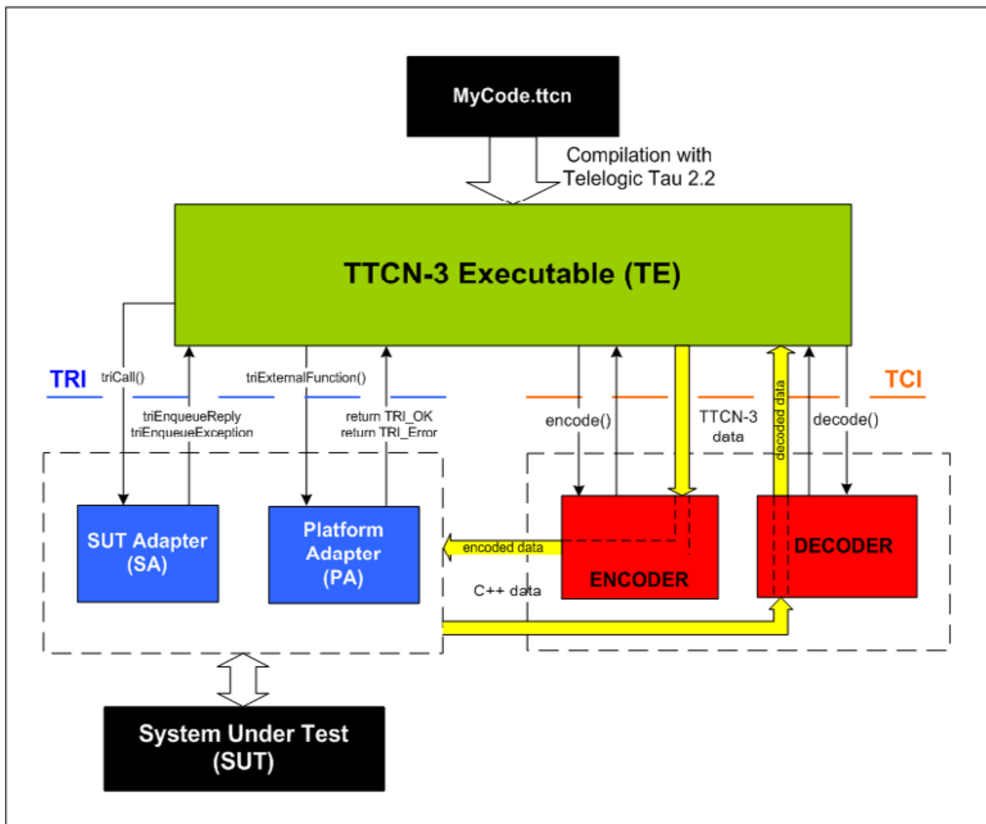


Figure 16. Block diagram of the used test system.

Yellow arrows in Figure 16 show the data flow, i.e. the route in which parameters and return values flow. Black arrows are those TRI/TCI functions which are used in this exercise. The function *triCall()* is used to perform a signature call, which corresponds to some function of the system under test. Every call is followed by either *triEnqueueReply()* or *triEnqueueException()*, which contains information such as return value, parameters (in-direction) or data of exception. The *triExternalFunction()* executes an external function (e.g. *New_CppPtr()*) which simply returns information if the execution was a success or not.

The purpose of *encode()* and *decode()* functions is to convert parameters and return values passed by calls and external functions from C++ to TTCN-3 and vice versa. For instance, when a signature call occurs, the system checks its parameters and converts all out-direction parameters into C++ by calling the *encode()* function several times, once for each parameter. *encode()* takes the required information (i.e. TTCN-3 type and its value) as parameters, converts TTCN-3 value to the C++ form and stores it into a parameter list.

When all parameters have been converted into C++ and stored in the parameter list, the function *triCall()* is called by passing the parameter list and other required information (e.g. port id, component id, SUT address, etc.) as parameters. Now a function of SUT can be called with the desired parameters. At the end of the *triCall()* the function *triEnqueueReply()* (or *triEnqueueException()*) is performed, and return data is stored in a parameter list for the decoder. Finally *decode()* is performed as many times as needed in order to convert return values in the in-direction parameters back to the TTCN-3 form.

All of the functions described earlier are standardized TRI/TCI functions. TRI and TCI contain also several other functions, but they are not used in this exercise.

The system under test in this exercise is a C++ software module. Thus, the interface of this module, which is described in a header file (i.e. afx.h) is to be mapped to TTCN-3 language. TTCN-3 executable (TE) in Figure 16 is produced from an abstract test suite (ATS) just as was described in section 3.1.1. The ATS describes the tests at an abstract level, i.e. test cases described by TTCN-3. Thus, by using the data type mapping provided in chapters 4 and 5, C++ data types described in the header file of the SUT are mapped to TTCN-3. Then these mapped data types can be used, for example, when test data is being produced in the test cases.

The TTCN-3 test script including test cases (i.e. *MyCode.ttcn*), is compiled with the Telelog Tau/Tester, which produces C code. Implementations of adapters (e.g. signature calls, external functions) and coders (i.e. encode and decode functions) are written by C++. Finally all parts of C/C++ codes are linked together and compiled by Visual C++ 6.0. So we get executable test software, which after execution produces the test results, such as final verdicts.

6.2. Tested Module

The software module to be tested in this case study is the CFile class. This is the base class for Microsoft Foundation file classes in the MFC library [35], which provides binary disk input/output services. In addition manipulation of text files and memory files is supported through its derived classes. CFile and its derived classes are normally used for general-purpose disk I/O operations.

Table 19 below gives a brief description of the member functions and their parameters, which will be tested later in this exercise [35].

Table 19. Description of some member functions and parameters of CFile class.

Member Functions	Description

`CFile( );`
`CFile( LPCTSTR lpczFileName,`
`       UINT nOpenFlags );`
`~CFile( );`
	Constructor and destructor of CFile class.
	
	lpczFileName: a string. Path to the desired file.
	nOpenFlags: unsigned int. Sharing and access mode, which specifies the action to take when opening the file.

`virtual BOOL Open (`
`    LPCTSTR lpszFileName,`
`    UINT nOpenFlags,`
`    CfileException* pError`
`);`
	Opens a file with an error-testing option.
	
	lpczFileName: string. Path to the desired file.
	nOpenFlags: unsigned int. Sharing and access mode, which specifies the action to take when opening the file.
	pError: pointer to a CFileException object that describes the error occurred, if any.
	return value: int. A Boolean value that describes whether or not an error has occurred.

`virtual void Write (`
`    const void* lpBuf,`
`    UINT nCount`
`);`
	Writes data from a buffer to the file associated with the CFile object.
	
	lpBuf: pointer to the buffer, which contains the data to be written to the file.
	nCount: unsigned int. The number of bytes to be transferred.

`virtual UINT Read (`
`    void* lpBuf,`
`    UINT nCount`
`);`
	Reads data into a buffer from the file associated with the CFile object.
	
	lpBuf: pointer to the buffer, into which data is read from the file.
	nCount: unsigned int. Maximum number of bytes to be read.
	return value: unsigned int. The number of bytes transferred.

`virtual LONG Seek (`
`    LONG lOff,`
`    UINT nFrom`
`);`
	Repositions the pointer in a previously opened file.
	
	lOff: long. Number of bytes to be moved the pointer
	nFrom: unsigned int. Pointer movement mode, which describes the spot, which pointer is started to move from.
	return value: long. The new byte offset from the beginning of the file.

Table 19. Continues...

`virtual CFile* Duplicate( );`	Constructs a duplicate CFile object for a given file. **return value**: pointer to a duplicate CFile object.
`static BOOL GetStatus ( LPCTSTR lpszFileName, CFileStatus& rStatus );`	Retrieves the status of the desired file. **lpczFileName**: string. Path to the desired file. **rStatus**: reference to a CFileStatus structure that will receive the status information. **return value**: int. A Boolean value that describes whether or not the status information is successfully obtained.

6.3. TTCN-3 Test Software

The TTCN-3 test script for this exercise consists of four different files, and each file contains one module. The first module, called *Cpp*, consists of the type definitions for the C++ fundamental types, and the external function declarations that are needed in order to use pointers within the TTCN-3 modules. The second module, named the *T3_CFile*, contains all type definitions of *CFile* class that are needed for this exercise. The rest of the necessary type definitions are collected in the third module, called *MSDN_Typedefs*. These three modules are produced according to the mapping rules provided in Chapters 4 and 5, and they compose a static part of the TTCN-3 test data. The fourth module called *CFile_test* holds all implementations for the test cases and realization of the control part; thus it composes a dynamic part of test data. All of static test data (i.e. modules *Cpp*, *T3_CFile* and *MSDN_Typedefs*) are imported into the dynamic part (i.e. *CFile_test* module), thus they can be utilized when the test cases are being implemented. After the execution of the tests, NuMega BoundsChecker 6.50 is used for ensuring that there are no memory leaks the in program.

The following sub-chapters provide a more specific description of each module. The entire TTCN-3 source code is attached in Appendices 1 through 4.

6.3.1. Mappings for C++ Fundamental Types and Pointers

The module that contains the type definitions for the C++ fundamental types and external function declarations for the mapping of C++ pointer is named *Cpp*. This module can be thought of as a library file, because it should be unchanging, even though the current version is not complete. However it is sufficient for this exercise and all necessary types and pointer features can be accomplished by importing this module to the other modules. The entire TTCN-3 script of this module is in Appendix 1.

The first part of this module contains data type definitions (lines 15 through 28). The naming practice is simply implemented by using the prefix *Cpp* and appropriate type name and putting them together; for example *unsigned long* is declared as *CppUnsigned-Long*. In addition, integer and floating point types contain the range of acceptable numbers. These data type definitions are implemented according to the mapping rules described in Chapter 4.

Another section in this module (lines 36 through 74) holds declarations for the external functions, which are used to bring up the feature of pointers. In C++ the asterisk character (*) signifies the pointer, but it is not allowed by TTCN-3, and therefore it is replaced by the abbreviation *CppPtr*. Hence, each pointer, regardless of the type of it, is defined as *CppPtr* that can be thought as *void**. However, each type needs the unique external functions for creating and deleting the pointer, and setting and getting its value. This module provides external function declarations for:

- pointer to char (lines 46–50),
- pointer to string (i.e. *char**, lines 53–59),
- pointer to int (lines 62–66),
- pointer to double (lines 69–74), and
- pointer to another pointer (i.e. double pointer, lines 40–43).

Thus, the current version of this module is not perfect for the library, since the pointer feature for rarer types are excluded from it. All of the pointer mappings are implemented according to the rules that were found out section 5.2.2.

6.3.2. Mapping for the CFile Class

The mapping for *CFile* class is implemented in a module called *T3_CFile*, which stands for "TTCN-3 representation for the CFile class". In this exercise, *CFile* class is the system under test, and thus, the purpose of the *T3_CFile* module is to represent the interface of *CFile* class at the TTCN-3 language level. The TTCN-3 source code of the *T3_CFile* is in Appendix 2.

Generally, the mapping of the entire C++ class must also deal with the architectural aspect, that is, in addition to the member data, for example member functions should also be implemented somehow. In fact, this issue has been, among others, one of the research topics in this project, but has not been included in this thesis. This *CFile* mapping is implemented according to the rules that have been defined in this research. A complete C++ to TTCN-3 language mapping is presented in the project document [36], but it is not a public document. However these results will be published in the form of conference papers in the near future.

At the beginning of this module, two import statements are declared, meaning that all the type definitions in the modules *Cpp* and *MSDN_Typedefs* are available also in this module. This is followed by the type definitions for the test component and communication port. The test component is named *CFileTSI*, meaning "Test System Interface of CFile class". This exercise uses procedure-based communication, in which communication between the test system and system under test are performed by means of procedure signatures. That is in our case, each member function of *CFile* class is mapped to a procedure signature. Thus, the definition of *CFilePort* (in lines 17 through 29 in Appendix 2) declares all procedure signatures. The actual definitions for signatures are at the end of this module (lines 92 through 111). Every non-static member functions, except constructors, take an extra argument, which is a pointer to an object of *CFile* class (corresponding to *this* pointer in C++). Since the constructor creates an object, it returns *this* pointer, i.e. the pointer to the object itself. In contrast, the destructor takes an argument that is the pointer to the object to be deleted. It is also noticeable that overloaded C++ member functions are mapped to several distinct procedure signatures.

Lines 35, 36 and 37 define three external functions, which are used for manipulating pointers to the *CFile* class. After that, the mapping of *CFile* data types is carried out (lines 46 through 50); *CFile* class contains only three data type fields, which are defined as optional. This is because while a pointer to a *CFile* class is being declared, the values

of *CFile* members can be omitted and thus C++ default values are being used. In consequence, in the next declaration, that is the "empty" signature template, the value *omit* is given to all *CFile* members (lines 52–57). The rest part of mapping (lines 59–90) is for the four enum structures that the *CFile* class contains. These are the flag values that are used, for example, while opening or creating a file. Flag values are mapped to constant integers as it was described in Chapter 5.1.3.

All in all, the *T3_CFile* module contains all necessary type mappings for the *CFile* class. However some of the member function declarations are excluded from this mapping, because they are not used in this exercise.

6.3.3. Other Data Types

The *CFile* class contains many more yet undefined types that the earlier described modules (*Cpp* and *T3_CFile*) include. Therefore, the remaining necessary type definitions have been collected into one module. These types are mostly special for the MSDN (The Microsoft Developer Network) library, and thus this module is named *MSDN_Typedefs*. This module can be found from Appendix 3.

The module starts with an import statement that imports all definitions from the *Cpp* module. This is followed by the MSDN specific type definitions, such as LPTSTR (i.e. Long Pointer To STRing, lines 13 through 30). Besides the *CFile* class the following classes were also needed in this case study: *CFileException*, *CException*, *CString*, *CFileStatus* and *CTime*. These classes, or actually pointers or references to these classes, are passed as arguments by *CFile* member functions. Thus, corresponding TTCN-3 representations for these classes are also needed and they are placed into this module (lines 34 through 136). Basically, according to [36], each class should be mapped to a module. However, in this case only data types are being mapped, and thus, all of these data types are mapped in the same module. However, mappings of these classes are implemented according to the mapping rules that were presented in Chapter 5. At the end of this module two external functions called *FileExists()* and *CompareFileToBuffer()* are declared. These are used in the test cases to check out the test results.

6.3.4. Test Cases

The main module that contains all test cases is called *CFile_test*. This module includes nine test cases altogether, which are intended to verify the correctness of the functions described in Table 19. In addition, this module contains the control part in which test cases are executed sequentially. The specification for the *CFile* class is provided in [35], on which these test cases are based. Appendix 4 contains TTCN-3 source code for the *CFile_test* module.

So far the definitions for the data types, procedure signatures and also ports and components are implemented. Thus, according to section 3.2 in this work, test data definitions and execution of test cases are still required in order to fully implement tests. Test data is produced in parallel with test case implementations, because test data contains among others pointers and references, and thus, templates cannot be taken advantage of. The tests in this module are implemented by calling a function to be tested with certain parameters, and then a set of checks are performed. If the expected results were achieved, a test case passes, otherwise it fails. Some test cases also include preconditions, i.e. for example to call an earlier tested, so called, trusted function.

At the beginning of the module there are defined common functions and alt statements (i.e. alternative statement, described in section 3.1.2), which are used in test cases. The first test case simply verifies that the constructor and destructor of *CFile* class work faultlessly. Since the constructor and destructor are needed also in other test cases, these are implemented in the functions of their own (lines 55–69 and 71–78, respectively). The constructor does not take parameters, but according to the architectural mapping provided in [36], it returns a pointer to the created object. Correspondingly, the destructor takes *this* pointer as an argument, and destroys the object pointed to by *this* pointer.

The second test case (lines 104–130) verifies the correctness of the *CFile* constructor as well, but this case is little bit more complex; in this case, the constructor is called with two parameters. The first parameter is a pointer to a string (i.e. *char**) and the second is an unsigned integer that specifies the open flags (i.e. the action to take when opening the file). By calling the constructor with these two parameters, the file with the given path is opened and a *CFile* object is created. Thus, at the end of the test case it is made sure that the file exists by calling the external function *FileExists()*.

The next two test cases verify that the opening of a file works correctly. The open function takes three arguments, and again, according to architectural mapping [36], one extra argument is taken, which is *this* pointer (i.e. pointer to the *CFile* object). Three other arguments are: pointer to a file name, unsigned integer (i.e. open flags), and pointer to an existing *CFileException* object. Thus, before we call the open function, we must create two pointers, i.e. pointer to string (line 146) and pointer to an object of *CFileException* class (lines 155 and 156). Pointers are implemented according to the rules presented in section 5.2. So, when the required arguments are created, the open function is called with these arguments (line 162). The open function returns a boolean value, which can be used to verify whether the open was successful on not. At the end of the test case, an external function *FileExists()* is executed to make sure that the file was created successfully.

In the other test case implemented for verifying the file opening (lines 195–243), it is tried to open a file with such arguments that the open is unsuccessful. There is used a file name that does not exist, and this file is tried to open in the read mode. Then, of course, we can see that open was unsuccessful and, according to the specification, an exception information is entered into the exception which was passed as an argument. Now, in a case that open works as it is supposed to do, we can observe that the exception was caused because the file was not found.

The next test case (lines 247–302) verifies that a text can be written into a file. The write function takes two arguments besides *this* pointer; these are pointer to the buffer from which data is read and unsigned integer that specifies the maximum number of bytes to be transferred. At the beginning of the test case, as a precondition, an empty file is opened with the options that it is created and is suitable for writing. Since the file opening was tested before, open can be thought as a precondition. Then 50 characters are written into a file. Finally an external function *CompareFileToBuffer()* is used to make sure that the writing works correctly.

The following test case verifies the correctness of the file reading (lines 306–384). The read function takes the same arguments as the write function, and it returns an unsigned integer, which specifies the number of bytes transferred. A precondition of this test case is that an empty file is opened (by creating a new file) and something is written into it. Then the file is closed and reopened in a read mode. After that the actual test step is implemented; 100 characters are read from the file into a buffer (if the file contains less that 100 characters, all characters are read). Since we have earlier in preconditions written characters into the file, we know that the file contains only 71 characters and we also

know what they are. Thus, we can compare these texts, and if the result is correct, the test case passes.

The next test case verifies the seek function (lines 388–454), which permits random access to the content of a previously opened file by moving the file pointer. This function takes two parameters: long integer, which specifies the number of bytes to be moved the pointer value, and an unsigned integer that described the movement mode. In addition, the function returns the new byte offset from the beginning of the file (long int). As a precondition, an empty file is opened and some text is written into it. Then the seek function is called and, since we know the content of the file and the position to which the pointer is supposed to point, we can read characters from the file and assure that the right text was read.

The following test case assures the correctness of the duplicate function (lines 458–509), which is intended to construct a duplicate *CFile* object for a given file. This function does not take arguments (except *this* pointer), but it returns a pointer to a duplicate *CFile* object. Again, in preconditions, an empty file is opened by creating it. Then the duplicate function is called, and then the initial *CFile* object is destroyed. Then the duplicate object is used to open and close a file, just to ensure that the duplicate object exists and works correctly.

The last test case verifies a static member function *GetStatus()* (lines 513–569), which retrieves the status of the desired file. Since this function is declared static, there is exactly one copy of a static member, and thus, *this* pointer is not passed for this function. The *GetStatus()* function takes two arguments: pointer to the path of the desired file (i.e. *char**), and a reference to a *CFileStatus* structure that will receive the status information. The return value of this function is a boolean value that describes whether or not the status information is successfully obtained. As the preconditions, an empty file is created and an "empty" object of *CFileStatus* is created (i.e. C++ default values are used). Then the *GetStatus()* function is called with the appropriate parameters. If the return value is nonzero (i.e. status information is successfully received), the received status information can be verify. That is, the file size should be zero and the path to the file appropriate, and if they match, the verdict can be assigned as pass.

Finally, at the end of the *CFile_Test* module, there is a control part, in which test cases are executed one after the other. The execution continues on the condition that the verdict of previous test case is *pass*, but if some test case does not pass, execution is halted immediately

6.4. Runtime Implementation

This case study has been implemented in a workspace which includes all parts that are needed for the entire test software. This workspace consists of two projects; the main project that is called *CFileTester* and a library project called *t3cpp*. These projects include both C-files generated from the TTCN-3 code by Telelogic Tau/Tester 2.2 and TRI/TCI specific implementation files (i.e. implementation of adapters, codecs, external functions, signature calls, etc.). Figure 17 shows a snapshot of Microsoft Visual C++ development environment. The window of the left side of this figure shows a file view, in which all .c, .cpp and .h files that are needed in this program are listed.

The project *t3cpp* is linked to a library (t3cpp.lib), since it contains among others all codecs and external function implementations for the C++ type mappings. Thus, by using these functions, conversions for the data types between TTCN-3 and C++ can be accomplished. In addition, this library contains declarations for TRI and TCI interfaces and some other files that contain useful functions that are purposed to facilitate programming. The *CFileTester* project contains all those implementations that are specific for the testing of the *CFile* class.

Every .c file in Figure 17 is generated from TTCN-3 code and for each source file there is a corresponding header file (.h). TTCN-3 compiler produces unique files for each module (i.e. *CFile_Test*, *T3_CFile*, *MSDN_Typedefs* and *Cpp*) that are described earlier in Chapter in 6.3. In addition, each of these modules, except *CFile_Test*, has an adapter file (e.g. T3_CFile_Adapter.cpp) and corresponding header file (e.g. T3_CFile_Adapter.h), which include TRI/TCI specific operations. That is, adapter files contain encode and decode functions for converting the data types from TTCN-3 to C++ and vice versa, as well as implementation of signature calls and external functions that are defined within the module in question. Module *CFile_Test* does not have an adapter file, because this module does not contain any signature or external function declarations, and thus adapters need not be implemented.

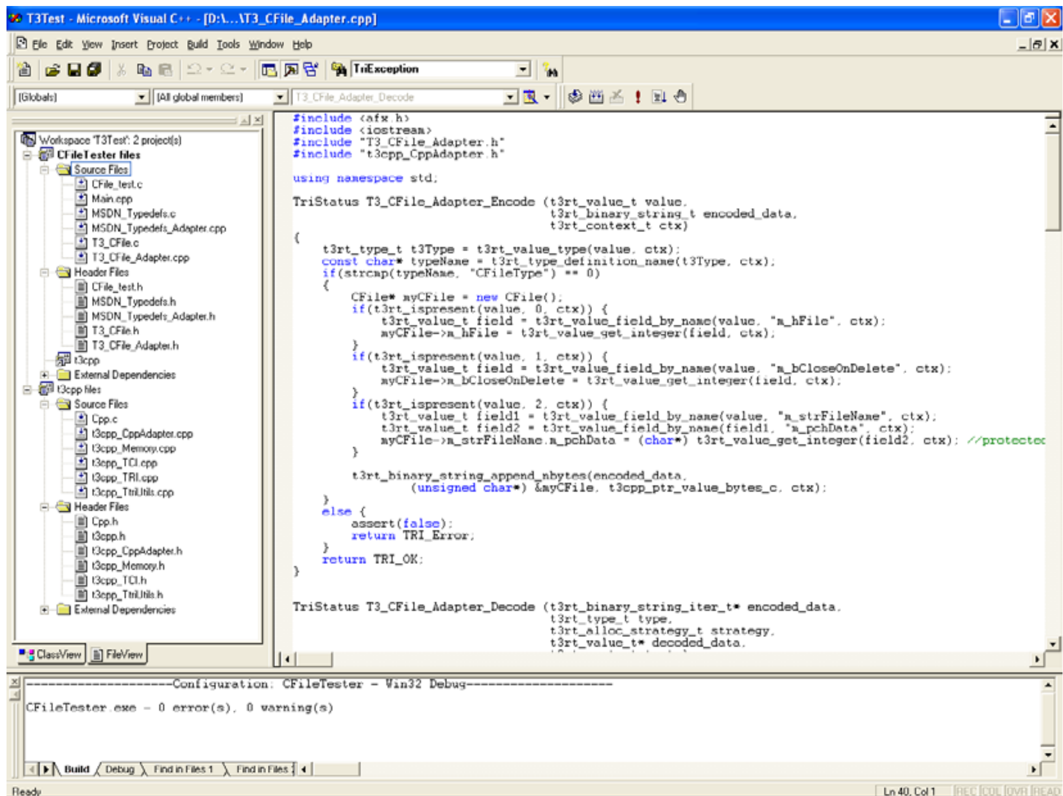


Figure 17. A snapshot of the CFile tester.

6.5. Test Runs and Results

When all of the earlier described operations have been implemented, compiled and linked, we finally get an executable file (CFileTester.exe). When executing tests Telelogic Tau/Tester provides a verbosity level of the built-in textual event log [34]. That is a value, which determines how much log information is shown. The value range is from 0 to 3 with the following meaning: 0 is off, 1 is minimal, 2 is normal and 3 is extended. Figure 18 shows a snapshot of the test run with the minimal log level, that is, only the results for all nine test cases are shown.

As it can be seen from Figure 18, all test cases pass. It means that the CFile class seems to be working as it is supposed to do. However, this test is far from full extensive testing, since in this exercise only a small portion of the CFile class was tested.

All in all, even though various different C++ data types were used in this exercise, all required data types were able to be described by TTCN-3, and thus, test date could be produced without bigger problems. All kinds of type mappings that were provided in chapters 4 and 5 were verified and they worked rather well. Thus the primary goal was achieved.

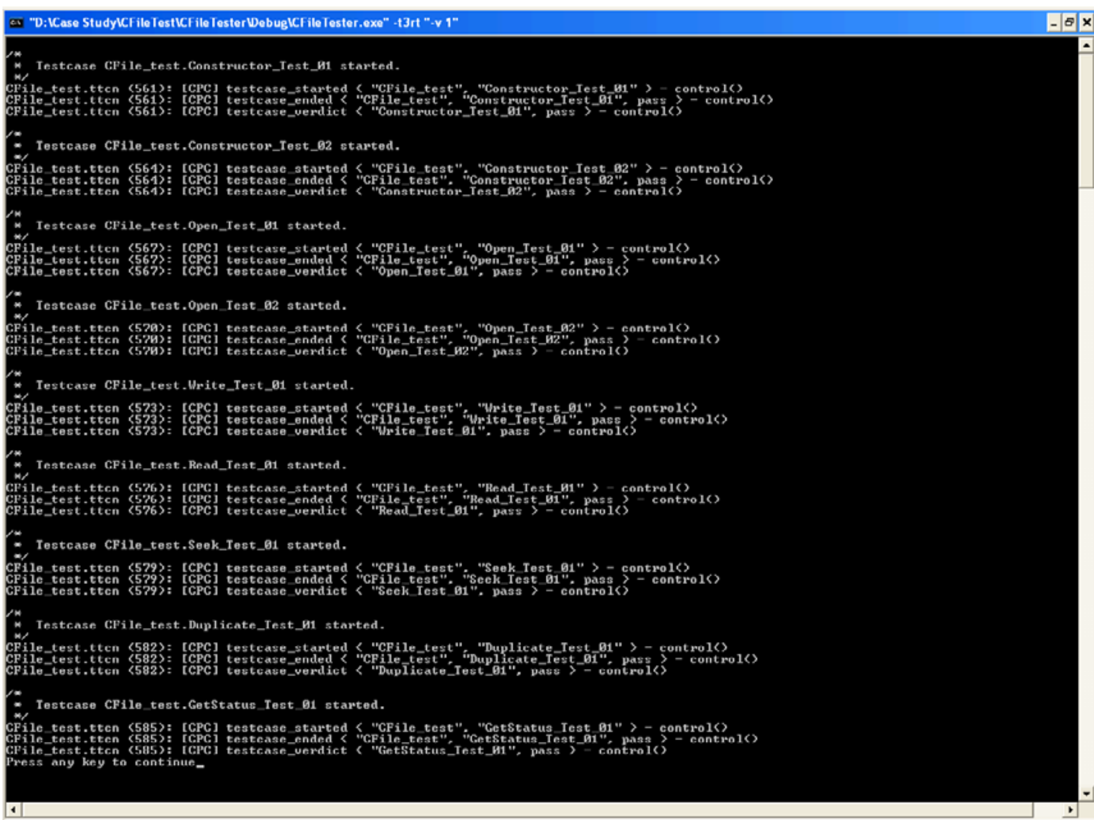


Figure 18. Test run with the minimal log level.

7. Discussion

7.1. General Evaluation

The primary purpose of this thesis was to produce data type mapping from C++ to TTCN-3 and gain experience of the usability of TTCN-3 in software module testing by testing C++ software. The first step in order to utilize TTCN-3 in testing C++ software modules is to implement the interface of the tested module at the TTCN-3 language level. That is why C++ data types need to be mapped to TTCN-3. Chapters 4 and 5 in this thesis provides type mapping rules from C++ to TTCN-3 and Chapter 6 presents a case study in which type mapping rules are utilized using TTCN-3 to test C++ software.

7.1.1. Evaluation of Type Mappings

Because of the different behavior of TTCN-3 and C++, data types also differ between these languages. The basic types are somewhat similar, but structured and built-in types differ considerably. Thus, during the work with the data type mappings, several problems were faced.

In Chapter 4 C++ fundamental types were mapped to TTCN-3. Most problems with these types were caused by the fact that C++ uses pure binary number representation for integer types (i.e. usually two's complement), whereas TTCN-3 uses a different numeration system. Thus, TTCN-3 requires type compatibility, whereas in C++ a variable is just a bit stream and its type determines what it represents. This brings about for example the following problems:

- C++ accepts an integer value to be put into a char,
- C++ integers can be assigned in the one of the literal form (e.g. decimal, octal, hexadecimal), but in TTCN-3 these are distinct types, and
- C++ accepts negative values to be placed in an unsigned integer.

The first problem could be solved either by using TTCN-3 predefined conversion functions (e.g. *int2char()*) or by using an alternative mapping, in which C++ characters are mapped to integers in TTCN-3. The solution for two other problems will be presented in section 7.2.

Another problem with the fundamental types was that C++ uses (in most cases) 8-bit characters, whereas in TTCN-3 characters are based on a 7-bit coded character set. This problem can be solved by making the decision that only ASCII [2] characters will be used. Another way is to map characters outside the ASCII range to wide characters, which is not very practical. The third way to solve this problem is to use distinct mapping, in which, as stated above, characters are mapped to integers.

On top of it all, C++ types are implementation-defined and that is why the sizes of variables (i.e. value ranges) depend on the hardware characteristics. Hence, two different development environments may have different type mappings.

During the mapping of C++ compound types into TTCN-3 in Chapter 5 some difficulties were also confronted. These troubles were caused due to the ambiguity of C++ pointers and lacking of an object model in TTCN-3. The pointers were mapped by means of external functions, and this mapping has been proved to be very efficient. C++ provides a wide range of pointers, i.e. pointer to basic types, pointer to class, pointer to function, pointer to member data, etc. By using the mapping developed in this thesis all kinds of pointers can be mapped to TTCN-3. This is a good achievement. Also, when the solution for the pointers was achieved, several other problems could be solved, such as mapping of references and arrays.

Object-oriented features bring up several benefits in C++, such as inheritance, access specifiers, member functions, etc, which complicates the mapping. This thesis does not provide solutions for all of these issues, because mapping is provided only for the data types and, for example, member functions are not handled as data types. However, these issues can be solved as well [36], and they were dealt with in the case study in Chapter 6.

All of the C++ data types that have been dealt with in chapters 4 and 5 are based on the C++ standard [30]. Thus, data type mapping provided in this thesis is comprehensive, i.e. all C++ data types are included in it. However due to the variety of differences between C++ and TTCN-3, these type mapping rules may be partly a little bit cumbersome, even though they work faultlessly.

[2] ASCII (American Standard Code for Information Interchange) is 7-bit standardized character set.

7.1.2. Evaluation of the Case Study

The purpose of the case study in Chapter 6 was to evaluate the data type mappings provided in this thesis in a real world like situation. The tested class (CFile class) is a relatively large software module that includes lots of functionality and that is why very extensive testing with small effort is somewhat impossible to achieve. However, the CFile class suits well as an example for this purpose, because it is not the simplest application and, due to its wideness, various kinds of test data need to be defined. Therefore, testing of the CFile class was intended to best reveal the weaknesses with the data type mappings.

The specifications for the CFile class can be found from the Microsoft Foundation Class library [35]. Designing of the test cases, which is the first step in the testing process, was started by looking through the specification and, at the same time, paying attention to the required inputs and outputs of member functions. Then a reasonable set of test cases was designed by focusing on the following aspects: test cases should be consecutive by supporting each other, i.e. tested function can be used as a trusted object in another test case, and the other hand, data type mappings should be utilized for all they are worth.

The case study is based on a pure functional black-box testing, without however slavishly following any special testing technique but, rather, using many of them. Probably the most common testing technique, special value testing, plays a major role in this work, but parts of the boundary value analysis and robustness testing have been utilized as well. To be more specific, a normal trend when declaring test data was to use the values that would correspond to a real world situation as well as possible. That is, as special value testing, general text files (fileName.txt) were used with different combinations of read, write and create, and some sensible text was written into the files. In addition, exception handling was tested, which is the part of robustness testing methodology. From some point of view, also boundary value analysis testing was utilized since, among others, the empty files were used, and data was read up to the end of the file.

All in all, the main goal of the testing CFile class was not to find bugs or defects, but the primary goal was to verify data type mappings. Due to the different motivation of testing compared to a usual testing event, it was not reasonable to follow any special testing methodology and that is why some kind of combination was used.

7.2. Problems and Solutions

During the construction of test cases, some unexpected problems were faced, of which some could easily be solved, but some of them needed more contemplation. The most troublesome problem was caused by the fact that when the software produced by a third party is being tested, one is not able to access private or protected member data. In order to solve this problem, there are two possibilities: to use friend functions to access protected data, or to make changes in the header file of the tested module. The latter one was used in this work. A considerate way to accomplish this is to take advantage of preprocessor options with *#ifdef* and *#endif* to conditionally compile the source code. For example, if a header file contains the following type definition:

```
private:
   int privateMemberData;
```

In order to access privateMemberData, the code should be modified as follows:

```
#ifdef T3_TEST
public:
#else
protected:
#endif
   int privateMemberData;
```

Now, when a preprocessor is provided with the definition T3_TEST, within the files in question one can access the privateMemberData, but access is prevented elsewhere.

Another problem was faced with while creating a pointer to an object of some class (for example CFileException). When we use the appropriate external functions to create a pointer to a class object, we need to give some initial values to it, even though we rather would like to use C++ default values. So, in order to solve this issue, there has been implemented an empty message template, in which all fields are omitted (by using the *omit* key word). This makes it possible to build encoding in a way that C++ default values are used. In order to be able to use *omit* values, member data fields within a *record* must be defined as *optional*.

Moreover, in C++ one can also assign negative values to unsigned integer variables. However, since unsigned values are mapped to TTCN-3 with the range, negative values are not accepted to them. This kind of problem can be solved by making the conversion manually. For example if there is the following data definition:

```
unsigned int UINT = -1;
```

According to two's complement representation, -1 corresponds to a binary value, in which all bits are 1s. Thus, the corresponding definition in TTCN-3 could be for example:

```
var CppUnsignedInt UINT := hex2int('FFFFFFFF'H);
```

Finally, TTCN-3 does not provide bit mask operations for integer variables; these operations are allowed only for binary, octal and hexadecimal variables. However, in C++, bit mask operations are commonly used, for example, when assigning the flag values (i.e. flags = modeCreate | modeWrite). In order to do the same operation in TTCN-3, we have two alternatives: to do some mental calculation, or to convert each flag value to *bitstring* (by using predefined function *int2bit*) and then performing a bit mask operation (e.g. *or4b*) with *bitstring* values and then converting the result back to an integer (by using *bit2int*).

7.3. Usability and Advantage of Type Mappings

It can be clearly seen that, during the testing of the CFile class the type mappings have been richly taken advantage of. A considerably part of the TTCN-3 test script was data type conversions, which may seem a waste of resources. One can probably think that do we really need that much code for testing such a simple program? However, these test cases have purposely been developed in a way that various different C++ types need to be converted to TTCN-3, and thus type mappings have been utilized. It is true, that a lot of effort is needed for type conversions, but once type conversions have been done, test data can be easily constructed and thus implementing of test cases is much easier and faster.

7.4. Conclusion

The C++ to TTCN-3 language mapping is essential and also a necessary part is order to efficiently utilize TTCN-3 in testing C++ software. Data type mapping is probably the most vital part of the entire language mapping. Many people would certainly be interested in knowing how C++ can be mapped to TTCN-3. That is why the result of this thesis will also be published in the form of conference papers. Currently, based on the achievement of this thesis work, two papers have been submitted to scientific conferences [3][4]. Also discussion with ETSI (European Telecommunications Standards Institute) has been engaged in prior to giving a contribution to the technical specification for the C++ to TTCN-3 mapping, which will be part of the TTCN-3 standard.

A clear strength of the case study was that the main goal which was to utilize data type mappings was mainly achieved. Even though C++ supports a wide spectrum of data types and some data types are very ambiguous, the testing process was run through without major problems. A particularly positive point was that mapping of pointers worked very well, even though the preconception was that pointers were going to be one of the biggest problems.

One observed weakness was that implementing the run time behavior proved to be quite laborious. Particularly encoding and decoding required a fairly large number of code lines, and memory management also brought additional difficulties. Consequently, it is reasonable to consider whether it is worth using TTCN-3 in testing C++ software or not. In conclusion it can be said that more work around TTCN-3 is still needed and, for example, some kind of tools or generators to facilitate the implementation of an executable test system will be required. However, in the long run, TTCN-3 may become a respectable alternative in large scale testing of C++ software.

Future work will be to widen C++ data type mapping to contain the entire C++ to TTCN-3 language mapping, including such issues as lexical conventions, names and scoping, preprocessing, and as mentioned earlier, mapping of member functions. These issues are under work at the moment. After the complete language mapping has been implemented, the next stage could be to provide a tool that would automatically generate mappings as well as codecs and adapters. When exact mapping rules are available, these can be constructed pretty straightforwardly, which is good work for the machine. If this kind of generator can be implemented, the test personnel need implement only the test cases and others will be generated automatically. This is a mature challenge but this is a good objective towards which this work can continue.

8. Summary

The primary purpose of this thesis was to develop methods for the standardized test specification language called TTCN-3 (Testing and Test Control Notation) in a way that TTCN-3 would be better utilized for testing software implemented with C++. The first step prior to achieving efficient use of TTCN-3, the interface of the tested C++ software module should be represented at the TTCN-3 language level. That is, C++ data types must be mapped to TTCN-3 in order to construct test data for the test cases. Thus, the research problem for this thesis was to find means how C++ data types can be represented at the TTCN-3 language level. These mapping rules were aimed to be put into practice by testing real C++ software with TTCN-3.

First, the motivation about the usefulness of C++ data type mappings was given and potential problems and challenges were presented. Then basic principles of the software testing process were discussed and general software testing techniques were also presented. Based on the standards and earlier research activity, descriptions of the TTCN-3 test system and standardized execution interfaces, as well as presentation formats were provided. Also earlier research experience about the TTCN-3 with other languages (ASN.1, IDL and XML) was presented and the speculation on use of TTCN-3 for C++ software testing was brought out.

The actual work, which was C++ data type mappings to TTCN-3, was divided into two groups; mapping of C++ fundamental types was achieved without a huge struggle but mapping of C++ compound types needed more study and thinking. However, after hard trying through trial and error somewhat uniform data type mapping rules were achieved. Probably the most challenging part of the mapping was to bring in the feature of pointers in all of their variety. After sufficient solution of the mapping of pointers, also mapping of arrays and references could easily be provided. User-defined types, such as classes and unions, due to their object oriented features, caused some inconvenience, but finally a satisfactory solution was achieved. Since all of the C++ data types could be mapped to TTCN-3, the primary goal was obtained well.

As a case study, the entire TTCN-3 based test system including type mappings, adapters and codecs was constructed. The purpose of this testing was to verify data type mappings in a real world-like situation and gain experience of the usability of TTCN-3 in testing C++ software. Within the nine test cases, all test data were created according to data type mapping rules that were provided earlier in this thesis, and so fairly extensive sampling

was accomplished. However, some problems appeared during the case study, but all problems could be solved one way or another. In addition, use of TTCN-3 in testing C++ software proved to be rather laborious, but workable, and after some development it is believed to be a respectable alternative for testing C++ software.

References

[1] McGregor J.D. & Sykes D.A. (2001) A Practical Guide to Testing Object-Oriented Software. Upper Saddle River, NJ, USA: Addison Wesley. 393 p. ISBN 0-201-32564-0.

[2] Grabowski J., Hogrefe D., Réthy G., Schieferdecker I., Wiles A. & Willcock C. (2003) An introduction to the testing and test control notation (TTCN-3). In: Computer Networks: The International Journal of Computer and Telecommunications Networking, June, New York, NY, USA, Vol. 42, issue 3, pp. 375–403.

[3] Kärki M. & Pulkkinen P. (2004) Representing C++ Pointers in a Test Implementation Language. Submitted to: The 19th IEEE International Conference on Automated Software Engineering (ASE 2004), September 20–25, Linz, Austria.

[4] Kärki M., Pulkkinen P. & Sihvonen M. (2004) C++ to TTCN-3 Mapping Challenges. Submitted to: Fourth International Conference of Quality Software, September 8–10, Braunschweig, Germany.

[5] Jorgensen P.C. (1995) Software Testing: A Craftsman's Approach. Boca Raton, FL, USA: CRC Press. 254 p. ISBN 0-8493-7345-X.

[6] Binder R.V. (1999) Testing Object-Oriented Systems: Models Patterns, and Tools. Reading, MA, USA: Addison Wesley. 1191 p. ISBN 0-201-80938-9.

[7] Fewster M. & Graham D. (1999) Software Test Automation, Effective use of test execution tools. London, UK: ACM Press. 574 p. ISBN 0-201-33140-3.

[8] Beizer B. (1990) Software Testing Techniques. New York, NY, USA: Van Nostrand Reinhold. 550 p. ISBN 0-442-20672-0.

[9] Patton R. (2001) Software Testing. Indianapolis, IN, USA: Sams Publishing. 389 p. ISBN 0-672-31983-7.

[10] ETSI ES 201 873-1 (2003) Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. Sophia Antipolis Cedex, France: European Telecommunications Standard Institute. 178 p.

[11] ETSI ES 201 873-2 (2003) Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular presentation Format (TFT). Sophia Antipolis Cedex, France: European Telecommunications Standard Institute. 33 p.

[12] ETSI ES 201 873-3 (2003) Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT). Sophia Antipolis Cedex, France: European Telecommunications Standard Institute. 160 p.

[13] ETSI ES 201 873-4 (2003) Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics. Sophia Antipolis Cedex, France: European Telecommunications Standard Institute. 138 p.

[14] ETSI ES 201 873-5 (2003) Methods for Testing and Specification (MTS); The Testing and Test Control Notation Version 3; Part 5: TTCN-3 Runtime Interface (TRI). Sophia Antipolis Cedex, France: European Telecommunications Standard Institute. 54 p.

[15] ETSI ES 201 873-6 (2003) Methods for Testing and Specification (MTS); The Testing and Test Control Notation Version 3 (TTCN-3); Part 6: TTCN-3 Control Interfaces. Sophia Antipolis Cedex, France: European Telecommunications Standard Institute. 75 p.

[16] ITU-T Z.140 (2003) The testing and test control notation version 3: TTCN-3 core language. Geneva, Switzerland: International Telecommunication Union (ITU). 178 p.

[17] ITU-T Z.141 (2003) Testing and test control notation version 3 (TTCN-3): Tabular presentation format. Geneva, Switzerland: International Telecommunication Union (ITU). 30 p.

[18] ITU-T Z.142 (2003) Testing and test control notation version 3: Graphical Presentation Format for TTCN-3 (GFT). Geneva, Switzerland: International Telecommunication Union (ITU). 152 p.

[19] Schulz S. & Vassiliou-Gioles T. (2002) Implementation of TTCN-3 Test System using the TRI. In: IFIP 14th International Conference on Testing of Communicating Systems – TestCom 2002, March 19–22, Berlin, Germany. Pp. 425–442.

[20] Schieferdecker I. & Vassiliou-Gioles T. (2003) Realizing distributed TTCN–3 test systems with TCI. In: 15th IFIP International Conference, TestCom 2003, May 26–28, Sophia Antipolis, France. Pp. 95–109.

[21] Willcock C. (2001) The TTCN-3 Runtime Interface (TRI), Concepts and Interface Definition of the TRI. In: European Telecommunications Standards Institute, Methods for Testing and Specification (MTS) #32, March 14–15, Sophia-Antipolis, France.

[22] Ebner M., Yin A. & Li M. (2002) Definition and Utilisation of OMG IDL to TTCN-3 Mappings. In: IFIP 14th International Conference on Testing Communicating Systems – TestCom 2002, March 19–22, 2002. Berlin, Germany. Pp. 443–459.

[23] Ebner M. (2001) A Mapping of OMG IDL to TTCN-3. SIIM Technical Report SIIM-TR-A-01-11. Institute for Telematics, Medical University of Lübeck, Schriftenreihe der Institute für Informatik / Mathematik, Lübeck, Germany. 39 p.

[24] Ebner M. (2002) Mapping CORBA IDL to TTCN-3 based on IDL to TTCN-2 mappings. In: Proceedings of the 11th GI/ITG Technical Meeting on Formal Description Techniques for Distributed Systems, June 21–22, Bruchsal, Germany.

[25] Schieferdecker I. & Stepien B. (2003) Automated Testing of XML/SOAP based Web Services. In: 13. Fachkonferenz der Gesellschaft für Informatik (GI) Fachgruppe "Kommunikation in verteilten Systemen" (KiVS), February 26–28, Leipzing, Germany.

[26] ETSI TS 102 219 (2003) Methods for Testing and Specification (MTS); The IDL to TTCN-3 Mapping. Sophia Antipolis Cedex, France: European Telecommunications Standard Institute. 27 p.

[27] ITU-T X.680 (2002) Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation. Geneva, Switzerland: International Telecommunication Union (ITU). 133 p.

[28] World Wide Web Consortium (23.3.2004). Extensible Markup Language, URL: http://www.w3.org/XML/

[29] The TTCN-3 User Conference, May 3–5 2004, Sophia-Antipolis, France. URL: http://www.ttcn-3.org

[30] ISO/IEC 14882 (1998) Programming languages – C++. New York: American National Standards Institute (ANSI). 776 p.

[31] Stroustrup B. (2000) The C++ Programming Language, Special Edition. Florham Park, NJ, USA: Addison-Wesley. 1019 p. ISBN 0-201-70073-5.

[32] ISO/IEC 646 (1991) Information technology - ISO 7-bit coded character set for information interchange. Geneva, Switzerland: International Organization for Standardization (ISO). 15 p.

[33] Standard ECMA-128 (1999) 8-Bit Single-Byte Coded Graphic Character Sets: Latin Alphabet No. 5: Geneva, Switzerland: ECMA Standardizing Information and Communication Systems. 17 p.

[34] Telelogic Tau 2.2 User Guide (2003) Telelogic AB, Malmö, Sweden. 1366 p.

[35] Microsoft Visual C++ (23.3.2004) Microsoft Foundation Class Library. URL:http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/mfchm.asp

[36] Kärki M., Pulkkinen P. & Hämäläinen J. (2004) TTCN-3 Test Architecture for Functional Testing of SW Modules (draft). VTT Technical Report in a project: Test & Testing Methodologies with Advanced Languages (TT-Medal-VTT), Oulu, Finland. 43 p.

APPENDIX 1   TTCN-3 Test Script: Mapping of C++ Basic Data Types and Pointers

```
/***************************************************************************************/
/*                                                                                     */
/*                                      Cpp                                            */
/*                                                                                     */
/*      This module contains all type definitions for the mapping of C++ basic         */
/*      types to TTCN-3.                                                               */
/*                                                                                     */
/***************************************************************************************/

module Cpp {

/////////////////////////////////////////////////////////////////////////////////////////
// C++ data type definitions
//
    type boolean CppBool;
    type integer CppShort(-32768..32767);
    type integer CppUnsignedShort(0..65535);
    type integer CppInt(-2147483648..2147483647);
    type integer CppUnsignedInt(0..4294967295);
    type integer CppLong(-2147483648..2147483647);
    type integer CppUnsignedLong(0..4294967295);
    type float CppFloat (-3.402823466E38 .. 3.402823466E38);
    type float CppDouble (-1.7976931348623158E308 .. 1.7976931348623158E308);
    type float CppLongDouble (-1.7976931348623158E308 .. 1.7976931348623158E308);
    type char CppChar;
    type char CppUnsignedChar;
    type char CppSignedChar;
    type universal char CppWchar_t;
//
/////////////////////////////////////////////////////////////////////////////////////////


/////////////////////////////////////////////////////////////////////////////////////////
// C++ Pointer mappings
//
    // type definition for the pointer
    type integer CppPtr;

    // external functions for double pointer
    type record of CppPtr CppPtrList;
    external function New_CppPtr(in CppPtrList aList) return CppPtr;
    external function Get_CppPtr(in CppPtr ptr, in CppInt index) return CppPtr;
    external function Del_CppPtr(in CppPtr ptr);

    // external functions for char
    type record of CppChar CppCharList;
    external function New_CppChar(in CppCharList aList) return CppPtr;
    external function Get_CppChar(in CppPtr ptr, in integer index) return CppChar;
    external function Set_CppChar(in CppPtr ptr, in integer index, in CppChar aValue);
    external function Del_CppChar(in CppPtr ptr);

    // external functions for char*.
    type charstring CppString;
    external function New_CppString(in CppString string) return CppPtr;
    external function New_CppString_with_size(in CppString string, in integer size)
        return CppPtr;
    external function Get_CppString(in CppPtr ptr) return CppString;
    external function Set_CppString(in CppPtr ptr, in integer index, in CppChar aValue);
    external function Del_CppString(in CppPtr ptr);

    // external functions for CppInt
    type record of CppInt CppIntList;
    external function New_CppInt(in CppIntList aList) return CppPtr;
    external function Get_CppInt(in CppPtr ptr, in integer index) return CppInt;
    external function Set_CppInt(in CppPtr ptr, in integer index, in CppInt aValue);
    external function Del_CppInt(in CppPtr ptr);

    // external functions for CppDouble
    type record of CppDouble CppDoubleList;
    external function New_CppDouble(in CppDoubleList aList) return CppPtr;
    external function Get_CppDouble(in CppPtr ptr, in integer index) return CppDouble;
    external function Set_CppDouble (
        in CppPtr ptr, in integer index, in CppDouble aValue);
    external function Del_CppDouble(in CppPtr ptr);
//
/////////////////////////////////////////////////////////////////////////////////////////
} // end of module Cpp
```

```
1  /*********************************************************************************/
2  /*                                                                               */
3  /*                                  T3_CFile                                     */
4  /*                                                                               */
5  /*      This module contains all type mappings for the CFile class.              */
6  /*                                                                               */
7  /*********************************************************************************/
8
9  module T3_CFile {
10     import from Cpp all;
11     import from MSDN_Typedefs all;
12
13     type component CFileTSI {
14         port CFilePort aCFilePort
15     }
16
17     type port CFilePort procedure {
18         out CFile_no_parameters,
19         CFile_one_parameter,
20         CFile_two_parameters,
21         Open,
22         GetStatus,
23         Duplicate,
24         Seek,
25         Read,
26         Write,
27         Close,
28         D_CFile
29     }
30
31
32 ///////////////////////////////////////////////////////////////////////////////////
33 // External function definitions
34 //
35     external function New_CFile(in CFileType aCFile) return CppPtr;
36     external function Get_CFile(in CppPtr this) return CFileType;
37     external function Del_CFile(in CppPtr this);
38 //
39 ///////////////////////////////////////////////////////////////////////////////////
40
41
42
43 ///////////////////////////////////////////////////////////////////////////////////
44 // CFile mapping (class CFile: public CObject)
45 //
46     type record CFileType {
47         UINT m_hFile optional,
48         BOOL m_bCloseOnDelete optional,
49         CStringType m_strFileName optional
50     }
51
52     template CFileType Empty_CFile :=
53     {
54         m_hFile := omit,
55         m_bCloseOnDelete := omit,
56         m_strFileName := omit
57     }
58
59     // Flag values...
60
61     // OpenFlag values
62     const CppInt modeRead        := hex2int('0000'H);
63     const CppInt modeWrite       := hex2int('0001'H);
64     const CppInt modeReadWrite   := hex2int('0002'H);
65     const CppInt shareCompat     := hex2int('0000'H);
66     const CppInt shareExclusive  := hex2int('0010'H);
67     const CppInt shareDenyWrite  := hex2int('0020'H);
68     const CppInt shareDenyRead   := hex2int('0030'H);
69     const CppInt shareDenyNone   := hex2int('0040'H);
70     const CppInt modeNoInherit   := hex2int('0080'H);
71     const CppInt modeCreate      := hex2int('1000'H);
72     const CppInt modeNoTruncate  := hex2int('2000'H);
73     const CppInt typeText        := hex2int('4000'H);    // typeText and typeBinary are
74     const CppInt typeBinary      := hex2int('8000'H);    // used in derived classes only
75
76     // Attribute values
77     const CppInt normal      := hex2int('0000'H);
78     const CppInt readOnly    := hex2int('0001'H);
```

```
 79     const CppInt hidden     := hex2int('0002'H);
 80     const CppInt system_    := hex2int('0004'H);  // system is a reserved word in TTCN-3
 81     const CppInt volume     := hex2int('0008'H);
 82     const CppInt directory  := hex2int('0010'H);
 83     const CppInt archive    := hex2int('0020'H);
 84
 85     // SeekPosition values
 86     const CppInt begin      := hex2int('0000'H);
 87     const CppInt current    := hex2int('0001'H);
 88     const CppInt end        := hex2int('0002'H);
 89
 90     const UINT hFileNull := hex2int('FFFFFFFF'H);  // correspongd to -1 for signed int
 91
 92     // Constructors
 93     signature CFile_no_parameters() return CppPtr;
 94     signature CFile_one_parameter(in CppInt hFile) return CppPtr;
 95     signature CFile_two_parameters(in LPCTSTR lpczFileName, in UINT nOpenFlags)
 96             return CppPtr;
 97
 98     // Operations
 99     signature Open(in CppPtr this, in LPCTSTR lpszFileName, in UINT nOpenFlags,
100             in CppPtr pError) return BOOL;
101     signature GetStatus(in LPCTSTR lpszFileName, in CppPtr rStatus) return BOOL;
102
103     // Overridables
104     signature Duplicate(in CppPtr this) return CppPtr;
105     signature Seek(in CppPtr this, in LONG lOff, in UINT nFrom) return LONG;
106     signature Read(in CppPtr this, in CppPtr lpBuf, in UINT nCount) return UINT;
107     signature Write(in CppPtr this, in CppPtr lpBuf, in UINT nCount);
108     signature Close(in CppPtr this);
109
110     // Implementation
111     signature D_CFile(in CppPtr this);  // Destructor
112 //
113 ///////////////////////////////////////////////////////////////////////////////////////
114
115 } // end of module T3_CFile
```

```
1  /*****************************************************************************************/
2  /*                                                                                       */
3  /*                                 MSDN_Typedefs                                         */
4  /*                                                                                       */
5  /*      This module contains type definitions and external functions that are           */
6  /*      needed for the testing of CFile class                                            */
7  /*                                                                                       */
8  /*****************************************************************************************/
9
10 module MSDN_Typedefs {
11     import from Cpp all;
12
13 ////////////////////////////////////////////////////////////////////////////////////////////
14 // MSDN Type definitions
15 //
16     // A 32-bit pointer to a constant character string.
17     type CppPtr LPCTSTR;
18     // A 16-bit unsigned integer on Windows versions 3.0 and 3.1
19     // and a 32-bit unsigned integer on Win32.
20     type CppUnsignedInt UINT;
21     // A Boolean value.
22     type CppInt BOOL;
23     // A 32-bit signed integer.
24     type CppInt LONG;
25     // A 32-bit pointer to a character string.
26     type CppPtr LPTSTR;
27     // An 8-bit value
28     type CppUnsignedChar BYTE;
29 //
30 ////////////////////////////////////////////////////////////////////////////////////////////
31
32
33 ////////////////////////////////////////////////////////////////////////////////////////////
34 // CFileException mapping (class CFileException : public CException)
35 //
36     // external function definitions
37     external function New_CFileException(in CFileExceptionType aVar) return CppPtr;
38     external function Get_CFileException(in CppPtr ptr) return CFileExceptionType;
39     external function Del_CFileException(in CppPtr ptr);
40
41     type record CFileExceptionType {
42         CExceptionType aCException optional,
43         CppInt m_cause optional,             // portable code corresponding to the exception
44         LONG m_lOsError optional,            // the related operating-system error number.
45         CStringType m_strFileName optional  // the name of the file for this exception.
46     }
47
48     const CppInt none_ := 0;
49     const CppInt generic := 1;
50     const CppInt fileNotFound := 2;
51     const CppInt badPath := 3;
52     const CppInt tooManyOpenFiles := 4;
53     const CppInt accessDenied := 5;
54     const CppInt invalidFile := 6;
55     const CppInt removeCurrentDir := 7;
56     const CppInt directoryFull := 8;
57     const CppInt badSeek := 9;
58     const CppInt hardIO := 10;
59     const CppInt sharingViolation := 11;
60     const CppInt lockViolation := 12;
61     const CppInt diskFull := 13;
62     const CppInt endOfFile := 14;
63
64     template CFileExceptionType Empty_CFileException := {
65         aCException := omit,
66         m_cause := omit,
67         m_lOsError := omit,
68         m_strFileName := omit
69     }
70 //
71 ////////////////////////////////////////////////////////////////////////////////////////////
72
73
74 ////////////////////////////////////////////////////////////////////////////////////////////
75 // CException mapping (class CException : public CObject)
76 // Note: COject does not have any data fields
77 //
78     type record CExceptionType {
```

```
79          BOOL m_bAutoDelete,
80          BOOL m_bReadyForDelete
81      }
82  //
83  ////////////////////////////////////////////////////////////////////////////////////////
84  
85  
86  ////////////////////////////////////////////////////////////////////////////////////////
87  // CString mapping
88  //
89      type record CStringType {
90          LPTSTR m_pchData        // pointer to ref counted string data
91      }
92  //
93  ////////////////////////////////////////////////////////////////////////////////////////
94  
95  
96  ////////////////////////////////////////////////////////////////////////////////////////
97  // File status
98  //
99      // external function definitions
100     external function New_CFileStatus (in CFileStatusType aVar) return CppPtr;
101     external function Get_CFileStatus (in CppPtr ptr) return CFileStatusType;
102     external function Del_CFileStatus (in CppPtr ptr);
103 
104     type record CFileStatusType {
105         CTimeType m_ctime optional,     // creation date/time of file
106         CTimeType m_mtime optional,     // last modification date/time of file
107         CTimeType m_atime optional,     // last access date/time of file
108         LONG m_size optional,           // logical size of file in bytes
109         BYTE m_attribute optional,      // logical OR of CFile::Attribute enum values
110         BYTE m_padding optional,        // pad the structure to a WORD
111         CppPtr m_szFullName optional    // absolute path name
112     }
113 
114     template CFileStatusType Empty_CFileStatus := {
115         m_ctime := omit,
116         m_mtime := omit,
117         m_atime := omit,
118         m_size := omit,
119         m_attribute := omit,
120         m_padding := omit,
121         m_szFullName := omit
122     }
123 //
124 ////////////////////////////////////////////////////////////////////////////////////////
125 
126 
127 ////////////////////////////////////////////////////////////////////////////////////////
128 // Mapping of the class CTime
129 //
130     type record CTimeType {
131         time_t m_time
132     }
133 
134     type CppLong time_t;        // time value
135 //
136 ////////////////////////////////////////////////////////////////////////////////////////
137 
138 
139 ////////////////////////////////////////////////////////////////////////////////////////
140 // External functions needed for the CFile mapping
141 //
142     // for chekcing if given file exists
143     external function FileExists(in CppPtr fileName) return boolean;
144 
145     // Compares the contents of a given file and buffer
146     external function CompareFileToBuffer(in CppPtr fileName, in CppPtr lpBuf)
147             return boolean;
148 //
149 ////////////////////////////////////////////////////////////////////////////////////////
150 
151 } // end of module MSDN_Typedefs
```

```
1  /***************************************************************************************/
2  /*                                                                                     */
3  /*                              CFile_test                                             */
4  /*                                                                                     */
5  /*      This module includes alltogetger 9 test cases, which are verifying the         */
6  /*      operaiton of CFile class.                                                      */
7  /*                                                                                     */
8  /***************************************************************************************/
9
10 module CFile_test {
11     import from Cpp all;
12     import from T3_CFile all;
13     import from MSDN_Typedefs all;
14
15     type component Tester {
16         port CFilePort aCFilePort
17     }
18
19 ////////////////////////////////////////////////////////////////////////////////////////
20 // Common functions, altstep used in test cases
21 //
22     function InitTestCase() runs on Tester {
23         map(mtc:aCFilePort, system:aCFilePort);
24         activate(DefaultAltSet());
25     }
26
27     function TestCaseFail() runs on Tester {
28         setverdict(fail);
29         unmap(mtc:aCFilePort, system:aCFilePort);
30     }
31
32     function StopIfFail() runs on Tester {
33         if(getverdict == fail) {
34             deactivate; // just in case
35             stop;
36         }
37     }
38
39     function TestCasePass() runs on Tester {
40         setverdict(pass);
41         unmap(mtc:aCFilePort, system:aCFilePort);
42         deactivate; // just in case
43         stop;
44     }
45
46     altstep DefaultAltSet() runs on Tester {
47         [] any port.getreply {
48             TestCaseFail();
49         }
50         [] any port.catch {
51             TestCaseFail();
52         }
53     }
54
55     // Constructor w/o parameters
56     function Constructor() runs on Tester return CppPtr {
57         var CppPtr ptr;
58         aCFilePort.call(CFile_no_parameters:{}, nowait);
59         alt {
60             [] aCFilePort.getreply(CFile_no_parameters:{} value ?) -> value ptr {
61                 var CFileType myCFile := Get_CFile(ptr);
62                 if (myCFile.m_hFile != hFileNull) {
63                     TestCaseFail();
64                 }
65             }
66         }
67         StopIfFail();
68         return ptr;
69     }
70
71     // Destructor
72     function Destructor(CppPtr ptr) runs on Tester {
73         aCFilePort.call(D_CFile:{ptr}, nowait);
74         alt {
75             [] aCFilePort.getreply(D_CFile:{-}) {}
76         }
77         StopIfFail();
78     }
```

```
79  //
80  ///////////////////////////////////////////////////////////////////////////////////
81
82
83  /*********************************************************************************/
84  /*                                                                               */
85  /*                                   TESTCASES                                   */
86  /*                                                                               */
87  /*********************************************************************************/
88
89
90  ///////////////////////////////////////////////////////////////////////////////////
91  // This test case verifies correctness of the CFile constructor and destructor
92  //
93      testcase Constructor_Test_01() runs on Tester system CFileTSI {
94          InitTestCase();
95          var CppPtr ptr := Constructor();    // call constructor
96          Destructor(ptr);                    // call destructor
97          TestCasePass();
98      }
99  //
100 ///////////////////////////////////////////////////////////////////////////////////
101
102
103
104 ///////////////////////////////////////////////////////////////////////////////////
105 // This test case verifies correctness of the CFile constructor with parameters.
106 // The constructor with two arguments creates a CFile object and opens the
107 // corresponding operating-system file with the given path.
108 //
109     testcase Constructor_Test_02() runs on Tester system CFileTSI {
110         InitTestCase();
111         var CppPtr lpszFileName := New_CppString("foo3.txt");
112         var UINT nOpenFlags := modeCreate;
113         var CppPtr ptrCFile;
114         aCFilePort.call(CFile_two_parameters:{lpszFileName, nOpenFlags}, nowait);
115         alt {
116             [] aCFilePort.getreply(CFile_two_parameters:{-,-} value ?)
117                     -> value ptrCFile {
118                 // to make sure that the file exists
119                 if (FileExists(lpszFileName) == false) {
120                     TestCaseFail();
121                 }
122             }
123         }
124         StopIfFail();
125         Del_CppString(lpszFileName);
126         Destructor(ptrCFile);   // call destructor
127         TestCasePass();
128     }
129 //
130 ///////////////////////////////////////////////////////////////////////////////////
131
132
133
134 ///////////////////////////////////////////////////////////////////////////////////
135 // File is opened for reading and writing. Function returns nonzero if the
136 // open was successful; otherwise 0. The pError parameter is meaningful
137 // only if 0 is returned.¨
138 //
139     testcase Open_Test_01() runs on Tester system CFileTSI {
140         InitTestCase();
141
142         // Parameter 1: this pointer (=pointer to CFile class)
143         var CppPtr thisPtr := Constructor();    // call constructor
144
145         // Parameter 2: pointer to filename
146         var CppPtr lpszFileName := New_CppString("foo.txt");
147
148         // Parameter 3: OpenFlags (nOpenFlags = CFile::modeCreate | CFile::modeWrite)
149         var bitstring bin_modeCreate := int2bit(modeCreate, 32);
150         var bitstring bin_modeReadWrite := int2bit(modeReadWrite, 32);
151         var bitstring bin_nOpenFlags := bin_modeCreate or4b bin_modeReadWrite;
152         var UINT nOpenFlags := bit2int(bin_nOpenFlags);
153
154         // Parameter 4: pointer to CFileException
155         var CFileExceptionType myException := valueof(Empty_CFileException);
156         var CppPtr pError := New_CFileException(myException);
```

```
157
158         // return value
159         var BOOL retValue;
160
161         // call open
162         aCFilePort.call(Open:{thisPtr, lpszFileName, nOpenFlags, pError}, nowait);
163         alt {
164             [] aCFilePort.getreply(Open:{-,-,-,-} value ?) -> value retValue { }
165         }
166         if (retValue == 0) {  // Function returns nonzero if the open was successful
167             TestCaseFail();
168         }
169         else {
170             // close earlier opened file
171             aCFilePort.call(Close:{thisPtr}, nowait);
172             alt {
173                 [] aCFilePort.getreply(Close:{-}) { }
174             }
175         }
176         StopIfFail();
177
178         // to make sure that the file exists
179         if (FileExists(lpszFileName) == false) {
180             TestCaseFail();
181         }
182         StopIfFail();
183
184         // garbage collection...
185         Del_CppString(lpszFileName);
186         Del_CFileException(pError);
187         Destructor(thisPtr);
188         TestCasePass();
189     }
190 //
191 ////////////////////////////////////////////////////////////////////////////////////////
192
193
194
195 ////////////////////////////////////////////////////////////////////////////////////////
196 // File is tried to open but open is unsuccesful. The file does not exsist
197 // and fileNotFound -exception is received.
198 //
199     testcase Open_Test_02() runs on Tester system CFileTSI {
200         InitTestCase();
201         var CppPtr thisPtr := Constructor();
202         var CppPtr lpszFileName := New_CppString("foo2.txt");
203
204         // to make sure that the file does not exist
205         if (FileExists(lpszFileName) == false)
206         {
207             var UINT nOpenFlags := modeReadWrite;
208             var CFileExceptionType myException := valueof(Empty_CFileException);
209             var CppPtr pError := New_CFileException(myException);
210             var BOOL retValue;
211             // call open
212             aCFilePort.call(Open:{thisPtr, lpszFileName, nOpenFlags, pError}, nowait);
213             alt {
214                 [] aCFilePort.getreply(Open:{-,-,-,-} value ?) -> value retValue {
215                     // Function returns nonzero if the open was successful
216                     if (retValue != 0) {
217                         // Close the file
218                         aCFilePort.call(Close:{thisPtr}, nowait);
219                         alt {
220                             [] aCFilePort.getreply(Close:{-}) { }
221                         }
222                         TestCaseFail();
223                     }
224                     StopIfFail();
225                 }
226             }
227             myException := Get_CFileException(pError);
228             if (myException.m_cause != fileNotFound) {
229                 TestCaseFail();
230             }
231             StopIfFail();
232             Del_CFileException(pError);
233         }
234         else {
```

```
235             TestCaseFail();
236         }
237         StopIfFail();
238         Del_CppString(lpszFileName);
239         Destructor(thisPtr);
240         TestCasePass();
241     }
242 //
243 ////////////////////////////////////////////////////////////////////////////////
244 
245 
246 
247 ////////////////////////////////////////////////////////////////////////////////
248 // Test case for the CFile.Write()
249 //
250     testcase Write_Test_01() runs on Tester system CFileTSI {
251         InitTestCase();
252 
253         // Precondition: Open an empty file that is suitable for writing
254         var CppPtr this := Constructor();
255         var CppPtr fileName := New_CppString("writeTest.txt");
256         // modeCreate directs the constructor to create a new file. If the file exists
257         // already, it is truncated to 0 length.
258         var bitstring bin_modeCreate := int2bit(modeCreate, 32);
259         // modeWrite Opens the file for writing.
260         var bitstring bin_modeWrite := int2bit(modeWrite, 32);
261         var bitstring bin_openFlags := bin_modeCreate or4b bin_modeWrite;
262         // OpenFlags (nOpenFlags = CFile::modeCreate | CFile::modeWrite)
263         var UINT openFlags := bit2int(bin_openFlags);
264         var CFileExceptionType myException := valueof(Empty_CFileException);
265         var CppPtr pError := New_CFileException(myException);
266         var BOOL retValue;
267         aCFilePort.call(Open:{this, fileName, openFlags, pError}, nowait);
268         alt {
269             [] aCFilePort.getreply(Open:{-,-,-,-} value ?) -> value retValue { }
270         }
271         if (retValue == 0) {    // Function returns nonzero if the open was successful
272             TestCaseFail();
273         }
274         StopIfFail();
275 
276         // Call write...
277         var CppPtr lpBuf :=     // 50 characters
278             New_CppString("This text will be placed into a file writeTest.txt");
279         aCFilePort.call(Write:{this, lpBuf, 50}, nowait);
280         alt {
281             [] aCFilePort.getreply(Write:{-,-,-}) { }
282         }
283 
284         // Close the file
285         aCFilePort.call(Close:{this}, nowait);
286         alt {
287             [] aCFilePort.getreply(Close:{-}) { }
288         }
289 
290         // Check if the text is in the file
291         if (CompareFileToBuffer(fileName, lpBuf) == false) {
292             TestCaseFail();
293         }
294         StopIfFail();
295         Del_CppString(lpBuf);
296         Del_CFileException(pError);
297         Del_CppString(fileName);
298         Destructor(this);   // call destructor
299         TestCasePass();
300     }
301 //
302 ////////////////////////////////////////////////////////////////////////////////
303 
304 
305 
306 ////////////////////////////////////////////////////////////////////////////////
307 // Test case for the CFile.Read()
308 // At first a file is opened and some text is written into it and then the
309 // file is closed and reopened in a read-mode. After that 'Read' member
310 // funciton is tested by reading all characters from the file into a buffer.
311 //
312     testcase Read_Test_01() runs on Tester system CFileTSI {
```

```
313         InitTestCase();
314 
315         // Precondition: Open an empty file and write some text into it
316         var CppPtr this := Constructor();
317         var CppPtr fileName := New_CppString("readTest.txt");
318         // modeCreate directs the constructor to create a new file.
319         // If the file exists already, it is truncated to 0 length.
320         var bitstring bin_modeCreate := int2bit(modeCreate, 32);
321         // modeReadWrite Opens the file for reading and writing.
322         var bitstring bin_modeReadWrite := int2bit(modeReadWrite, 32);
323         var bitstring bin_openFlags := bin_modeCreate or4b bin_modeReadWrite;
324         // OpenFlags (nOpenFlags = CFile::modeCreate | CFile::modeWrite)
325         var UINT openFlags := bit2int(bin_openFlags);
326         var CFileExceptionType myException := valueof(Empty_CFileException);
327         var CppPtr pError := New_CFileException(myException);
328         var BOOL retValue;
329 
330         aCFilePort.call(Open:{this, fileName, openFlags, pError}, nowait);
331         alt {
332             [] aCFilePort.getreply(Open:{-,-,-,-} value ?) -> value retValue { }
333         }
334         var CppPtr lpBuf := New_CppString("This is a text that we want to read
335             later on from the file readTest.txt"); // 71 characters
336         // contents of lpBuf is written to the file
337         aCFilePort.call(Write:{this, lpBuf, 71}, nowait);
338         alt {
339             [] aCFilePort.getreply(Write:{-,-,-}) { }
340         }
341 
342         // Close the file
343         aCFilePort.call(Close:{this}, nowait);  // Close the file
344         alt {
345             [] aCFilePort.getreply(Close:{-}) { }
346         }
347 
348         // and open it again...
349         openFlags := modeRead;
350         aCFilePort.call(Open:{this, fileName, openFlags, pError}, nowait);
351         alt {
352             [] aCFilePort.getreply(Open:{-,-,-,-} value ?) -> value retValue { }
353         }
354 
355         // Test step for reading...
356         var CppPtr readString := New_CppString("This is just something crap...we don't
357             need this, but the lenght of this buffer must be at least 71 characters");
358         // The file contains only 71 characters and nCount is 100.
359         // Thus characters are read up to the end of file
360         aCFilePort.call(Read:{this, readString, 100}, nowait);
361         alt {
362             [] aCFilePort.getreply(Read:{-,-,-} value 71) {}
363         }
364         var CppString aText := Get_CppString(readString);
365         if (aText != "This is a text that we want to read later on from
366             the file readTest.txts buffer must be at least 71 characters") {
367             TestCaseFail();
368         }
369         StopIfFail();
370         Del_CppString(readString);
371 
372         // Close the file and carbage collections
373         aCFilePort.call(Close:{this}, nowait);  // Close the file
374         alt {
375             [] aCFilePort.getreply(Close:{-}) { }
376         }
377         Del_CppString(lpBuf);
378         Del_CFileException(pError);
379         Del_CppString(fileName);
380         Destructor(this);   // call destructor
381         TestCasePass();
382 }
383 //
384 //////////////////////////////////////////////////////////////////////////////////
385 
386 
387 
388 //////////////////////////////////////////////////////////////////////////////////
389 // The purpose of this test case is to verify the operation of CFile member
390 // function 'Seek'.
```

```
391  //
392      testcase Seek_Test_01() runs on Tester system CFileTSI {
393          InitTestCase();
394          var CppPtr this := Constructor();
395          var CppPtr lpszFileName := New_CppString("seekTest.txt");
396          var bitstring bin_modeCreate := int2bit(modeCreate, 32);
397          var bitstring bin_modeReadWrite := int2bit(modeReadWrite, 32);
398          var bitstring bin_openFlags := bin_modeCreate or4b bin_modeReadWrite;
399          var UINT nOpenFlags := bit2int(bin_openFlags);
400          // OpenFlags (nOpenFlags = CFile::modeCreate | CFile::modeWrite)
401          var CFileExceptionType myException := valueof(Empty_CFileException);
402          var CppPtr pError := New_CFileException(myException);
403          var BOOL retValue;
404
405          // call open
406          aCFilePort.call(Open:{this, lpszFileName, nOpenFlags, pError}, nowait);
407          alt {
408              [] aCFilePort.getreply(Open:{-,-,-,-} value ?) -> value retValue { }
409          }
410
411          // write test text into a file
412          var CppPtr lpBuf :=
413              New_CppString("The beginning...middle part...the end"); // 37 characters
414          aCFilePort.call(Write:{this, lpBuf, 37}, nowait);
415          alt {
416              [] aCFilePort.getreply(Write:{-,-,-}) { }
417          }
418          Del_CppString(lpBuf);
419
420          // Seek "middle part..." (14 chars)
421          // Seeks 21 bytes from the end of the file, returns the new
422          // byte offset from the beginning of the file (= 16)
423          aCFilePort.call(Seek:{this, -21, end}, nowait);
424          alt {
425              [] aCFilePort.getreply(Seek:{-,-,-} value 16) { }
426          }
427
428          // a char buffer, which has size of 14 bytes
429          var CppPtr aBuf := New_CppString("14 characters.");
430          // read 14 characters and check if the text is read
431          // from the right spot of the file
432          aCFilePort.call(Read:{this, aBuf, 14}, nowait);
433          alt {
434              [] aCFilePort.getreply(Read:{-,-,-} value 14) {}
435          }
436          var CppString aText := Get_CppString(aBuf);
437          if (aText != "middle part...") {
438              TestCaseFail();
439          }
440          StopIfFail();
441          Del_CppString(aBuf);
442
443          // Close the file
444          aCFilePort.call(Close:{this}, nowait);
445          alt {
446              [] aCFilePort.getreply(Close:{-}) { }
447          }
448          Del_CFileException(pError);
449          Del_CppString(lpszFileName);
450          Destructor(this);
451          TestCasePass();
452      }
453  //
454  ///////////////////////////////////////////////////////////////////////////////////////////
455
456
457
458  ///////////////////////////////////////////////////////////////////////////////////////////
459  // This test case is verifying the Duplicate -member function, which
460  // constructs a duplicate CFile object for a given file and returns
461  // a pointer to a duplicate CFile object.
462  //
463      testcase Duplicate_Test_01() runs on Tester system CFileTSI {
464          InitTestCase();
465          var CppPtr this := Constructor();
466
467          // create an empty file...
468          var CppPtr fileName := New_CppString("DuplicateTest.txt");
```

```
469         var UINT openFlags := modeCreate;
470         var CFileExceptionType myException := valueof(Empty_CFileException);
471         var CppPtr pError := New_CFileException(myException);
472         aCFilePort.call(Open:{this, fileName, openFlags, pError}, nowait);
473         alt {
474             [] aCFilePort.getreply(Open:{-,-,-,-} value ?) { }
475         }
476 
477         // call duplicate... creates a new object
478         var CppPtr copy;
479         aCFilePort.call(Duplicate:{this}, nowait);
480         alt {
481             [] aCFilePort.getreply(Duplicate:{-} value ?) -> value copy { }
482         }
483 
484         // close sut and garbage collection
485         aCFilePort.call(Close:{this}, nowait);
486         alt {
487             [] aCFilePort.getreply(Close:{-}) { }
488         }
489         Del_CFileException(pError);
490         Del_CppString(fileName);
491         Destructor(this);
492 
493         // just for testing the earlier copied object...
494         var CFileType copyOfSut := Get_CFile(copy);
495         var CppPtr fileName2 := New_CppString("DuplicateTest2.txt");
496         aCFilePort.call(Open:{copy, fileName2, openFlags, pError}, nowait);
497         alt {
498             [] aCFilePort.getreply(Open:{-,-,-,-} value ?) { }
499         }
500         aCFilePort.call(Close:{copy}, nowait);
501         alt {
502             [] aCFilePort.getreply(Close:{-}) { }
503         }
504         Del_CppString(fileName2);
505         Destructor(copy);   // Delete 'copy' -object
506         TestCasePass();
507     }
508 //
509 ////////////////////////////////////////////////////////////////////////////////////////
510 
511 
512 
513 ////////////////////////////////////////////////////////////////////////////////////////
514 // This test case verfies GetStatus member function with two arguments
515 // This is a static member function, thus 'this' parameter is not passed.
516 //
517     testcase GetStatus_Test_01() runs on Tester system CFileTSI {
518         InitTestCase();
519 
520         // an emty file 'GetStatusTest.txt' is created...
521         var CppPtr this := Constructor();
522         var CppPtr fileName := New_CppString("GetStatusTest.txt");
523         var UINT openFlags := modeCreate;
524         var CFileExceptionType myException := valueof(Empty_CFileException);
525         var CppPtr pError := New_CFileException(myException);
526 
527         // open sut
528         aCFilePort.call(Open:{this, fileName, openFlags, pError}, nowait);
529         alt {
530             [] aCFilePort.getreply(Open:{-,-,-,-} value ?) { }
531         }
532 
533         // close sut
534         aCFilePort.call(Close:{this}, nowait);
535         alt {
536             [] aCFilePort.getreply(Close:{-}) { }
537         }
538         Del_CFileException(pError);
539         Destructor(this);
540 
541         // create &CFileStatus
542         var CFileStatusType myStatus := valueof(Empty_CFileStatus);
543         var CppPtr rStatus := New_CFileStatus(myStatus);
544         var BOOL retVal;
545 
546         // call GetStatus
```

```
547         aCFilePort.call(GetStatus:{fileName, rStatus}, nowait);
548         alt {
549             [] aCFilePort.getreply(GetStatus:{-,-} value ?) -> value retVal { }
550         }
551         if (retVal == 0) {
552             TestCaseFail();
553         }
554         StopIfFail();
555         myStatus := Get_CFileStatus(rStatus);
556         if (myStatus.m_size != 0) {
557             TestCaseFail();
558         }
559         if (Get_CppString(myStatus.m_szFullName) !=
560                 "D:\Case Study\CFileTest\CFileTester\GetStatusTest.txt") {
561             TestCaseFail();
562         }
563         StopIfFail();
564         Del_CFileStatus(rStatus);
565         Del_CppString(fileName);
566         TestCasePass();
567     }
568 //
569 ///////////////////////////////////////////////////////////////////////////////////////////
570
571
572
573
574 /*****************************************************************************************/
575 /*                                                                                       */
576 /*                                    CONTROL PART                                       */
577 /*                                                                                       */
578 /*****************************************************************************************/
579
580     control {
581         var verdicttype aVerdict := none;
582         if(aVerdict == none) {
583             aVerdict := execute(Constructor_Test_01());
584         }
585         if(aVerdict == pass) {
586             aVerdict := execute(Constructor_Test_02());
587         }
588         if(aVerdict == pass) {
589             aVerdict := execute(Open_Test_01());
590         }
591         if(aVerdict == pass) {
592             aVerdict := execute(Open_Test_02());
593         }
594         if(aVerdict == pass) {
595             aVerdict := execute(Write_Test_01());
596         }
597         if(aVerdict == pass) {
598             aVerdict := execute(Read_Test_01());
599         }
600         if(aVerdict == pass) {
601             aVerdict := execute(Seek_Test_01());
602         }
603         if(aVerdict == pass) {
604             aVerdict := execute(Duplicate_Test_01());
605         }
606         if(aVerdict == pass) {
607             aVerdict := execute(GetStatus_Test_01());
608         }
609     }
610 } // module MySUT_header_Test
```

Published by

VTT

Series title, number and report code of publication

VTT Publications 542
VTT–PUBS–542

Author(s)
Pulkkinen, Pekka

Title

Mapping C++ Data Types into a Test Specification Language

Abstract

Software testing is becoming a more and more important and challenging part of software development nowadays. Since the complexity and size of software is growing day by day, software developers must concentrate increasingly on testing, which costs both time and money. Therefore, different methods and tools have been developed to facilitate and precipitate software testing and also improve the quality of software.

One emerging new testing technology is TTCN-3 (Testing and Test Control Notation 3), which is a standardized test specification and implementation language. TTCN-3 provides a broad spectrum of testing abilities and is among others designed for testing software modules. It is also intended to be used for several applications with several data description languages. Even if C++ is one of the most popular programming languages nowadays, TTCN-3 cannot be yet efficiently utilize for testing C++ software. In order to take advantage of TTCN-3 in testing C++ modules, the interface of the tested component should be defined at the TTCN-3 language level. Therefore, C++ data types need to be mapped to TTCN-3.

The purpose of this thesis is to provide data type mappings from C++ to TTCN-3, and to implement a TTCN-3 based test system in order to test a C++ software module. Due to the differences between C++ and TTCN-3, such as lacking of object model in TTCN-3 and ambiguity of C++ pointers, several challenges are faced during this work. However, fairly comprehensive data type mapping is provided, which is finally verified in a real world-like situation by using TTCN-3 to test a C++ module. This example gives a clear insight of the usability and advantage of data type mappings and also valuable experience on the suitability of TTCN-3 in testing C++ software module is gained.

Keywords
Testing and Test Control Notation 3 (TTCN-3), software testing, software development

Activity unit
VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland

ISBN	Project number
951–38–6402–2 (soft back ed.) 951–38–6403–0 (URL:http://www.vtt.fi/inf/pdf/)	E3SU00131

Date	Language	Pages	Price
June 2004	English, Finnish abstr.	89 p. + app. 13 p.	C

Name of project	Commissioned by
TT-Medal-VTT	

Series title and ISSN	Sold by
VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)	VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374

Julkaisija

Julkaisun sarja, numero ja raporttikoodi

VTT Publications 542
VTT–PUBS–542

Tekijä(t)
Pulkkinen, Pekka

Nimeke

C++-tietotyyppien määrittely testienkuvauskielellä

Tiivistelmä

Ohjelmistotestaus on yhä tärkeämpi ja haastavampi osa ohjelmistonkehitysprosessia. Ohjelmistojen koon ja kompleksisuuden kasvaessa testauksen merkitys korostuu. Tämän vuoksi ohjelmistotestauksen helpottamiseksi ja nopeuttamiseksi sekä ohjelmistojen laadun parantamiseksi onkin kehitelty erityisiä menetelmiä ja työkaluja.

Eräs testaukseen kehitetyistä uusista menetelmistä on TTCN-3 (Testing and Test Control Notation 3), joka on standardoitu testien kuvaus- ja toteutuskieli. TTCN-3 tarjoaa laajan valikoiman eri testausmenetelmiä ja sitä voidaan käyttää muun muassa ohjelmistomoduulien testaukseen. TTCN-3 on myös suunniteltu käytettäväksi yhdessä monien kuvauskielten kanssa erityyppisten sovellusten testaamisessa. Vaikka C++ on nykyään eräs suosituimmista ohjelmointikielistä ei TTCN-3:a voida vielä tehokkaasti käyttää C++-ohjelmistojen testaamiseen. Käytettäessä TTCN-3:a C++-ohjelmistomoduulin testaukseen tulee testattavan komponentin rajapinta määritellä TTCN-3-kielellä. Tämän vuoksi tarvitaan määrittelysäännöt C++-tietotyyppien muuntamiseksi TTCN-3-kielelle.

Tässä diplomityössä määritellään C++-tietotyypit TTCN-3-kielellä sekä toteutetaan TTCN-3 testijärjestelmä C++ moduulin testaamiseksi. TTCN-3- ja C++-kielten välillä on suuria eroavaisuuksia, kuten olio-ohjelmointimallin puuttuminen TTCN-3:sta sekä C++-osoittimien moniselitteisyys, minkä vuoksi työn aikana kohdataan useita ongelmia. Tästä huolimatta työssä toteutetaan suhteellisen kattavat tyyppimäärittelyt, joita verifioidaan käyttämällä TTCN-3:a erään C++-moduulin testaukseen. Tämä esimerkki antaa selkeän kuvan tyyppimäärittelyjen käytettävyydestä ja hyödyllisyydestä. Lisäksi saadaan arvokasta kokemusta TTCN-3:n soveltuvuudesta C++-ohjelmistojen testauksessa.

Avainsanat
Testing and Test Control Notation 3 (TTCN-3), software testing, software development

Toimintayksikkö
VTT Elektroniikka, Kaitoväylä 1, PL 1100, 90571 OULU

ISBN			Projektinumero
951–38–6402–2 (nid.) 951–38–6403–0 (URL: http://www.vtt.fi/inf/pdf/)			E3SU00131
Julkaisuaika	Kieli	Sivuja	Hinta
Kesäkuu 2004	Englanti, suom. tiiv.	89 s. + liitt. 13 s.	C
Projektin nimi		Toimeksiantaja(t)	
TT-Medal-VTT			
Avainnimeke ja ISSN		Myynti:	
VTT Publications 1235–0621 (nid.) 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)		VTT Tietopalvelu PL 2000, 02044 VTT Puh. (09) 456 4404 Faksi (09) 456 4374	

VTT PUBLICATIONS

522 Jokinen, Tommi. Novel ways of using Nd:YAG laser for welding thick section austenitic stainless steel. 2004. 120 p. + app. 12 p.

523 Soininen, Juha-Pekka. Architecture design methods for application domain-specific integrated computer systems. 2004. 118 p. + app. 51 p.

524 Tolvanen, Merja. Mass balance determination for trace elements at coal-, peat- and bark-fired power plants. 2004. 139 p. + app. 90 p.

525 Mäntyniemi, Annukka, Pikkarainen, Minna & Taulavuori, Anne. A Framework for Off-The-Shelf Software Component Development and Maintenance Processes. 2004. 127 p.

526 Jäälinoja, Juho. Requirements implementation in embedded software development. 2004. 82 p. + app. 7 p.

527 Reiman, Teemu & Oedewald, Pia. Kunnossapidon organisaatiokulttuuri. Tapaustutkimus Olkiluodon ydinvoimalaitoksessa. 2004. 62 s. + liitt. 8 s.

528 Heikkinen, Veli. Tunable laser module for fibre optic communications. 2004. 172 p. + app. 11 p.

529 Aikio, Janne K. Extremely short external cavity (ESEC) laser devices. Wavelength tuning and related optical characteristics. 2004. 162 p.

530 FUSION Yearbook. Association Euratom-Tekes. Annual Report 2003. Ed. by Seppo Karttunen & Karin Rantamäki. 2004. 127 p. + app. 10 p.

531 Toivonen, Aki. Stress corrosion crack growth rate measurement in high temperature water using small precracked bend specimens. 2004. 206 p. + app. 9 p.

532 Moilanen, Pekka. Pneumatic servo-controlled material testing device capable of operating at high temperature water and irradiation conditions. 2004. 154 p.

534 Kallio, Päivi. Emergence of Wireless Services. Business Actors and their Roles in Networked Component-based Development. 2004. 118 p. + app. 71 p.

535 Komi-Sirviö, Seija. Development and Evaluation of Software Process Improvement Methods. 2004. 175 p. + app. 78 p.

537 Tillander, Kati. Utilisation of statistics to assess fire risks in buildings. 2004. 224 p. + app. 37 p.

538 Wallin, Arto. Secure auction for mobile agents. 2004. 102 p.

539 Kolari, Juha, Laakko, Timo, Hiltunen, Tapio, Ikonen, Veikko, Kulju, Minna, Suihkonen, Raisa, Toivonen, Santtu & Virtanen, Tytti. Context-Aware Services for Mobile Users. Technology and User Experiences. 2004. 167 p. + app. 3 p.

540 Villberg, Kirsi, Saarela, Kristina, Tirkkonen, Tiina, Pasanen, Anna-Liisa, Kasanen, Jukka-Pekka, Mussalo-Rauhamaa, Helena, Malmberg, Marjatta & Haahtela, Tari. Sisäilman laadun hallinta. 2004. 172 s. + liitt. 20 s.

541 Saloheimo, Anu. Yeast *Saccharomyces cerevisiae* as a tool in cloning and analysis of fungal genes. Applications for biomass hydrolysis and utilisation. 2004. 84 p. + app. 51 p.

542 Pulkkinen, Pekka. Mapping C++ Data Types into a Test Specification Language. 2004. 89 p. + app. 13 p.

Tätä julkaisua myy	Denna publikation säljs av	This publication is available from
VTT TIETOPALVELU PL 2000 02044 VTT Puh. (09) 456 4404 Faksi (09) 456 4374	VTT INFORMATIONSTJÄNST PB 2000 02044 VTT Tel. (09) 456 4404 Fax (09) 456 4374	VTT INFORMATION SERVICE P.O.Box 2000 FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374

ISBN 951–38–6402–2 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6403–0 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)