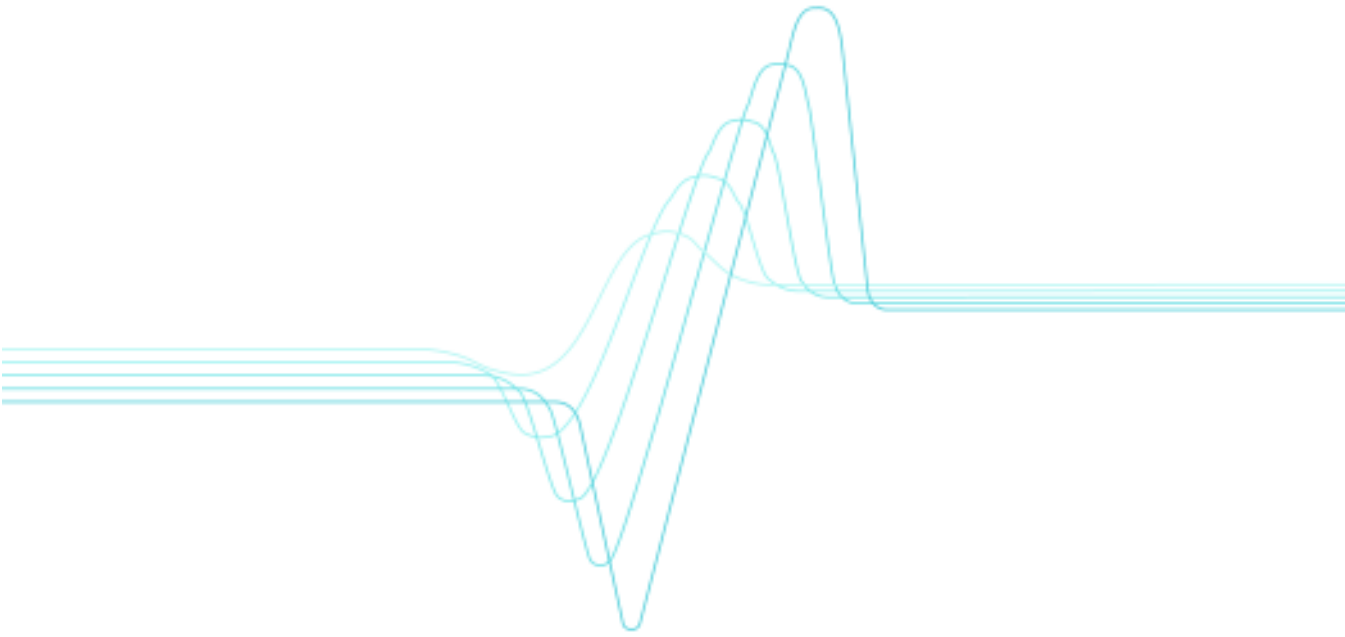
ESPOO 2005 VTT PUBLICATIONS 557

Pekka Mäki-Asiala

Reuse of TTCN-3 Code



VTT PUBLICATIONS 557

Reuse of TTCN-3 Code

Pekka Mäki-Asiala

VTT Electronics

ISBN 951–38–6431–6 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6432–4 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. +358 20 722 111, fax + 358 20 722 4374

VTT Elektroniikka, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde 020 722 111, faksi 020 722 2320

VTT Elektronik, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel 020 722 111, fax 020 722 2320

VTT Electronics, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. +358 20 722 111, fax +358 20 722 2320

Technical editing Leena Ukskoski

Valopaino Oy, Helsinki 2005

Mäki-Asiala, Pekka. Reuse of TTCN-3 Code [TTCN-3-koodin uudelleenkäyttö]. Espoo 2005. VTT Publications 557. 112 p.

Keywords software testing, software reuse, test reuse

Abstract

Today, the growing size and complexity of software along with decreasing development times causes tremendous challenges to software testing. This has driven the whole software industry to seek new ways to test more efficiently and effectively.

Software reuse has been practiced for decades and successful industrial studies have demonstrated such profits as increased productivity and quality as well as decreased development times and costs. This raises the question of whether software reuse could be applied to a testing context as well.

This work studies the reuse of tests that are created with a new test specification and implementation language TTCN-3 (Testing and Test Control Notation). In order to apply reuse into a testing context, a set of guidelines for reusable TTCN-3 code is presented. These guidelines are based on the techniques familiar from software reuse, TTCN-3 test system and language characteristics, and on some of the specifics of software testing. Applicability of the guidelines, and the level and profits of TTCN-3 test reuse are determined in a case study. The case study plainly demonstrates that the majority of the guidelines were successfully applied and that they had a positive impact on measured levels and profits of reuse. The overall results, experiences and impressions of TTCN-3 test reuse during this work were very encouraging and will hopefully lead to future projects in areas of test reuse.

Mäki-Asiala, Pekka. Reuse of TTCN-3 Code [TTCN-3-koodin uudelleenkäyttö]. Espoo 2005. VTT Publications 557. 112 s.

Keywords software testing, software reuse, test reuse

Tiivistelmä

Ohjelmistojen koon ja kompleksisuuden kasvaminen ja samanaikainen kehitysajan lyhentyminen aiheuttavat ohjelmistotestaukselle suuria haasteita. Tämä suuntaus on pakottanut ohjelmistoteollisuuden etsimään uusia keinoja testauksen tehostamiseksi.

Ohjelmistojen uudelleenkäyttöä on harjoitettu vuosikymmenien ajan. Uudelleenkäytön menestyksekkään soveltamisen on huomattu tarjoavan merkittäviä etuja, kuten tuottavuuden ja laadun parantumista, kehityskulujen vähenemistä ja kehitysajan lyhentymistä. Tämä herättää kysymyksen uudelleenkäytön soveltamisesta myös ohjelmistotestauksessa.

Tässä työssä tutkittiin TTCN-3-kielellä luotujen testien uudelleenkäyttöä. Tätä varten luotiin erityiset testien uudelleenkäyttöä edistävät ohjeet, jotka pohjautuvat tunnettuihin ohjelmistojen uudelleenkäyttötekniikoihin, TTCN-3-testijärjestelmän ja -kielen ominaisuuksiin ja ohjelmistotestauksen erityispiirteisiin. Ohjeiden soveltuvuutta ja testien uudelleenkäyttöä arvioitiin tapaustutkimuksessa, joka osoitti ohjeiden hyödyllisyyden saavutetuissa tuloksissa. Yleisvaikutelma tuloksista ja kokemuksista oli rohkaiseva, mikä toivottavasti heijastuu tulevaisuuteen testien uudelleenkäyttöä tutkivina jatkohankkeina.

Foreword

The research work for this thesis has been carried out at the Technical Research Centre of Finland in the Software Platforms group of the Embedded Software research field during the spring and fall of 2004. The work was done as a part of the TT-Medal project (Tests & Testing Methodologies with Advanced Languages).

I would like to thank all the people that have contributed to this work and supported me during this process. First of all, I would like to thank Mr. Markus Sihvonen for having so much faith in my capabilities so much that I was hired as a research scientist trainee and had the opportunity to work with this subject. Secondly, my deepest gratitude goes to Mr. Matti Kärki whose significant guidance during the whole process and countless comments and suggestions, especially concerning chapters 6 and 7, were a tremendous help. Thirdly, I would like to express my gratitude to Mrs. Annukka Mäntyniemi for her expertise in software reuse and to Mr. Pekka Pulkkinen for sharing his knowledge of TTCN-3, as well as for commenting on the guidelines.

Finally, my supervisor at the university, Professor Tapio Seppänen and the work's 2nd reviewer, Professor Jukka Riekki, receive my appreciation for the effort that they have used to review and comment on this work.

Outside the work environment, I would like to thank all my friends and relatives and especially my family for giving me something other than work or studies to be occupied with during the past few years.

Oulu, October 25, 2004

Pekka Mäki-Asiala

Contents

Acronyms and Abbreviations

AA ATM Adaptation, conversion of data to and from the ATM cell.

AAL ATM Adaptation Layer a collection of standardized protocols that allow multiple applications to have data converted to and from the ATM cell.

ASN.1 Abstract Syntax Notation One, a language used by the OSI protocols for describing data types independent of particular computer structures and representation techniques.

ATM Asynchronous Transfer Mode, a dedicated connection switching technology in which the information is organized into cells.

ATS Abstract Test Suite, an abstract collection of test cases.

BIT Built-in Test, a test that is built in the system itself.

CBSE Component-based Software Engineering, a sub-discipline of software engineering that emphasizes design and construction of software systems by using components.

CD Coding and Decoding, an entity in a TTCN-3 test system. Encodes the TTCN-3 values into bitstrings suitable to be sent to the System Under Tests and decodes the received values into TTCN-3 values.

CP-AAL Common Part AAL Protocol, provides unassured information transfer and a mechanism for detecting corruption of SSCOP PDUs.

CPCS Common Part Convergence Sublayer, a portion of the convergence sublayer that is independent of the type of traffic being converted.

CUT Component Under Test, a particular component of a software system that is tested.

ETS Executable Test Suite, an executable realization of the Abstract Test Suite (ATS).

ETSI European Telecommunications Standards Institute

FIFO First In, First Out, a queue handling method that operates on a first-come, first-served basis.

GFT Graphical presentation Format of TTCN-3, a graphical presentation format that provides a visualization of TTCN-3 behavior definitions.

HCI Human-Computer Interaction, the way people interact with computer systems.

ITU-T International Telecommunication Union-Telecommunication Standardisation Sector

IUT Implementation Under Test, a particular portion of a software system that is under test.

IDL Interface Definition Language, a language for defining interfaces enabling communication between modules implemented in different languages.

KLOC Thousands (Kilo) of Lines Of Code, measures the size of computer programs in thousands of lines of code.

LOC Lines Of Code, measures the size of computer programs in lines of code.

MTC Main Test Component

OSI Open Systems Interconnection, a standard for representing network protocols.

PA Platform Adaptor, an entity in a TTCN-3 test system. Adapts the TTCN-3 Executable (TE) to a particular execution platform and provides the TTCN-3 test system with a single notion of time.

PDU Protocol Data Unit, a basic transferable data unit of a protocol.

RCR Relative Cost Of Reuse

RCWR Relative Cost of Writing Reusable Software

ROI Return On Investment, amount of value received relative to the amount of investment.

SA System Adapter, an entity in a TTCN-3 test system. Adapts the TTCN-3 communication operations with the SUT based on an abstract test system interface.

SAAL Signalling ATM Adaptation Layer, a service that resides above the ATM layer and ensures that signalling messages reach the receiver. SAAL has two main sublayers: the Common Part Convergence Sublayer (CPCS) and the Service Specific Convergence Sublayer (SSCS).

SAP Service Access Point, a connection point between a protocol in one OSI layer and a protocol in the layer above.

SIP Session Initiation Protocol, a signalling protocol that uses text based messages and supports multimedia communication.

SSCF Service Specific Coordination Function, maps the service of SSCOP to the needs of the AAL user.

SSCOP Service Specific Connection Oriented Protocol, provides a generic reliable data transfer service for different AAL interfaces defined by the SSCF.

SSCS Service Specific Convergence Sublayer, a portion of the convergence sublayer that is dependent upon the type of traffic being converted.

SUT System Under Test

TC Test Control, a part of the Test Management (TM) entity. Responsible for the proper invocation of the TTCN-3 modules.

TCI TTCN-3 Control Interface, comprised of the three interfaces that define the interaction of the TTCN-3 Executable (TE) with the Test Management (TM), the coding and decoding (CD), and the test component handling (CH) in a test system.

TE TTCN-3 Executable, the part of a test system that deals with interpretation or execution of a TTCN-3 ETS.

TFT 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 in a TTCN-3 test system. Provides a user interface as well as the administration of the TTCN-3 test system.

TP Test Purpose, a prose description of a well-defined objective of testing, focusing on a single conformance requirement or a set of related conformance requirements.

TRI TTCN-3 Runtime Interface, a standardized interface that provides an adaptation for timing and communication of a test system to a particular processing platform and the system under test, respectively.

TTCN Tree and Tabular Combined Notation

TTCN-2 Tree and Tabular Combined Notation, version 2

TTCN-3 Testing and Test Control Notation, version 3

1. Introduction

The motivation for the work presented in this thesis roots itself in the challenges that are faced in software development organizations all over. The growing size and complexity of software along with the trend of decreased time-to-market has driven the whole software industry to seek new means of quality assurance and testing.

The amount of effort consumed by software testing varies from 30 to 50 percent in a typical software development project [1]. Adding the amount of effort consumed by testing in the maintenance phase increases the overall cost of testing so that it is likely to become the most expensive part of the software's lifespan. However, even though testing is expensive, the cost of software errors may in some cases be even higher or financially unbearable.

According to a recent study [2], software errors cost the U.S economy an estimated $59.5 billion annually, which is about 0.6 percent of the gross domestic product or one-third of the total sales of software ($180 billion) in 2000. Obviously, no matter what the scale, the cost of software errors is high. However, according to the same study, more than one-third of these costs, an estimated $22.2 billion, could be avoided by an improved testing infrastructure that is able to find an increased percentage (but not 100 percent) of errors [2].

It is fair to say that the problems with software errors are well-known on this side of the Atlantic as well, and their origins are not in the only in the U.S. One of the indications that software errors are a worldwide concern is that the global market for automated software quality tools reached $931 million in 1999. Compared to 1998, the increase was 23.6% and the same increase is expected annually, meaning that this year (2004) the market is about $2.6 billion. [3]

Obviously, software testing is the best way to fight software errors. But it is not the only way. One of the advantages of software reuse is that it can decrease the amount of errors and improve the quality of a software product. This is caused by the fact that reusable components, i.e. the building blocks of software products, are examined by more people than they would be if they were developed for one product only. Software reuse also promises better productivity and shorter development times, which brings us back to the original problem –

the pressures for software testing. This raises the question of whether software reuse techniques could be applied to a testing context as well.

This work studies the reuse of tests that are created with a new test specification and implementation language TTCN-3 (Testing and Test Control Notation, version 3). TTCN-3 is a fairly new test specification and implementation language for all types of black box testing. [4] With TTCN-3, specific test software, also known as testware, can be developed to test software components and products.

The research work for this thesis has been carried out at the Technical Research Centre of Finland (VTT) as a part of the TT-Medal project (Tests & Testing Methodologies with Advanced Languages). The project aims at developing methodologies, tools and industrial experience to enable European industry to test more efficiently and more effectively [5]. The role of VTT is to develop the test infrastructure and to study test reuse. Reuse is studied on three levels: the TTCN-3 language level, the test process level and the test system level. The author's role in the project has so far focused mainly on studying reuse on the TTCN-3 language level.

The scope of this thesis is limited to studying reuse of testware created with TTCN-3 and not to determine, e.g. how the testing process or the test organization infrastructure changes as the principles known from software reuse are applied to the testing context. Since TTCN-3 is a relatively new language and test reuse, at least compared to software

reuse, is a fairly uncharted field of study, some groundwork needs to be done instead of just rushing into creating reusable testware.

The first objective of this work is to create general guidelines for reusable TTCN-3 code. These guidelines will hopefully help in creating generic and adaptable solutions when implementing TTCN-3 test scripts. The guidelines are based on the reuse techniques known from software reuse, the TTCN-3 test system and language features and on a few specifics of software testing.

The second objective of this work is to apply these guidelines in a case study. The case study will determine the applicability of the guidelines and provide

feedback that will lead to changes and updates. The reuse level of testware is measured based on some of the reuse metrics known from software reuse that can be applied to the test reuse context, within the scope of the case study.

The third objective is to gain valuable experience of TTCN-3 test reuse in order to promote the use of TTCN-3 and test reuse in the field of science and the software industry.

The rest of this thesis is organized as follows. The second chapter gives a brief overview of software testing in general, providing the primary information on testing process, levels and types. The basic issues related to software reuse such as reuse approaches, techniques and metrics are presented in chapter 3. TTCN-3 core language and the test system is laid out in chapter 4. Chapter 5 presents test reuse based on the work done in the TT-Medal project by the author and others, and some of the past studies on test reuse.

The actual work done for this thesis, by the author himself begins in chapter 6, which presents guidelines for reusable TTCN-3 code. The guidelines are applied and evaluated, and the level of reuse is measured in a case study in chapter 7. Chapter 8 discusses how the objectives laid out for the work were reached and contemplates the future work. Chapter 9 summarizes the work.

2. Software Testing

A literature search provides multiple definitions for software testing. Perhaps the most commonly known is the definition by Myers: “Testing is the process of executing a program with the intent of finding errors” [6 p. 5]. Clearly, Myers’ definition approaches testing from a traditional point of view, where errors are found by executing programs. Hetzel’s definition takes a wider scope as he notes: “Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.” [7 p. 6] Hetzel’s definition covers the two widely recognized reasons for software testing: problem discovery and quality assessment.

2.1 Testing Process

The testing process is the glue that binds people, methods, tools and measurements together in order to achieve the common goal of testing a software product. The quality of the software product is a result of the used software development process. In the same way, the quality of testing is mostly determined by the testing process used. [8] In fact, the testing process not only affects the quality of testing, but the quality of the software product as well. Even though a group of people may be able to produce a high quality software product without a well-defined testing process, they will not be able to prove it if the testing process is immature and incapable of presenting any indications of this quality. Furthermore, not only will a mature testing process reveal problems in the software product, but in the development process as well [8]. Hence, the software development process and testing process should have a close and productive relationship.

The V-Model presented in Figure 1 is based on the traditional software development model called the Waterfall model and the vision that software development and testing process should be closely bound together. The presented model is based on similar models presented in [9 p. 9; 10 p. 7; 11 p. 15; 12 p. 159; 13 p. 52; 14 p. 18]. In this model, the development of tests is a concurrent process with the software development process.

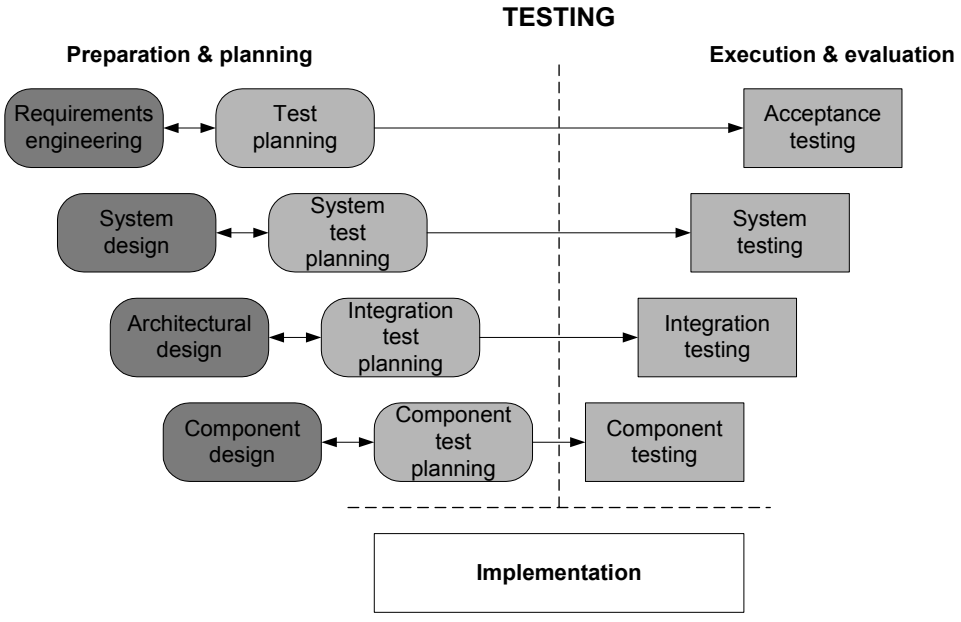


Figure 1. V-model for software development and testing.

Testing-wise Figure 1 is divided into three parts: preparation and planning; implementation; and execution and evaluation. The left-hand side of preparation and planning has a testing activity corresponding to every stage of the software development process. For instance, necessary system tests are identified and designed in parallel with system design, and a test plan for system testing is created. In the implementation phase, the tests are implemented according to their designs. The right-hand side of execution and evaluation exploits the previously created test plans when the corresponding test execution activity is topical.

Another view of the software testing process is that of Test Management Approach (TMap). This view divides the overall testing process into five consecutive steps, instead of the vertical levels presented in the V-model. [11]. The TMap process is illustrated in Figure 2.

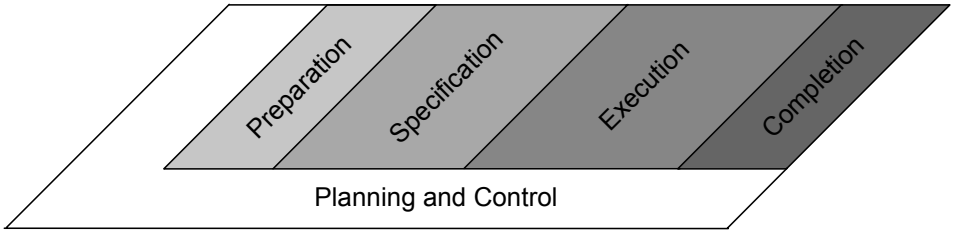


Figure 2. TMap process model.

The TMap process model is generic and applicable to all types of testing. The planning and control starts during the specification of the functional requirements and spans the whole testing process. The planning and control phase provides several documents and plans that define the test organization, testing strategy and effort, risk taxation, etc. These documents are needed for test management and quality reporting. The preparation phase begins as soon as the necessary test plans and system specifications have been drawn up and agreed. Preparation includes training the staff and a study of the specifications. During the specification phase, the test infrastructure is set up and the test cases are specified. The execution phase begins after the completion of test preparation and planning and when the first components to be tested are available and mature enough to be tested. During the completion phase, the testing process and the quality of the IUT are evaluated and the final report prepared and presented. [11] The connection between the V-model and TMap is that the steps of the TMap model can be applied on every level of the V-model.

2.2 Static and Dynamic Testing

There are two ways to test a software implementation: static testing (also known as verification testing) and dynamic testing (also known as validation testing) [8, 11, 15].

Static testing refers to testing something that is not executed [15]. This includes inspections, walkthroughs and technical reviews of work products such as requirements, designs and source code [8]. Generally, static testing can begin very early in the development process and reveal bugs at an early stage. Thus, it usually has a tremendous impact when improving the quality of the product and

cutting down on the development costs [11]. However, static testing cannot reveal all the bugs, especially those that are visible only in the running system.

Dynamic testing refers to testing conducted by executing the implementation [15]. Referring to the testing definition of Myers [6 p. 5], dynamic testing cannot begin before there is something to be executed. In this thesis dynamic testing is simply called *testing*.

2.3 Designing and Identifying Tests

There are two basic approaches to identifying and designing test cases: functional and structural testing. Both of these approaches have several test case identification methods, also known as testing methods. [12]

Functional testing, also known as black box, data-driven, input/output-driven, behavioral, responsibility-based or specification-oriented testing, pays no attention to the internal structure or behavior of the implementation [6, 12, 15, 16, 17].

Functional testing is based on the perception that every program can be modeled as a function that performs a specified action when triggered by a specific input. Test cases in functional testing are identified and designed based on the specifications of the software, thus providing two major advantages: concurrent design of software and testware, and implementation independence. When test design begins very early in the development process, the maturity of testware and the test system is more likely to be adequate when the actual test execution takes place. In addition, it is common for the software implementation to change repeatedly during the development process, thus it is better to base test cases on more static specifications than on constantly changing implementations. However, functional testing has its downsides as well. The identified test cases may stress some parts of the program aggressively and redundantly, while paying insufficient attention to the other parts. [12] This is because test cases are based on what the implementation is supposed and not supposed to do, but not on how it is actually done.

Structural testing, also known as implementation-based, glass box, clear box, white box or logic-driven testing, uses the actual implementation (i.e. source code) to identify test cases [6, 12, 15, 16, 17].

Structural testing is based on understanding the internal structure and behavior of the implementation. Test cases are identified and designed by examining the code. [15] On the one hand, this approach can reveal errors even if they are associated with behavior that is not specified, since it focuses on the implementation and not on the specification. On the other hand, structural testing is unable to reveal that some specified behavior is missing. This is because test cases are specified by what is implemented and not by what should be implemented. [12]

The question of which one of the two presented approaches is better cannot be answered explicitly. Both approaches have had their profound spokesmen over the years, but the common conception today is that neither one is sufficient used on its own, since they complement each others' limitations in finding bugs [12, 17].

2.4 Testing Levels

Software testing is usually done at several levels as depicted on the right-hand side of Figure 1. Test levelling is intended to ensure that the implementation works according to corresponding designs. For example, component tests are needed to validate that the component implementations work according to component design. [11] The most common levels during the development process are: component testing (also known as unit testing or module testing), integration testing and system testing [16, 15]. In addition, acceptance testing is included in Figure 1, however, it is considered to be post-development testing [16]. Testing levels can be divided into two categories: low-level and high-level testing [8, 11].

Low-level testing means testing of individual components or component combinations, in other words component and integration testing. Low-level testing requires a detailed knowledge of the internal structure and behavior of the implementation and is therefore usually performed by the software development team mostly using structural testing methods. [8, 11]

Component testing is the process of testing the individual components or groups of related components of a software system [8]. Components are usually relatively small compared to the size of the complete software system. This eases the task of pinpointing errors, since the errors are known to exist in the specific component under test (CUT) [6]. However, the downside is that component tests may require some parts of the actual system to be simulated using drivers and stubs (also known as upper and lower tester) [8]. A driver is a small program used to invoke and test a CUT, providing inputs and control of test execution [18]. A stub is a small program used to simulate the parts of the system that the CUT uses [16]. Drivers and stubs are replaced with real implementations in the integration testing.

Integration testing is the process of testing in which software components are combined and tested to evaluate the interaction between them [8]. There are two approaches to performing integration testing of components: incremental and non-incremental. Non-incremental, also known as “big bang”, integration is the simplest of the known approaches. [8] Components are combined all at the same time in an attempt to prove basic system stability. The two weaknesses of this approach are the difficulty of locating the errors and usually insufficient coverage of testing. [16] In incremental testing, components are combined one or a few at a time until all of the components have been integrated. There are several integration strategies (also known as patterns), but all of them are variations of either the top-down or bottom-up approach. [8] In the top-down approach, components are integrated starting from the components on the highest level of the control hierarchy. In bottom-up, components are integrated starting from the components with the least dependencies with other components. [16] Regardless of the chosen approach, the two primary goals of integration testing are to find errors between component interfaces [8] and to ensure sufficient interoperability of components before moving on to system testing [16].

High-level testing means testing of complete software systems. It requires a certain level of objectivity and should therefore be conducted by an independent test team (system testing) or by the customer (acceptance testing), mostly using functional testing methods. [8, 11]

System testing is the process of testing a whole integrated application in order to evaluate the systems correspondence to its specified requirements [18]. System

tests are usually based on requirements specifications and system designs. This makes system testing a demanding task, since the requirements are usually generic, providing freedom of implementation, and yet they should be explicit enough to be testable. In addition, requirements may not only be functional, but also cover demands for security performance, usability, etc. [8] Types of testing are addressed briefly in section 2.5.

Acceptance testing is the process of comparing the end product to the needs of end users (i.e. client). It is usually conducted by the end user(s) after successful completion of system testing. [8] Acceptance testing is conducted against the requirements specification [11] and/or specific acceptance criteria determined by the client [8].

2.5 Types of Testing

According to Binder: "System test cases must be derived from an implementation-independent specification of capabilities." [16 p. 718]. Implementation-independent source material means almost all the material produced in the software development project, like for example, requirements, designs, models, use cases and GUI-prototypes. However, system tests should test all the capabilities of the system, not only those that are implementation-independent but implementation-specific as well. [16]

Implementation-specific capabilities, for example performance and usability, are not always specified beforehand, mostly due to the amount of extra effort it would take, and the lack of vision to see them as real requirements. In addition, they are sometimes considered to be "standard accessories" by clients and end users who presume that the software automatically has high security and good usability combined with great performance. Therefore, in order to meet the unspecified demands of clients, a number of testing types (also known as strategies) have evolved. The purpose of these testing types is to exercise the implementation-specific capabilities of the software system before releasing it to acceptance testing [16]. Different testing types are presented in Table 1.

Table 1. Types of testing.

Type of testing	Definition
Concurrency	To determine the effects of multiple calls to application data and functions, in an operating system with shared resources and support for concurrent execution [16].
Configuration	To determine the legal hardware and/or software environment combinations that cause the SUT to fail [16].
Compatibility	To determine whether the compatibility objectives of the software system have been met [8].
Load/stress	To determine the peak load conditions that cause the software system to fail [8]. A sub-category of performance testing [16].
Localization	To determine the software systems ability to be configured for use with locale parameters, e.g. different languages [16].
Performance	To determine whether the software system complies with its performance requirements [18].
Recovery/ restart	To determine whether the software system recovers after a failure as specified in the requirements [8].
Reliability/ availability	To determine whether the system complies with its reliability and availability requirements [8].
Resource usage	To determine if the software system exceeds the level of resources (memory, CPU time, etc.) appointed to it in the requirement specification [8].
Security	To determine whether the software system's security requirements have been met [8]. To evaluate the software system's capability to shield itself against security breaches [16].
Serviceability	To determine the software system's ability to accept updates and repairs [16].
Usability	To determine the level and quality of the HCI design [16].
User documentation	To determine the completeness and correctness of user documentation and automated assistants [16].
Volume	To determine whether the software system can process specified amounts of data or requests [8]. A sub-category of performance testing [16].

In addition, two types of testing have been subjected to closer examination because of their significance to the case study and test reuse.

Conformance testing is the process of testing the extent to which an implementation is a conforming one. A conforming implementation satisfies both static and dynamic conformance requirements. [19] Static requirements define the minimum capabilities to facilitate interoperability whereas dynamic are all those requirements that specify the observable behavior in instances of communication. The goal of conformance testing is to increase the probability that different conformance implementations actually interoperate in heterogeneous systems. [20]

Regression testing is selective retesting of a component or system in order to verify that changes in the implementation under test (IUT) have not caused any unintended errors and that the IUT still conforms to its specified requirements [18]. Regression testing and the reuse of tests have some commonalities. In both of them the idea is to create tests once and execute them multiple times. As rational regression testing demands test automation and careful consideration of what to automate, reuse similarly demands careful analysis of reuse potential. Furthermore, *rerunning* tests (in regression testing) is not the same as *reusing* tests. Whereas rerunning does not take any position on the change of context or IUT, reusing does. High maintenance effort every time the IUT changes is one of the major problems of traditional regression testing tools. As Fewster and Graham [10 p. 11] state: "Test maintenance effort has been the death of many test automation initiatives." Reusing the tests with minimal maintenance cost is one of the expected advantages of test reuse.

2.6 Test Suite and Test Case Structures

This section describes test suite structure on a logical level, being basically a related collection of test cases. A test suite structure is illustrated in Figure 3. The decisive level is the test case level. Each test case has a specific test purpose and they should be grouped accordingly into nested test groups to provide logical ordering of the test cases. A test group may have a common objective that is a specific goal of all the test cases (i.e. test purposes) within that group. Test cases themselves can be modularized into nested test steps that can be grouped together

into test step libraries. Test steps can be further refined into test events that are the smallest units within a test suite (e.g. send or receive a PDU). [19]

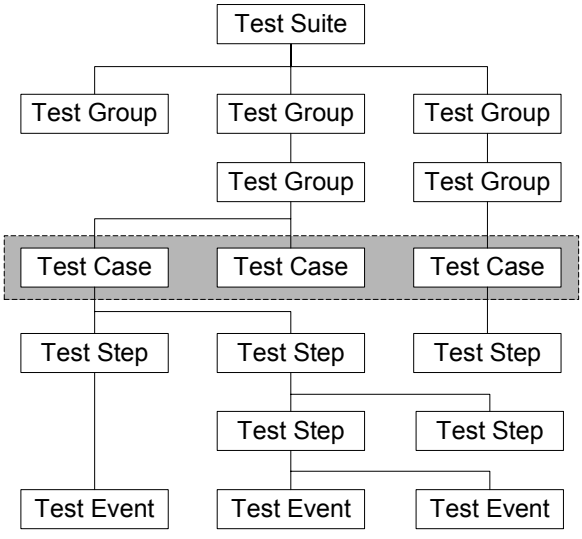


Figure 3. Test suite structure.

Figure 4 describes a test case using a slightly different approach where test case parts and test states are coupled. The test case can be divided into three parts; preamble, postamble and test body. The test states are stable state, test state and end state. The test case starts and ends in states that are stable, but not necessarily identical. With the preamble, the IUT is driven from a stable state to the test state from which the test body is executed. If the test body does not end in a unique end state, it has to be ascertained using a verification step, after which a postamble is used to drive the IUT into a stable testing state. However, if the end state is unique, the IUT is driven into a stable state without the verification step. [21]

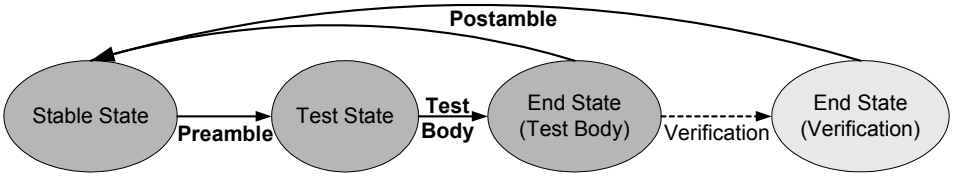


Figure 4. Test case scheme.

3. Software Reuse

Software reuse has been practiced for decades, evolving from ad-hoc code reuse into today's component-based software engineering and product-line engineering approaches [22]. During this time, reuse has found many spokesmen who have actively highlighted the profits of reuse and the ways to achieve them. However, their views have not gone unchallenged, as the results of applying reuse to industrial practice have not always matched expectations.

Software reuse does not only involve the reuse of source code but the reuse of designs, specifications, documentation, etc. [23]. In this thesis, however, the focus is on code reuse.

3.1 Motivation

Today software organizations seek new ways to obtain or retain a competitive edge when compared to rivals. When practiced systematically, software reuse can provide several virtues. Practicing systematic reuse has been found to [24, 25, 26]:

- increase software productivity,
- increase software product interoperability,
- increase software product portability,
- increase market agility,
- improve predictability of the process,
- reduce software development costs,
- reduce software maintenance costs,
- reduce software defect density,
- shorten software development time,
- produce higher quality software products, and
- provide competitive advantage.

Even though practical experience has proved that software reuse can at its best provide such encouraging results, introducing reuse to the software process has not always been successful.

3.2 Challenge

Some studies indicate that the problem with reuse is caused by poor management, while some consider that reuse is a challenging technical problem. Glass states that reuse has failed on the large scale because there simply is not that much that we can reuse. [27] Glass makes a good argument. However, it has never been the goal of reuse spokesmen to suggest that everything is reusable or even that most of the software should be. It takes planning and consideration to find out just what is worth reusing [24, 28]. In another article, Glass explains what causes the low levels of reuse: "It's the variability in the problems we solve, and in the solutions we create." Meaning that the probability of facing the exact same problem that someone else has faced before is small, if not zero. In addition, even though the problem might be the same, finding a solution that is nearly suitable is usually just not good enough [29]. However, as Glass and others have found out, reuse has succeeded, especially in software organizations with a narrow field of application (i.e. application domain). As an example, Glass points out the Software Engineering Laboratory at Nasa, which generally reaches reuse levels above 70 percent when creating new software systems [27].

Morisio et al. analyzed success and failure factors in software reuse based on the experiences of 24 software projects conducted in separate European companies between the years 1994 and 1997. They found out that despite the potential success for reuse, around one-third of the projects failed. This was the result of three main factors: not introducing reuse-specific processes, not modifying non-reuse processes, and not considering human factors. [30] Some examples of risks and disadvantages compromising the success of reuse are as follows:

- *Time and effort required for development of reusable components*. Developing reusable components requires more effort than required to develop a specific solution for one time use only [24]. Many organizations are not willing to make the upfront investment in reuse and wait for the cumulative profits over time. Not even though reuse is considered a long-term investment. [25]
- *Unclear and ambiguous requirements*. Reusable components are to be used in various applications, some of them being unknown. Therefore, requirements for reusable components are hard to predict [31].

- *Conflict between usability and reusability*. Reusability usually means that components are general, adaptable and therefore easier to reuse. However, general components demand more of resources, and adaptability means that they are more complicated to use. [31] Szyperski summarizes the conflict by stating: "Maximizing reuse minimizes use." [32 p. 37]. Meaning that the more reusable the component is made, the less usable it is and vice versa.
- *Component maintenance costs*. Component maintenance costs can be high since the component must respond to various requirements of different applications and environments [31]. For example, when a new application version is introduced it may contain a new component version as well. However, this does not mean that the old version of the component should not be supported anymore, because some of the reusers may still need it.
- *Reliability and sensitivity to change.* Components may cause unexpected side effects in the final product, e.g. because of concealed characteristics not visible to application developers. Changes in the component or in the application may lead to system failures. [31]
- *Resistance to change*. Reuse demands a new set of methods to be promoted and applied to different stages of the software development processes. This means changes in the ways that people work. Changes, even though commonly proven and recognized to be useful, usually generate resistance. [25]

3.3 The Two Sides of Reuse

Reuse is generally divided into two main sides or activities: development *for* reuse and development *with* reuse [28, 26], also referred to as *asset* development and *product* development [33] or even *producer* reuse and *consumer* reuse [25]. The latter terms are more descriptive, as they clearly define the general objectives of the two sides of software reuse. First, the reusable assets are produced and second products are built by reusing these assets. However, even though reusable tests can be called assets, combinations of reusable tests are never called products. Therefore, in this thesis the selection is made in favor of the terms development *for* and *with* reuse.

Development *for* reuse is a planned activity in a software process aimed at producing reusable components. Developing new reusable components needs

careful planning and consideration for reuse to be profitable. [28] To begin with, the reusable component has to have more than one reuser (i.e. reuse target). As a rule of thumb, Jacobson et al. [24] state that a reusable component has to be reused three to five times to recover the initial costs of creating it. The costs of creating and maintaining a reusable component are 1.5 to 3.0 times as much when compared to a component that is created for one use only [24].

If the development of a reusable component is justified, i.e. the cost issues are covered, three basic tasks have to be performed. First, the variability of requirements between the candidate reusers is analyzed. Second, an evaluation of the pros and cons of incorporating these requirements is made. Third, an optimum reusable component is designed and implemented with an appropriate level of generality to serve the purposes of as many reusers as possible. [28] The created component is then ready to be reused in development *with* reuse.

Development *with* reuse is a planned activity aimed at producing new software products with the help of reusable components [28]. At this stage the profits from *for* reuse development start to payoff. "It costs only one quarter as much to utilize a reusable component as it does to develop a new one from scratch." [24 p. 23]. Development *with* reuse includes at least three basic activities. First, suitable components are searched for and a set of components selected for further study. Second, candidate components are evaluated in order to find the most suitable one. Third, the selected component is adapted, if necessary, to fit the new requirements. [28] Naturally, not everything can be composed of reusable components, but product or context-specific parts also have to be created at this stage.

3.4 Reuse Approaches

Karlsson [28] classifies reuse approaches using three criteria: the *scope* of reuse, the *target* reuser and the *granularity* of reusable components. The classification is depicted in Figure 5 [28 p. 13].

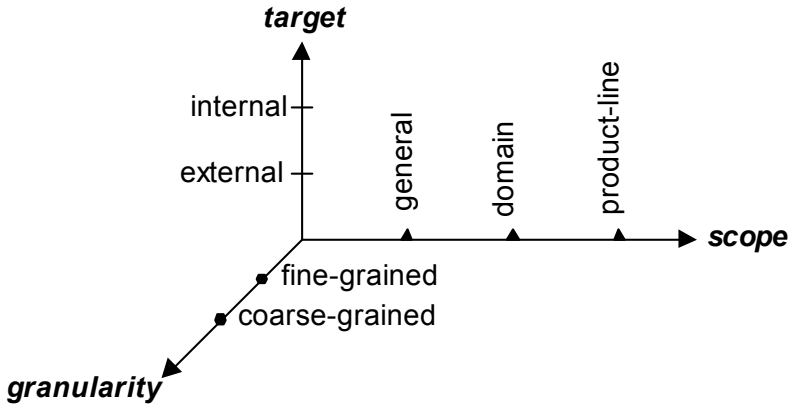


Figure 5. Classification for reuse approaches.

The *scope* axis in Figure 5 has three categories: general, domain and product-line. General reuse is domain-independent reuse. Meaning that components under this category are not tied to any specific application domain (e.g. telecommunication or information management). Examples of such general-purpose components are mathematical functions and database management systems. [28]

Domain reuse is domain-dependent reuse. In this category, the components are reused within a specific application domain. An example of such a component could be a protocol implementation used in several protocol stacks in the same domain. [28]

Product-line reuse is dependent on a specific application family [28]. Product line defines a generic domain architecture for an application family, from which different application systems are derived [26]. Reusable components are developed to either specialize or generalize this architecture. Because, the product-line reuse is strongly related to a specific application family, it is usually applied in-house only. [28]

The *target* axis in Figure 5 has two values: internal and external, meaning the internal and external markets of the company [28]. Regardless of the market approach, producing reusable components requires extra effort compared to specific solutions, thus resulting in higher financial costs as well. Therefore, reuse is viable only if the investment used in creating components is returned in some form after they have been deployed. [32]

The internal approach (also known as the in-house approach) means that the company is producing reusable components for its internal use only [28]. Examples of such activity could be found in the case of a large application family providing the same basic features in a range of customer products. Return on investment (ROI) is usually indirect, e.g. shorter time to market and higher quality of software products [32].

The external approach means that reusable components are produced for external use (i.e for another company). Attempts to develop reusable components in external markets have not been very successful, because for two prime reasons. First, if the components are general enough, they are likely to be very small and usually available for free from non-profit-making organizations, universities or even from programming language libraries. Second, if the components are very specific, the market base for them can not be very wide and it requires an enormous amount of interaction with the client to understand the functional and non-functional requirements for the component. [28] Nevertheless, if the ROI received from components is insufficient, components can also be coupled with services. For example, component pricing is moderate but their efficient use requires expertise, which is offered as a service. [32]

The *granularity* axis in Figure 5 has two classes: fine-grained and coarse-grained. Fine-grained components are usually generic and domain-independent, e.g. libraries of mathematical functions and object classes. Coarse-grained components are usually more specific and domain-dependent, for example, application subsystems and user-interface packages. [28]

The trade-off between fine-grained and coarse-grained components is not straightforward. Small components tend to have high reusability, i.e. they can be reused often and with minimum effort. However, the benefits achieved when reusing a small component is usually not very high and the real profits come from widespread reuse. [23] Large components, on the other hand, have lower reusability, i.e. they are reused rarely and they usually require more rework in order to meet the requirements of the new context [28]. Nevertheless, successful reuse of coarse-grained components can pay for itself even if the components are reused only once or twice [23].

3.5 Reuse Techniques

In order to promote reuse of components, several techniques have been identified to help in creating reusable solutions. To meet the common and differing requirements between reusers, the level of generality and adaptability of components requires planning [28]. General components may meet the needs of reusers without any changes, but at the expense of efficiency and excess functionality. Adaptable components may provide more specific and thus more powerful solutions, but they always require some changes before they can be reused.

Karlsson presents five general techniques that can be used to make general reusable components. The techniques are not orthogonal, therefore, the same generality can be represented by using more than one technique. The five techniques are as follows: [28]

- **Widening:** means identifying a set of requirements that are not contradictory and making a general component that satisfies all of them.
- **Narrowing:** means identifying functionality common to several reusers that can be represented by an abstract component.
- **Isolation:** means isolating different requirements to a small part of the system, while the rest of the system is constructed relatively independent of whatever specialization is chosen.
- **Configurability:** means making a set of smaller components that can be configured or composed in different ways to satisfy different requirements.
- **Generators**: means making a ”new” application-domain-specific language with which one can describe an application and satisfy different requirements. Application description can then be used to automatically generate executable code.

Jacobson et al. present seven techniques that can be used to create adaptable components. Component adaptations are realized at places where variations are possible. These places are called variation points. Jacobson et al. define following variability mechanisms and when they should be especially used: [24]

- **Inheritance:** is used when specializing abstract types or classes, or adding some operations while keeping others.
- **Extensions:** is used if parts of a component need to be extended with additional behavior by attaching several variants at each variation point at the same time.
- **Uses:** is used when creating a specialized use case from an abstract one.
- **Configuration:** is used when choosing alternative functionality and implementations.
- **Parameterization:** is used when there are several small variation points for each variable feature.
- **Template instantation:** is used in case of type adaptations or selecting alternate pieces of code.
- **Generation:** is used when doing large-scale creation of one or more types or classes from a problem-specific language.

All in all, the difference between the techniques presented by Karlsson [28] and Jacobson et al. [24], is a slender one. Karlsson describes reusability as useful generality [28 p. 257], whereas Jacobson et al. present variability techniques as ways to generalize and specialize abstract components [24 p. 100]. Karlsson states that it is better to make a general component that can be adapted to reuser needs than to make a generic one without the possibility of adaptations. However, the cost of possible adaptations has to be balanced against the cost of creating a generic component. [28] Hence, components should be generic enough to be reusable, but also offer variability to support adaptability to different contexts.

3.6 Reuse Metrics

Software metrics are used to assist management by quantifying the software development and maintenance process. The use of metrics will improve the software process in terms of cost estimation, planning, productivity, quality, customer confidence and overall controllability of the process. [34] In an organization practicing reuse, the traditional metrics may not always provide

correct results. For instance, measuring the productivity of a reuse organization by using the produced lines of code as a metric may indicate that productivity is very low, when in fact it can be very high. This is because the organization may not produce new lines of code but instead reuse the old ones. [28].

Poulin and Caruso state that reuse metrics have three roles: quantifying reuse, encouraging reuse and standardizing reuse methods. Quantifying reuse is very close to the traditional role of software metrics and when it is applied in a meaningful and standardized way, it encourages practice of reuse in the software organization. [35]

Poulin and Caruso present seven reuse metrics that are based on observable data elements in the source files of a product. However, only two of them have any relevance in terms of the derived metrics usable in the case study. The two metrics are [35 p. 156]:

- Shipped Source Instructions (SSI). The number of non-comment source code instructions, not including the Reused Source Instructions (RSI), in the source files. SSI are the newly implemented lines of code and the call to a reusable part counts as one SSI.
- Reused Source Instructions (RSI). The number of lines that are not developed or maintained but included in the source files.

The presented metrics are combined to form three derived reuse metrics applicable in test reuse: Reuse Percent, Additional Development Cost and Development Cost Avoidance. Reuse Percent indicates the portion of a product, product release or organizations effort that can be attributed to reuse. Additional Development Cost is used to measure the extra costs of developing *for* reuse, whereas Development Cost Avoidance is used to measure the savings of developing *with* reuse. [35] The three metrics are as follows [35 p. 158–160]:

- Reuse Percent of a product is:

$$\text{Reuse Percent} = \frac{\text{RSI}}{\text{RSI} + \text{SSI}} \times 100\% \quad (1)$$

- Additional Development Cost (extra cost of *for* reuse) is:

$$\text{Additional Development Cost} = (\text{RCWR} - 1) \times \text{RSI} \times (\textit{New Code Cost}) \quad (2)$$

Development Cost Avoidance (avoiding costs by reusing) is:

$$\text{Development Cost Avoidance} = (1 - \text{RCR}) \times \text{RSI} \times (\textit{New Code Cost}). \quad (3)$$

RCWR in equation (2) stands for Relative Cost of Writing Reusable Software (RCWR). RCWR is the cost of writing reusable software compared to the cost of writing software that is not to be reused. If writing software for one time use only takes one unit of effort, then the portion of that effort that it takes to write reusable software is called RCWR. [35] The cost of making software reusable is caused by making components generic and adaptable, extra documentation, maintenance and support, etc. [28].

RCR in equation (3) stands for Relative Cost of Reusing Software and it describes the effort of reusing a component compared to the effort of creating a similar component from scratch. If developing a new component from scratch takes one unit of effort, then the portion of that effort that it takes to simply reuse a similar component is called RCR. The cost of reusing software (i.e. development *with* reuse) is caused by such activities as: finding, understanding, selecting, evaluating, etc. reusable components. [35]

RCR and RCWR values are usually based on long-term knowledge, measurements and observations of the development process and the changes that introducing reuse to the process has resulted in. Usually the values are rough estimates, especially when the software development organization has not had a defined process model before introducing reuse. Jacobson et al. estimate the value of RCR to be around 0.25 and the value of RCWR ranging from 1.5 up to 3.0, in general [24]. Favaro reports that RCR varies between 0.1 and 0.4 and RCWR from 1.0 to 2.2 depending on the complexity of the component [36]. According to Lim's study at Hewlett Packard, RCR is 0.19 and RCWR 1.11, on average [37]. Bardo et al. studied reuse in a very homogeneous development environment where the RCR was as little as 0.05 and RCWR was also very low, ranging from 1.15 to 1.25 [38]. Poulin and Caruso recommend using an RCR value of 0.2 and an RCWR value of 1.5 [35].

In the case study the *New Code Cost*, found in equations (2) and (3), is treated as an unknown constant. This is because it was not applicable in the scope of this study to try to determine the cost for 1 LOC, or the cost of 1 KLOC for that matter, created without reuse. Therefore, in the following equations *New Code Cost* is marked with the abbreviation *NCC* and the amount of source code lines with the abbreviation *LOC*.

Based on equation (3), another equation for describing the development cost of *with* reuse (instead of development cost avoidance) can be derived:

- Development cost (cost of *with* reuse) is:

$$\text{Development Cost} = \text{RCR} \times \text{RSI} \times \textit{NCC}. \quad (4)$$

Since the *New Code Cost* is treated as an unknown constant in the case study, three comparable equations can be derived for total development costs without reuse, *for* reuse and *with* reuse based on equations (1), (3) and (4).

- Development cost without reuse (WR) is:

$$\text{WR} = \textit{LOC} \times \textit{NCC} \quad (5)$$

- Total development cost *for* reuse is:

$$\text{Total cost } \textit{for} \text{ reuse} = \text{WR} + (\text{RCWR} - 1) \times \frac{\text{RSI}}{\text{RSI} + \text{SSI}} \times \text{WR} \quad (6)$$

- Total development cost *with* reuse is:

$$\text{Total cost } \textit{with} \text{ reuse} = \left(1 - \frac{\text{RSI}}{\text{RSI} + \text{SSI}}\right) \times \text{WR} + \text{RCR} \times \frac{\text{RSI}}{\text{RSI} + \text{SSI}} \times \text{WR}. \quad (7)$$

The total development cost *for* reuse is comprised of the cost of the whole software product as if it would have been produced without reuse and the extra cost for those parts that are made reusable. The total development cost *with* reuse includes the cost of specific and reusable parts. When the total development cost *for* or *with* reuse is compared to (i.e. divided by WR) development cost without reuse, the unknown constant for *New Code Cost* is tailed off.

4. Introduction to TTCN-3

TTCN-3 is a test specification and implementation language developed at the European Telecommunications Standards Institute (ETSI). TTCN-3 supports all types of black box testing of local or distributed, and reactive systems. [4, 39] TTCN-3 was developed on the basis of two versions of Tree and Tabular Combined Notation (TTCN and TTCN-2) that were confined only to protocol and conformance testing. TTCN-3 offers the well-proven basic functionalities of TTCN and TTCN-2, which have been included directly or enhanced in TTCN-3. However, from a syntactical point of view TTCN-3 differs a great deal from its predecessors. TTCN-3 is based on a textual core language providing interfaces to different data description languages, making it language independent. [39]

TTCN-3 was developed to answer the need for a unified test notation in the area of telecommunication industry and science [4, 40]. As it has turned out, the use of TTCN-3 has spread to other areas of industry and science (e.g. automotive and software engineering), because of its applicability to all types of black box testing. [39]. Typical areas of application for TTCN-3 are protocol testing, module testing, and testing of CORBA based platforms etc. [4].

TTCN-3 is an abstract test specification language, meaning that the tests are implementation independent and can therefore be reused for different implementations. The abstract level of specification also makes it possible to create standardized test suites to increase the probability that the implementations of different vendors actually interoperate. In addition, TTCN-3 is not restricted to functional testing alone but is applicable to other types of testing as well, e.g. performance, scalability, load and interoperability. [41]

Exploitation of TTCN-3 at Ericsson has shown that TTCN-3 is a reliable and efficient even in the most challenging test scenarios. The results also indicate that TTCN-3 provides a high degree of automation that has a positive impact on regression testing. [39]

4.1 Core Language and Presentation Formats

From a syntactical point of view, TTCN-3 has changed drastically in comparison to TTCN and TTCN-2, providing a flexible, powerful and unified solution to test all types of reactive systems [4, 39]. The syntax of the TTCN-3 core language has a look and feel similar to other programming languages (e.g. C, Java) and should therefore be easy to learn and understand for someone with programming experience [39, 42].

The top-level entity of TTCN-3 is a module that contains a definition part and an optional control part. Modules can import definitions from other modules and they can be parameterized. [41] In [43 p. 21] the language elements of TTCN-3 are divided into several categories as illustrated in the Figure 6.

The data types can be either predefined types determined in the TTCN-3 standard [4] (e.g. integer and char) or user-defined types (e.g. record and set). Test data consists of constants, variables and template declarations. [43] Templates provide the possibility to specify, organize and structure test data. Templates can either be used to define specific values to be transmitted or describe conditions to be matched by the received values, using matching mechanisms. [39]

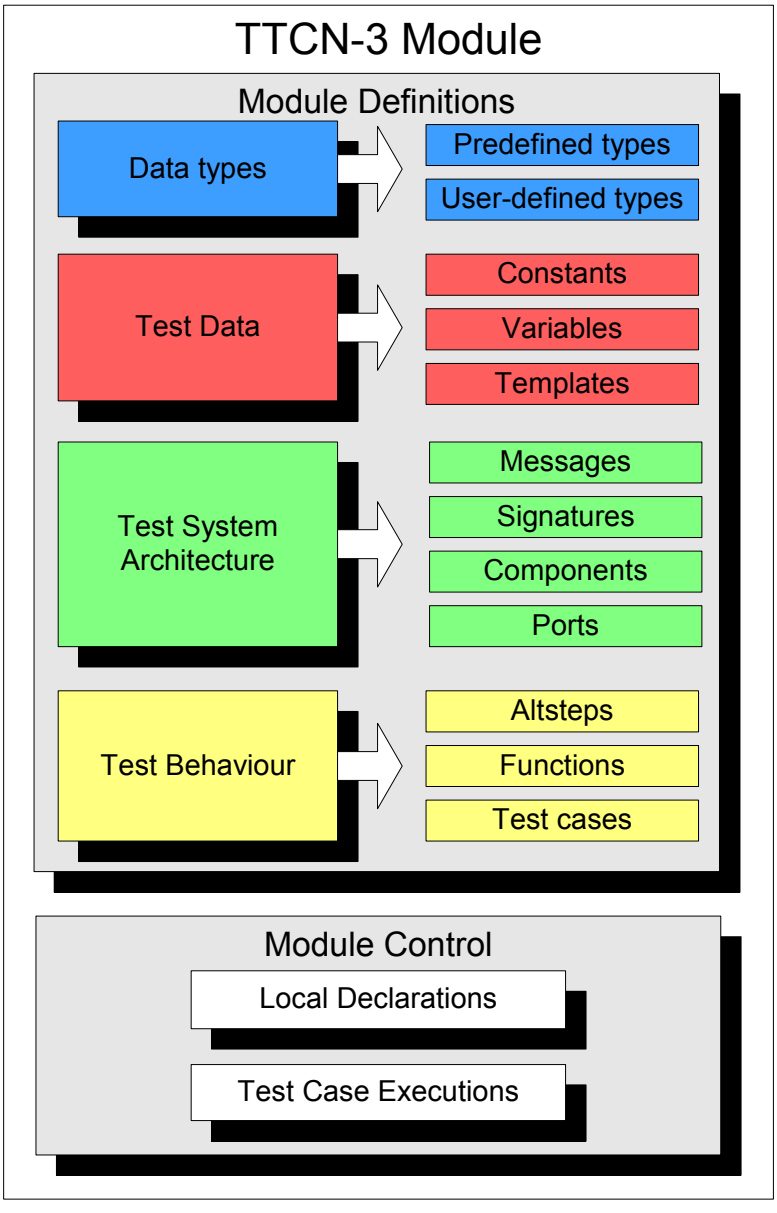


Figure 6. TTCN-3 language elements.

Messages and signatures are used for communication over the communication ports by means of message exchange and procedure calls [4]. Hence, the ports are message-based, procedure-based or mixed (i.e. message- and procedure-based) and they are directional. Each port may have an **in**, **out** or **inout** list; for

in, out and both directions, respectively. The test component in TTCN-3 is an instance of corresponding component type definition. The type definition declares constants, variables, timers and ports owned by an instance of that type. [39]

Altsteps are function-like descriptions that are used for structuring component behavior. Altstep has special semantics used to define an ordered set of alternatives. In TTCN-3, functions can be used to structure computation or to calculate a single value, similar to other programming languages. [39] Test cases are a special kind of function that are executed in the module control part in order to check whether the SUT passes the test or not. [4]

The control part of the TTCN-3 module describes the execution order of the test cases. Hence, it can be seen as the main program of the TTCN-3 module. [39] The control part can also contain local declarations e.g. variables [4].

The code example in Figure 7 illustrates the language elements in the TTCN-3 core language. In this example, the reserved words for the TTCN-3 language are printed in **bold-face**.

Figure 8 illustrates how TTCN-3 is built based on textual core language providing interfaces to various data description languages and the option to select from different presentation formats [4]. The tabular presentation format of TTCN-3 is designed for users who are familiar with the predecessors of TTCN-3, TTCN and TTCN-2. The graphical format is based on MSCs and it eases the understanding and representation of test execution and analysis of the test results. [39]

```
module MyModule { /******** DEFINITION PART ********/
  type integer MyInteger;           // predefined type
  type record MyRecord {            // user defined type
    MyInteger myInt,
    char myChar
  }
  signature MyRemoteProcedure();
  const charstring MYCONST := "My string";  // constant declaration
  var bitstring := '0001'B;                 // variable declaration
  template MyRecord MyTemplate := {         // template based on record type
    myInt := 2,
    myChar := "a"
  }
  type port MyMessagePort message {
    inout MyRecord            // message port with inout direction
  }
  type port MyProcedurePort procedure {
    out MyRemoteProcedure     // procedure port with out direction
  }
  type component MyTester {   // Test component definition
    port MyMessagePort PCO1;
    port MyProcedurePort PCO2;
    timer T1;                 // Timer declaration
  }
  type component TSI {        // Test System Interface definition
    port MyMessagePort PCO1;
    port MyProcedurePort PCO2;
  }
  altstep default_altstep() runs on MyTester {
    [] any port.receive {     // first alternative
      setverdict(fail);
    }
    [] T1.timeout {           // second alternative
      setverdict(inconc);
    }
  }
  function myFunction() runs on MyTester {
    PCO1.send(MyTemplate);
    T1.start;                 // start timer and activate altstep
    var default my_altstep := activate(default_altstep());
    alt {
      [] PCO1.receive(MyTemplate) {
        setverdict(pass);
      }
    }
    deactivate(my_altstep);   // deactivate altstep
  }
  testcase MyTestCase() runs on MyTester system TSI {
    map(mtc:PCO1, system:PCO1);    // mapping the mtc (MyTester) to
    map(mtc:PCO1, system:PCO1);    // system (TSI - Test System Interface)
    myFunction();                  // function call
    : // test case actions
    unmap(mtc:PCO1, system:PCO1);  // unmapping connections
    unmap(mtc:PCO1, system:PCO1);
  }
/******** DEFINITION PART ********/

control {  /******** CONTROL PART ********/
  var boolean MyVariable;
  execute (MyTestCase());
}
/******** CONTROL PART ********/
}
```

Figure 7. Example of the core language syntax.

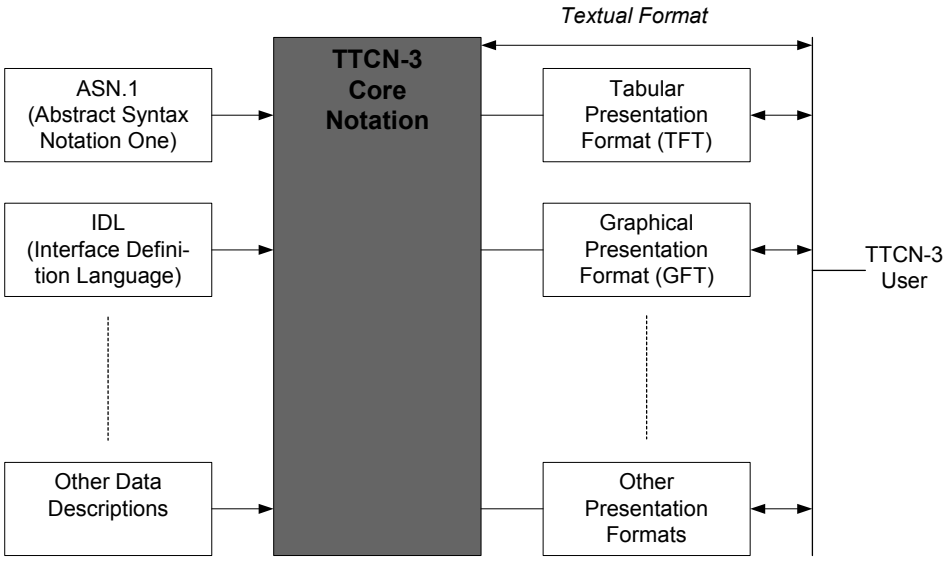


Figure 8. Core language and various presentation formats.

4.2 Test System and Execution Interfaces

The TTCN-3 test system is depicted in Figure 9 [43]. A test system is a set of test components, ports, specific interfaces and the SUT. Every test system has one Main Test Component (MTC), which is created automatically at the beginning of a test case execution and optional Parallel Test Components (PTC), which can be created, started and stopped dynamically (i.e. during test execution). MTC termination ends the test case execution. Test components can be interconnected using *connected* communication ports and connected to the SUT using *mapped* ports. The communication port's *in-direction* is modeled as an infinite FIFO queue, where the incoming information is stored until it is processed by the test component that owns the port. In case of overflow, the test case results in error. The *out-direction* is unbuffered. [39]

TTCN-3 test system has two interfaces: the abstract and the real test system interface. The abstract test system interface can be seen as a collection of ports defining the abstract interface to the SUT. Port mapping between a test component and the abstract test system interface is in fact a mere name translation determining the means of referring to communication streams.

Components, ports and the abstract test system interface are application independent, whereas the real test system interface is application specific, implementing the real interface to the SUT. [39]

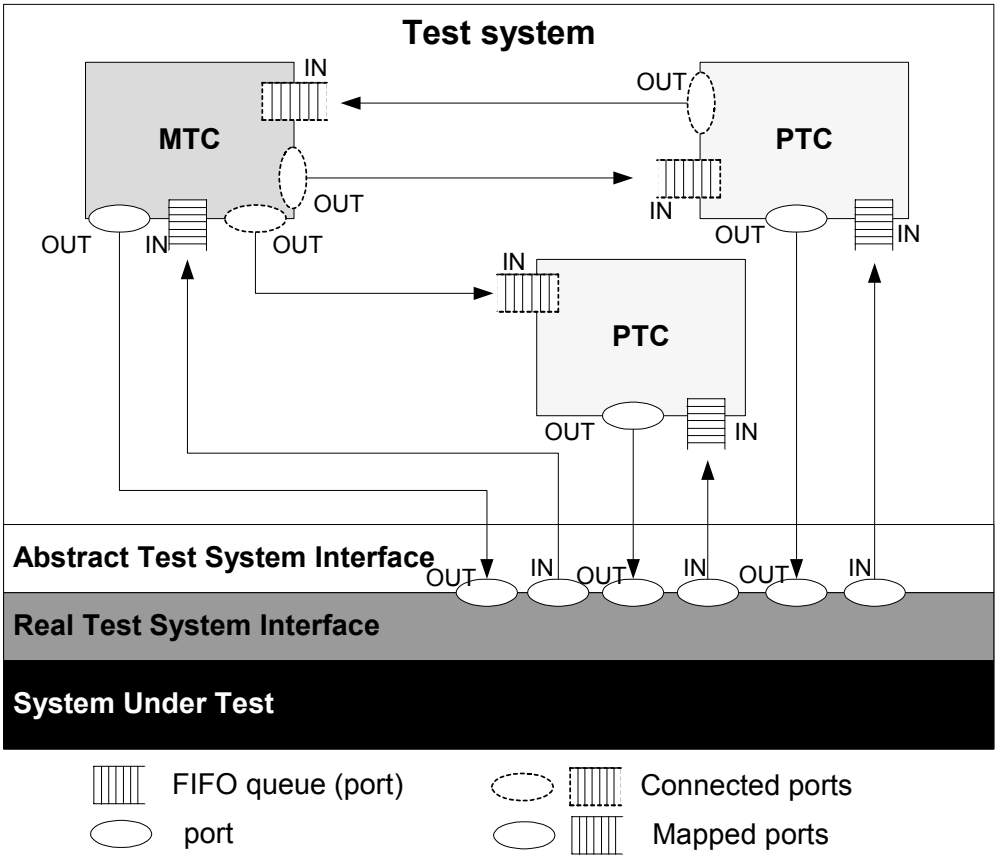


Figure 9. Conceptual view of the test system.

TTCN-3 offers two execution interfaces: TTCN-3 Runtime Interface (TRI) and TTCN-3 Control Interface (TCI), illustrated in Figure 10. They specify a standardized adaptation of the test system for management, communication, component handling, external data representation (with encoding and decoding) and logging for local and distributed test setups. These interfaces make TTCN-3 executable independent of the SUT, processing platform, implementation language, etc. [39] Figure 10 depicts an overall view of a TTCN-3 test system architecture. The test system can be seen as a group of interacting entities, where each entity has a specific well-defined functionality in the test system. [44]

The TTCN-3 Executable (TE), also referred to as Executable Test Suite (ETS), and depicted in green in Figure 10, is responsible for the interpretation and execution of TTCN-3 modules [45]. Within TE various structural elements can be separated: control, behavior, components, types and data, ports and timers [41]. These structural elements represent functionalities that are defined within the TTCN-3 module (e.g. control represents the control part within the TTCN-3 module) or by the TTCN-3 core language standard [45]. However, this refinement of TE is merely a conceptual aid used to define the TTCN-3 test system interfaces. The TTCN-3 module is implemented on an abstract level in TE and it is the task of other entities (TM, CH, CD, SA and PA) to make these abstract concepts concrete. Figure 10 clearly presents how the TTCN-3 Executable (TE) is surrounded by the TRI and TCI interfaces that enable the adaptation of TE to different contexts. [39, 45]

Test Management and Control (TMC), depicted in red in Figure 10, includes functionality related to management of test execution, handling of components (CH) and encoding and decoding (CD) of values. Test Management (TM) is responsible for the overall management of the test system. [45] Two functionalities can be separated within the TM, those related to the test execution control (TC) and those related to test event logging (TL). TC is responsible for the proper invocation of the TTCN-3 modules and after the initialization of the test system, test execution starts within the TC. The TL entity maintains a test log as it is explicitly notified by the TE to log test events. [44]

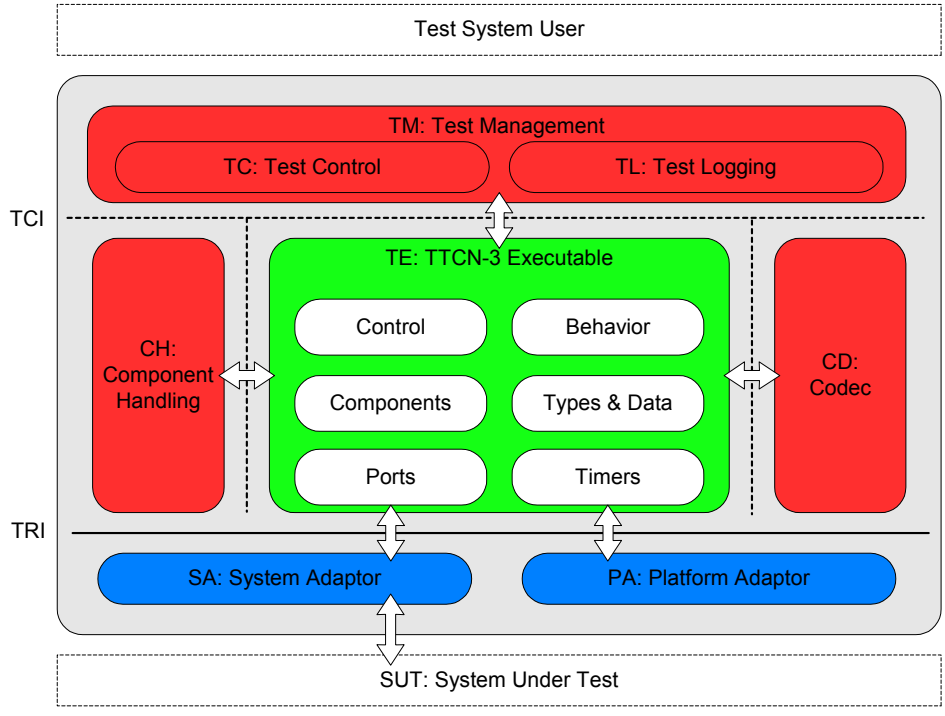


Figure 10. Overall test system architecture.

The TE can be executed in a centralized, i.e., on a single test device or distributed manner, i.e., on a multiple test devices [39]. In the case of a distributed test system (TE is distributed between several test devices) Component Handling (CH) implements and synchronizes communication between distributed test system entities. Encoding and decoding of TTCN-3 values is the responsibility of the Coding and Decoding (CD) entity. TE selects the appropriate codec and sends the TTCN-3 values to the CD to be coded. In the same manner TE selects the appropriate decoder in order to decode the valid TTCN-3 values from the received data. [45]

The System Adaptor (SA), depicted in blue in Figure 10, is responsible for the adaptation of message and procedure-based communication with the TTCN-3 test system and SUT. The Platform Adaptor (PA), also depicted in blue in Figure 10, implements TTCN-3 external functions and realizes timers that are instantiated in the TE. [39, 44]

5. Test Reuse

Demands for software testing are ever-increasing as the size and complexity of software systems are growing every day and the markets demand new products and higher quality in shorter time frames. Improving the efficiency and effectiveness of testing through test reuse could provide remarkable savings. [22] Section 3.1 introduced some of the potential of software reuse, such as reductions in development and maintenance costs and improvements in quality and productivity. It is fair to say that these promises can be related to test reuse as well. Perhaps the expectations for test reuse can be even higher than those for software reuse. In addition to reusing tests in testing different products or product releases, tests can also be reused between different testing levels (e.g. component and integration) and types (e.g. functional and load). [22]

Just as with software reuse, testware reuse can be divided into *for* and *with* reuse sides. Reusable tests are developed *for* reuse with an appropriate level of generality and adaptability. In development *with* reuse, the tests are reused with necessary adaptations and complemented with context- or product-specific tests. It is important to note that, as with software, not all testware is meant to be developed *for* reuse, and usually it is impossible to cover every part of the test suite by just reusing tests. Furthermore, the reuse of testware is bound to the reuse of software in many cases. [46]

Testware differs from other software merely due to its specific purpose of finding errors from the IUT or validating that the IUT functions as it was supposed to [47]. Therefore, one can conclude that the same set of techniques used to create reusable software, presented in section 3.5, could be used in developing testware *for* reuse. However, unlike the *with* reuse side of software development, which is mainly interested in building something new, the *with* reuse side of test reuse is interested in "tearing down". This means that, regardless of the reusability, the purpose of testware is to find errors or to give some level of assurance that the software meets its specification. [46]

When considering the reuse possibilities of testware, one must always take into account the reusability of the software as well. This means that, if the software itself shows very few possibilities for reuse, it narrows down the possibilities of reuse for testware as well. However, some exceptions exist, e.g. combining an

abstract test specification language such as TTCN-3 and standardized test suites allows us to create reusable testware without any ties to software implementations. [46]

5.1 Three Viewpoints of Test Reuse

Reuse of tests offers a wide range of reuse possibilities and viewpoints. Tests may be reused during the development process (vertical viewpoint) of the software product, between similar products (horizontal viewpoint) in a product family or application domain and between different product versions (historical viewpoint). [46]

During the development process, the same tests may be reused between different testing levels and types. E.g. component tests are reused in integration testing and functional tests are reused in testing performance and scalability. A product family may have similar products composed of the same components and features e.g. mobile phones from the same vendor. Therefore, the same tests could be reused to test these shared components and features, and new tests would only be needed to verify the product specific parts. [46] It is common for software products to evolve. New versions usually introduce new features, but retain most of the old ones. The old features could be tested reusing the tests developed for these features, whereas the new features require new tests. However, as Smith [Smith] has discovered, even the new features may have some relations to old features or some times use the same functions as the old features do. Hence, depending on the granularity of the reusable tests some parts may be reused. Reusability is therefore an obvious approach in the development of test software. [46]

5.1.1 Vertical Reuse

Vertical reuse in this thesis concentrates on the reuse possibilities between different testing levels. Schieferdecker and Vassiliou-Gioles [40] have briefly contemplated the idea of using the same tests between different types of testing, e.g. parts of tests developed for functional testing could be reused for performance or scalability testing.

In the past, software test engineers developed component tests for an application using an *ad hoc* approach. In this approach, the primary concern was to develop simple upper testers and lower testers that would help with testing the software component against the given requirements. Tests created in this way were, and still are, difficult to reuse on other testing levels. However, the point of view was not to reuse them at all [48].

Vertical reuse demands that we identify the similarities between different testing levels and identify those tests that have the potential to be reused and the potential to reveal errors on different levels. Vertical reuse is illustrated in Figure 11. For example, tests developed for component testing may be reused in integration and system testing. This pattern can be applied in reverse order as well: tests designed for system testing can be utilized in the lower level tests.

Vertical reuse has three evident virtues: narrow domain, low variability of problems and static interfaces. From a vertical viewpoint, the software product itself is the domain and the reuse possibilities are found by analyzing the similarities between testing levels. Narrow domain has been one of the key success factors in fortuitous industrial cases [27]. Variability of problems and solutions pose no issue, since the problems are the functions, features and characteristics we test and the solutions are their respective tests. During the testing process, the interfaces of the components are likely to remain unchanged, thus adaptations required to test scripts are likely to remain very moderate.

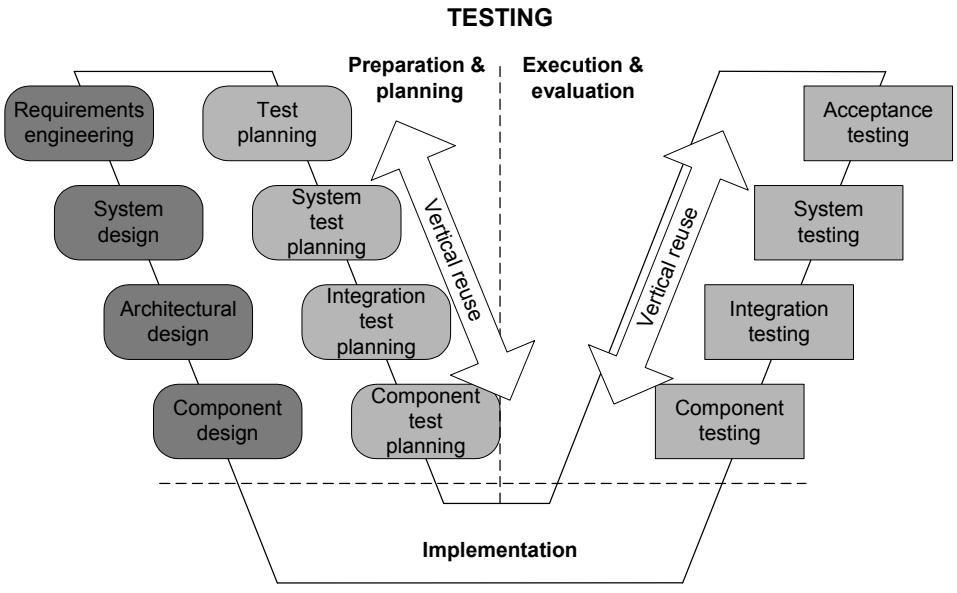


Figure 11. V-model. In vertical reuse test designs and implementations are reused in different levels of the testing process.

5.1.2 Horizontal Reuse

The horizontal viewpoint means reusing tests between various products. Horizontal reuse generally requires that products are mostly composed of reusable software components and therefore likely belong to a certain product family. This means that some parts of the product specifications are identical and form a solid basis for development of reusable tests. [46] These tests are developed to accompany the reusable software components, thus maximizing the profits of reuse [23]. In addition, this ensures that testing keeps up with the pace of shorter development times that the software reuse itself promises [22].

It is also possible to identify similar features of different products even though the products may not be composed of reusable components or are not part of the same product family. In this case, more effort is needed to identify the similar and differing elements of the products to be tested, since it is not done during the development process. Therefore, this option is not a feasible one. [46]

Furthermore, one example of the horizontal viewpoint is to implement the software components according to some commonly approved (used to solve common use case problems) or standardized specifications, which are typical in the telecommunication domain. Even though the standard specifications are the same, the implementations may vary. Reusing tests that are developed on the basis of these standard specifications helps to construct conforming and interoperable implementations. [23]

Horizontal reuse is probably most profitable if the tests can be developed to accompany reusable components or on the basis of standardized specifications as illustrated in Figure 12 [46].

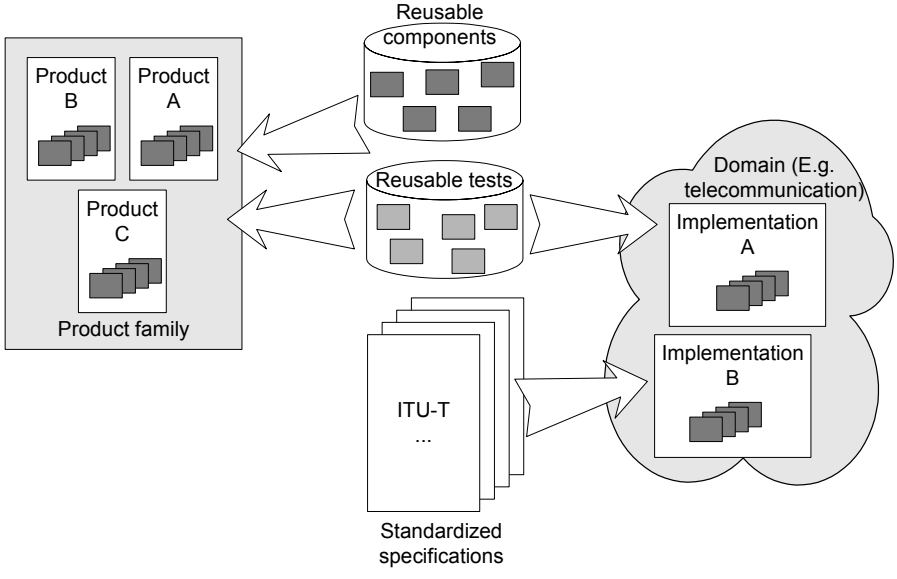


Figure 12. Applicable approaches for horizontal reuse where tests are reused between products and implementations.

5.1.3 Historical Reuse

The historical viewpoint addresses test reuse between product generations. Figure 13 illustrates software and test reuse from the historical viewpoint. As is clearly noticeable from the figure, historical and horizontal viewpoint are very much alike. “Different products” in horizontal reuse are “product versions” in

historical reuse. Test suites in Figure 13 contain reusable tests and product-specific tests (i.e. they are not made for reuse). A typical practical case is when a new version(s) of a product should fulfil the same basic requirements that the earlier products satisfied. New tests may only be needed for verifying the new features or components of the new product version. [46]

It is a well known fact that reuse is always a long-term investment. Therefore, historical reuse is a natural approach to reuse tests. As illustrated in Figure 13, reusable parts increase their share of the overall portion in products and in test suites over time, thus increasing the profits of reuse as well. [46]

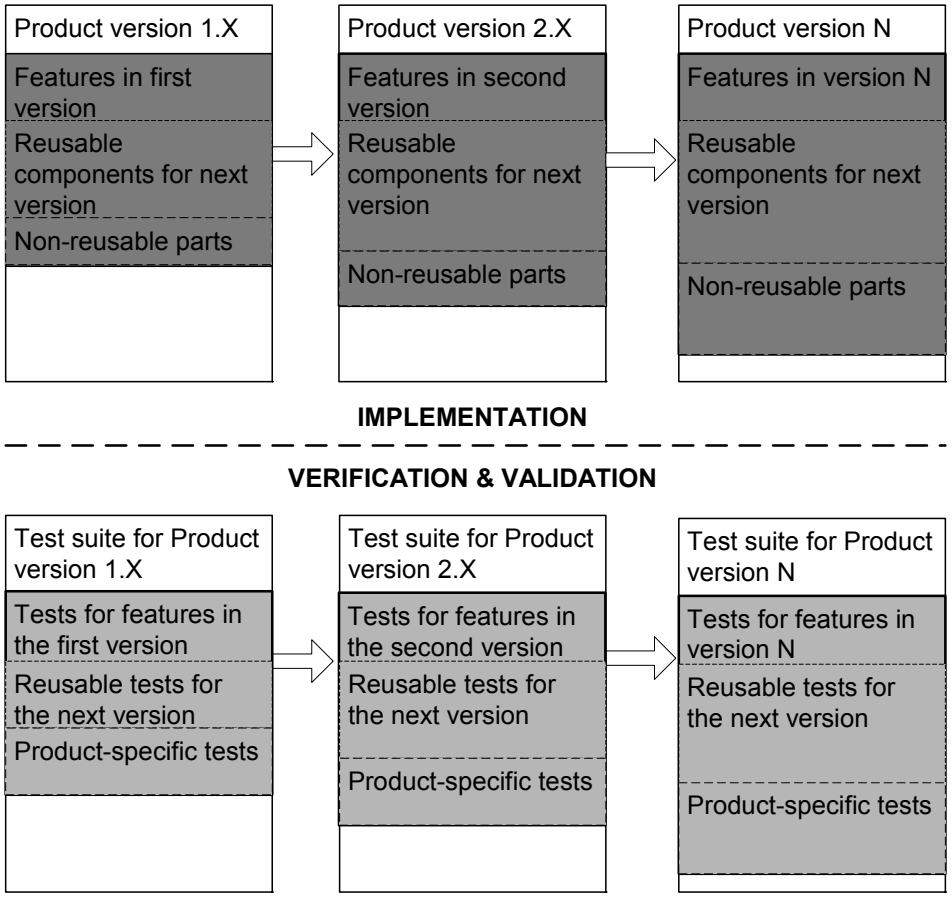


Figure 13. In historical reuse tests are reused between product implementations.

5.2 Past Studies

Reuse of tests, although clearly a promising area of study, has not been subject to a wide scope of research [46]. This section presents some of the past studies on test reuse and compares them to the work presented in this thesis.

ETSI has published a technical report [49] that provides guidance in creating reusable Abstract Test Suites (ATSs). The focus of this report is clearly on conformance testing, and it is aimed at standardization bodies and organizations involved with conformance testing. Rules and their associated examples are implemented using TTCN. Compared to the issues covered in this thesis, the scope is narrower, however, the ideas and groundwork are more specific and laid out in greater detail. Being a technical report, it does not present any case studies where the rules and examples are applied in practice and thus, no real implications of their applicability are provided.

Korhonen et al. [13] explore the possibility of speeding up the testing process for configured software products by using a feature-based testing approach. In this approach, existing tests and product feature descriptions are used to select, modify and configure tests. However, their approach focuses on systems that apply features for defining their properties or are at least composed of reusable software components. In comparison to this thesis the scope of test reuse is narrower, however, their study also presents tools and describes how test reuse effects the testing process and test infrastructure. A case study is also demonstrated, but the results and profits of test reuse are only vaguely presented.

Smith [50] presents an approach to code reuse where complete tests are assembled using reusable high-level operations. The idea is to select a product feature to be tested and then divide this feature into a sequence of test operations. Obviously this enhances the code reuse in testing a product that has a great deal of “common” functions (e.g. word processing; open file, save file) used to implement various use cases by means of function combinations (e.g. open file, write a text block, edit and save file). [50] Smith states: “Code reuse in automated software testing is highly effective when the scope is limited to the product under test.” [50 p. 2]. In addition, this enables reuse of the same test operations when testing a new product version with similar features implemented using the same “common” functions. Similar to Korhonen et al. the

focus is on software systems that are constructed based on features implemented by using reusable functions. Smith presents examples that support the use of his approach and rather impressive results of extensive research. However, in Smith's approach, tests are not really developed *for* reuse but, as he states: "Each time we had a new functional area of the product to test, we tried to wedge it into an existing driver or we wrote a new test driver, usually replicating code from existing drivers". There are two drawbacks to this approach. Wedging does not sound like a planned activity that would lead to good results in the long run and more importantly, replicating code leads to the maintenance of two similar components, which is not desirable either.

Hörnstein and Edler [51] approach test reuse in CBSE by using built-in tests (BIT). They base their views on mechanical and electronic engineering, where BITs are used commonly. Hörnstein and Edler state that because components are often used as black boxes and they usually work as state machines, their testability is low. To improve the verification and validation process components should be supplied with BITs, testers to exercise the BITs and evaluate the information, and handlers to manage errors. Clearly this approach tightly binds tests to the component, thus limiting the scope of reuse. In addition, using BITs makes the code more complex.

When compared to the earlier studies mentioned above, this thesis exploits techniques known from software reuse more extensively. In addition, guidelines with illustrative examples are created to improve TTCN-3 test reuse and a case study is provided, in which the use of guidelines is studied, and the level and profits of test reuse are determined.

6. Guidelines for Reuseable TTCN-3 Code

This and the following chapters present the work performed in this thesis. This chapter presents guidelines for reusable TTCN-3 code. The guidelines are created primarily for this thesis. However, most of them have also been published in [46] which is a project deliverable meant only for the project partners of TT-Medal. Nevertheless, after this restricted publication in July 2004, all the guidelines have been under constant rework by the author. The goal has been to cover more aspects of test reuse and to increase the practicality and understandability of the guidelines. The biggest impact on this evolutionary work was caused by a case study presented in chapter 7.

6.1 Background

This section explains the underlying factors that were the inspiration for the guidelines and summarizes these factors and guidelines in a table format. The underlying factors were: the reuse techniques presented in section 3.5; the test suite and test case structures explained in section 2.6; the TTCN-3 test system characteristics illustrated in section 4.2 and the language features of TTCN-3. Table 2 summarizes the factors and guidelines.

The software reuse techniques presented in section 3.5 had the biggest impact when trying to come up with practical guidelines for the TTCN-3 language. As it turned out, the techniques for creating generality were nearly as inspiring as those for creating adaptability. This was somewhat surprising, as during the literature review the concept of generality in test reuse was not as promising as that of adaptability. Perhaps this was influenced by the traditional belief of seeing testware as being created for some specific implementation, where there is no room for generalization. Nevertheless, it should be noted that the presented techniques for generality and adaptability are not that different. Hence, they have, in some cases, been an inspiration for the same guideline.

The test suite and test case structures presented in section 2.6 inspired the creation of guidelines for modularity within modules, between modules and within test cases.

Table 2. Summary of the factors and guidelines.

Factor \ Guideline	1	2	3	4	5	6	7	8	9	10
Reuse Technique										
Generality										
Widening						X				
Narrowing							X			
Isolation										
Configurability	X	X	X				X			
Generators										
Adaptability										
Inheritance							X			
Extensions				X						
Uses				X						
Configuration			X	X		X				
Parameterization					X	X				
Template instantation							X			
Generation										
Test suite or test case structure			X						X	X
TTCN-3 test system characteristic	X	X								
Language feature								X		

The TTCN-3 test system characteristics presented in section 4.2 provided some inspiration, especially when it was considered essential that the tests should follow the implementations. Although this has a somewhat debilitating effect on generality, it is applicable, especially from the vertical viewpoint.

TTCN-3 language features could be seen as a driving force behind any of the guidelines. However, they are marked as a factor in Table 2 only if the guideline could not have been implemented in any other language and the other factors played a lesser role.

6.2 Overview and Motivation

This section provides an overview of the guidelines and the motivation for why they should be used. The relations between factors and guidelines presented in Table 2 in the previous section is also briefly discussed.

Guidelines 1 and 2 were based on the idea of configurability in a conceptual TTCN-3 test system. Use of functions and test components provides a mechanism to encapsulate test behavior. The reusable part in these guidelines is the combination of the test component and the function running on it. Functions and components can be combined quite freely, which facilitates reuse.

Guidelines 1 and 2 present four approaches for test system configuration in component and integration testing. Both of these guidelines utilize the concepts of upper tester (UT) and lower tester (LT). In protocol testing, UT(s) provides control and observation of the upper service boundary of the IUT, and LT(s) provides indirect control and observation of the lower service boundary [19]. In software component testing, UT(s) simulates the calling component (and possibly the called component as well), and LT(s) the called component of the IUT. Every time a component is integrated into a new context, one of its interfaces may constitute an interface of a new component while the other is not visible anymore. Therefore, the testers (i.e the combination of a test component and a function running on it) linked to the visible interface can be reused.

Figure 9 (p. 43) presents the conceptual view of the test system. This conceptual view is the basis for the test system configuration in figures (Figure 14, Figure 16, Figure 18, Figure 20, Figure 22, Figure 24) presented in guidelines 1 and 2. MTC and PTC(s) are depicted as boxes and ports as ovals. MTC is named “MTC” and PTC(s) have “UT”, “LT” or both written in. They represent the control and observation purpose of the function running on the respective PTC. SUT is surrounded by a white box representing the Abstract Test System Interface and a grey box representing the Real Test System Interface.

Guideline 3 had its foundation in the test case schema presented in Figure 4 (p. 25). But configurability and configuration (nearly the same idea) also had an impact. The test case parts in Figure 4 should be encapsulated in functions or function groups so that they can be easily reused and configured in different ways to

increase coverage of testing. Guideline 3 presents an approach more applicable to the vertical viewpoint, whereas guideline 4 focuses on horizontal and historical viewpoints.

Guideline 4 was based on the idea of use cases. If use cases provide a basis to identify and structure commonly used functionality within one or several implementations, they could also provide a way to identify reuse and variation possibilities for testware. Reusable high-level functions based on use cases could be used to quickly create new test cases. High-level functions can also act as a variation point hiding the variations between the test suites for different products.

Guideline 5 naturally had its origins in parameterization. Parameterization is probably the most effective and simplest of the presented reuse techniques. When applied to test-ware it can provide multiple possibilities for reuse and improve the coverage of testing.

Guideline 6 was based on the idea that some parts of a test case could be implemented in a way that would fulfill all the needs of the reusers. This means that a test case is a reusable general component that provides the possibility to select the appropriate parts. Selection was realized by means of selection structures and parameterization.

Guideline 7 is a combination of several different ideas. First of all, using a common type definition or template modification could be seen as techniques of narrowing or inheritance. Second, combining structured types is a type of configurability. Third, template instantiation means use of templates. This guideline encourages the use, reuse and combination of common type definitions and modification of templates. This eases the maintenance of tests and provides later design decisions.

Guideline 8 was purely based on the use of wildcards, which is a TTCN-3 specific language feature. Wildcards prevent overspecialization of any parameterizable language elements, thus their use has the potential to promote reusability. Basically, wildcards could be related to widening technique, but it is more dependent on the language feature than it is on any other factors. This guideline was the least satisfactory in the group of finalized guidelines. Perhaps this resulted

from the fact that it was so effortless to come up with, (the language provides it), and its use is fairly limited (in received operations only, e.g. receive and getcall).

Guidelines 9 and 10 illustrate how to modularize testware within and between TTCN-3 modules. Good modular structures facilitate reuse and maintenance of testware. The guidelines were inspired by the test suite structuring presented in section 2.6.

All in all, the guidelines and the presented examples are the result of several iterations. During the process, some candidate guidelines have been disqualified because they were too awkward to be applied in practice or too obvious to be called guidelines. Nevertheless, the ten remaining ones represent a selection of guidelines covering a broad basis of factors that have an impact on test reuse.

6.3 Guidelines

The guidelines are numbered, however, they are in no specific order of importance. Use of every guideline is illustrated with one or more illustrative examples, where the reserved words of the TTCN-3 language are printed in **bold-face**. These examples, however, are not complete but only fragments of real implementations as it would consume a considerable amount of space to represent all of them in full-scale. Nevertheless, they present everything that is essential to understanding the use and meaning of the respective guidelines. Applicability of guidelines to different test reuse viewpoints, presented in sections 5.1.1, 5.1.2 and 5.1.3, is contemplated briefly.

6.3.1 Guideline 1. Reusing Testers in a Distributed Test System

Separate PTCs for Upper and Lower Tester

In Figure 14 the functionality of the UT and LT is implemented in functions that run concurrently on separate PTCs. These PTCs are mapped to the test system interface and connected to the MTC that controls the execution of functions through the connected ports (i.e. controls the test behavior). This approach is useful when the test components are distributed in every testing level, or when

the functionalities of the UT and LT are loosely tied together or their implementations are large and complex. One example of this is when testers simulate large independent subsystems interworking with the SUT.

In integration testing, the selection of reused testers should naturally be based on the visibility of the interfaces of the integrated SUTs. For instance, if SUT 1 is on top of SUT 2 (e.g. SUT 1 is a higher level component, higher layer in layered architecture, etc.), then the testers to be reused are UT1 and LT2. This is further explained later on in this guideline.

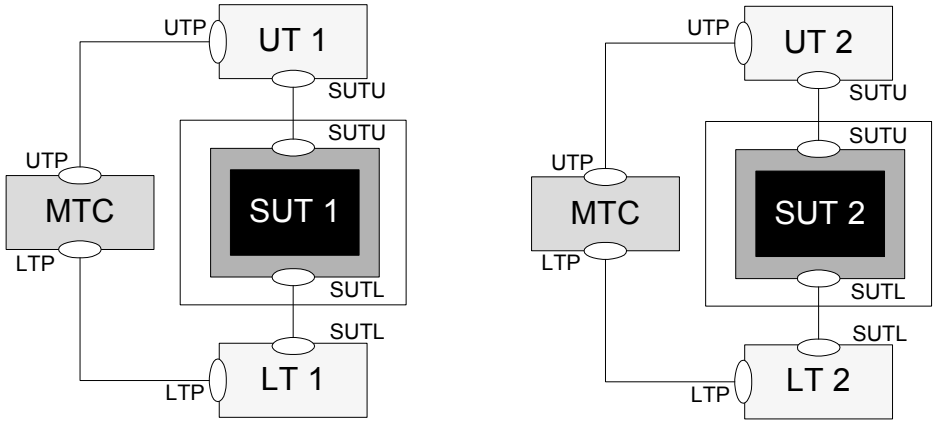


Figure 14. Test system configuration where UT and LT are running on separate mapped PTCs.

The code example in Figure 15 illustrates one way in which the functionality of the upper and lower tester could be implemented using functions that run on PTCs and are controlled by the MTC. After initiation (i.e. procedure call), the functions run concurrently on their respective PTC.

```
module MySUT1_tests {
  signature comp1();     // test step
  : // other test steps
  type component MainTC {
    port MyProcedurePort UTP;     // procedure port
    port MyProcedurePort LTP;     // procedure port
    var Upper_Tester UT1;         // component variable
    var Lower_Tester LT1;         // component variable
  }
  type component Upper_Tester {
    port MyProcedurePort UTP;     // procedure port
    port MyMessagePort SUTU;      // message port
  }
  : // other component definitions i.e Lower_Tester and TSI
  function UT1_Behavior() runs on Upper_Tester {
    alt {
      [] UTP.getcall(comp1:{}) {
        // Here is the UT functionality related to call comp1
        repeat; // re-evaluate alt statement
     }
      [] UTP.getcall(comp2:{}) { // another call, etc.
      :
    }
  }
  function LT1_Behavior() runs on Lower_Tester {
    alt {
      [] LTP.getcall(comp1:{}) {
        // Here is the LT functionality related to call comp1
        repeat; // re-evaluate alt statement
      }
      : // other calls, etc.
    }
  }
  :
  function ConfigureStart(MainTC a_mtc, TSI a_system) runs on MainTC {
    var Upper_Tester upper := Upper_Tester.create;
    var Lower_Tester lower := Lower_Tester.create;
    connect(upper:UTP, a_mtc:UTP);  // connect upper tester and MTC
    connect(lower:LTP, a_mtc:LTP);  // connect lower tester and MTC
    map(upper:SUTU, a_system:SUTU); // map upper tester to TSI
    map(lower:SUTL, a_system:SUTL); // map lower tester to TSI
    UT1 := upper;              // save the reference in component variable
    LT1 := lower;              // save the reference in component variable
    UT1.start(UT1_Behavior()); // start upper tester
    LT1.start(LT1_Behavior()); // start lower tester
  }
  testcase tc_001() runs on MainTC system TSI {
    ConfigureStart(mtc, system);
    UTP.call(comp1:{}, nowait);   // UT1 functionality related to comp1
    LTP.call(comp1:{}, nowait);   // LT1 functionality related to comp1
    all component.done;           // PTCs have ended their execution
    UnConfigure(mtc, system);
  }
  : // other test cases
}
```

Figure 15. UT and LT implemented as functions running on separate PTCs.

Common PTC for Upper and Lower Tester

In Figure 16, the functionality of the UT and LT is implemented in the same function that runs on a single PTC. The PTC is mapped to the test system interface and connected to the MTC that controls the execution of the function through the connected port (i.e. controls test execution). This approach is useful when the test system is distributed in component or in other testing levels or when the functionalities of the UT and LT are simple or closely tied together. One example of this is when simple inputs to the SUT create simple outputs (e.g. SUT is used for transforming values) and PTCs are not needed to simulate complex subsystems. In Figure 16, the observation of the upper and lower service boundary is realized using a single function running on PTC that observes the upper and lower service boundary of the SUT through the mapped ports.

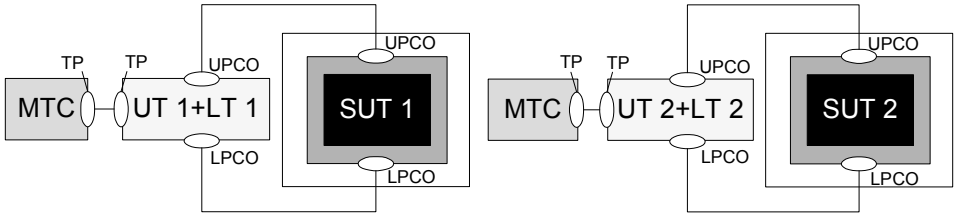


Figure 16. Test system configuration where UT and LT are running on the same mapped PTC.

Reusing the testers in integration testing requires that the unnecessary or excessive functionalities are blocked or removed. For instance, in component testing the test system (i.e. the tester) may have simulated a component that is replaced with real implementation in integration testing. Therefore, the simulated part that is no longer needed has to be blocked or removed. For example, if SUT 1 is on top of SUT 2 in the integrated system, then the functionality of LT 1 and UT 2 are redundant or undesired. Blocking or removing the functionality must be planned in component testing in order to make the transition onto other testing levels as easy as possible.

The code example in Figure 17 illustrates one way in which the functionality of the upper and lower tester could be implemented using a function that runs on a PTC and is controlled by the MTC. The if statements in *UT1_LT1_Behavior()* are used to initiate or block necessary or unnecessary actions on different testing levels.

```
module MySUT1_tests {
  :
  type component MainTC {
    port MyProcedurePort TP;        // procedure port
  }
  type component UT1_LT1_Tester {
    port MyProcedurePort TP;        // procedure port
    port MyMessagePort UPCO;        // message port
    port MyMessagePort LPCO;        // message port
  }
  type component TSI {
    port MyMessagePort UPCO;        // message port
    port MyMessagePort LPCO;        // message port
  }
  : // functions etc.
  function UT1_LT1_Behavior() runs on UT1_LT1_Tester {
    alt {
      [] TP.getcall(comp1:{blocking}) {
        if (blocking == false) {
          LPCO.send(data_req_msg); // only e.g. in component testing
        }
        // Here is the UT1_LT1 tester functionality related to comp1
        repeat;                     // re-evaluate alt statement
      }
      [] TP.getcall(comp2:{blocking}) {
        if (blocking == false) {
          :
      }
      : // another call, etc.
    }
  }
  function ConfigureStart(MainTC a_mtc, TSI a_system) runs on MainTC {
    var UT1_LT1_Tester upper_lower := UT1_LT1_Tester.create;
    connect(upper_lower:TP, a_mtc:TP);    // connect PTC and MTC
    map(upper_lower:UPCO, a_system:UPCO); // map upper PCO to TSI
    map(upper_lower:LPCO, a_system:LPCO); // map lower PCO to TSI
    UT1_LT1 := upper_lower;               // save the reference
    UT1_LT1.start(UT1_LT1_Behavior());    // start PTC
  }
  :
  testcase tc_001() runs on MainTC system TSI {
    ConfigureStart(mtc, system);
    TP.call(comp1:{false}, nowait);       // functionality related to comp1
    TP.call(comp2:{false}, nowait);       // functionality related to comp2
    :                                     // other calls, etc.
    UT1_LT1.done;                         // PTC has ended its execution
    UnConfigure(mtc, system);
  }
  :
}
```

Figure 17. UT and LT implemented as a function running on the same PTC.

Integrated System

The test system configuration presented in Figure 18 illustrates how the upper tester of SUT 1 and the lower tester of SUT 2 used in component testing (illustrated in Figure 14 and Figure 16) are reused in testing the integrated component.

As mentioned earlier this type of configuration is useful when the test system is distributed (i.e test components are distributed). Another advantage is that MTC can buffer instructions to the ports of the testers, so that test execution moves forward without unnecessary delays.

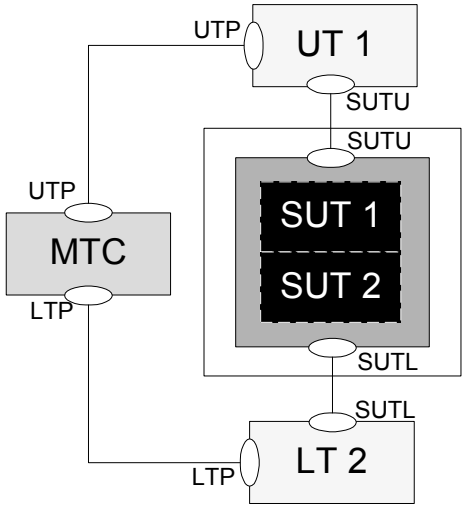


Figure 18. Test system configuration where UT 1 and LT 2 are reused in integration testing.

The code example in Figure 19 illustrates how the code presented in Figure 15 is reused. In this example, everything from modules *MySUT1_tests* and *MySUT2_tests* is imported and the test system is configured as illustrated in Figure 18. New test cases are created by reusing the functions *UT1_Behavior()* and *LT2_Behavior()*.

```
module MySUT3_tests {
  import from MySUT1_tests all; // importing everything
  import from MySUT2_tests all; // importing everything
  :
  function ConfigureStart(MainTC a_mtc, TSI a_system) runs on MainTC {
    : // create, connect and map testers and save references
    UT1.start(UT1_Behavior()); // start UT 1 functionality
    LT1.start(LT2_Behavior()); // start LT 2 functionality
  }
  testcase Int_tc_001() runs on MainTC system TSI{


    ConfigureStart(mtc, system);


    UTP.call(comp1:{}, nowait); // functionality related to comp1
    LTP.call(comp1:{}, nowait); // functionality related to comp1
    all component.done;         // PTCs have ended their execution
    UnConfigure(mtc, system);
  }
  testcase Int_tc_002() runs on MainTC system TSI{
    ConfigureStart(mtc, system);
    UTP.call(comp2:{}, nowait); // functionality related to comp2
    LTP.call(comp2:{}, nowait); // functionality related to comp2
    all component.done;         // PTCs have ended their execution
    UnConfigure(mtc, system);
  }
  :
}
```

Figure 19. Reusing the UT and LT behavior.

Reusing the code presented in Figure 17 is very similar. In contrast to the previous example, the undesired actions of the functions (*UT1_LT1_Behavior()* and *UT2_LT2_Behavior()*) have to be blocked by using the signature parameters. This guideline is especially useful when two components are integrated one on top of the other (e.g. layered architecture, protocols, etc.) as illustrated in Figure 18. This guideline is applicable to all viewpoints, but its utilization should especially be considered from the vertical viewpoint. Horizontal and historical reuse should be taken into account if the components under test are reusable and their interfaces are not likely to change between products of the same family or product generations.

6.3.2 Guideline 2. Reusing Testers in a Centralized Test System

Guideline 1 is most applicable when the test components are distributed or when the MTC needs to communicate with the PTC(s) during test case execution (i.e. synchronize and control test execution). However, when test system distribution or communication during test execution is not necessary, Guideline 1 has its downsides. Connecting the MTC and PTCs through ports requires extra effort and controlling the test execution through ports may be difficult to understand. In this guideline, PTCs are not connected to the MTC. Instead, after they are created and started, they execute their appointed actions independently.

Separate PTCs for Upper and Lower Tester

In Figure 20, the functionality of the UT and LT is implemented in functions that run simultaneously on separate PTCs that are mapped to the test system interface. This approach is useful when the test components are not distributed and when the functionalities of the UT and LT are loosely tied together or their implementations are large and complex. This approach is very much the same as the one first presented in Guideline 1, with the exception that the MTC does not control the test execution through communication ports.

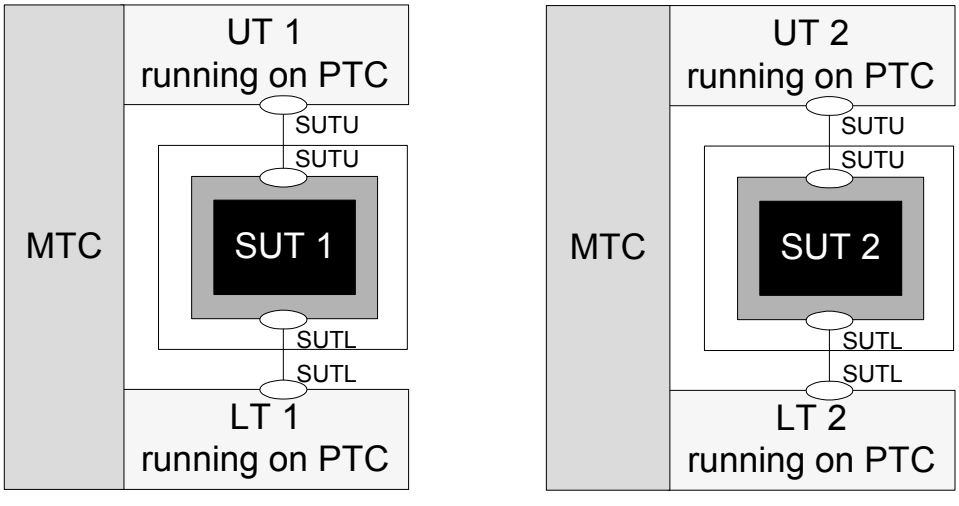


Figure 20. Test system configuration where UT and LT are running on separate PTCs.

The code example in Figure 21 illustrates how the functionality of the upper and lower tester could be implemented using functions that run on PTCs.

```
module MySUT1_tests {
  :
  template t_msg msg;
  :
  function data_req_ut(template t_msg a_msg) runs on Upper_Tester {
     // data_req_ut functionality related to Upper_Tester
  }
  function data_req_lt(template t_msg a_msg) runs on Lower_Tester {
     // data_req_lt functionality related to Lower_Tester
  }
  function data_ind_ut(template t_msg a_msg) runs on Upper_Tester {
     // data_ind_ut functionality related to Upper_Tester
  }
  function data_ind_lt(template t_msg a_msg) runs on Lower_Tester {
     // data_ind_lt functionality related to Lower_Tester
  }
  function data_rsp_ut(template t_msg a_msg) runs on Upper_Tester {
     // data_rsp_ut functionality related to Upper_Tester
  }
  :  // other functions
  testcase tc_001() runs on MainTC system TSI {
    :
    UT1.start(data_req_ut(msg)); // start first UT function
    LT1.start(data_req_lt(msg)); // start first LT function
    LT1.done;
    LT1.start(data_ind_lt(msg)); // start second LT function
    :
  }
  testcase tc_002() runs on MainTC system TSI {
    :
    LT1.start(data_ind_lt(msg)); // start first LT function
    UT1.start(data_ind_ut(msg)); // start first UT function
    UT1.done;
    UT1.start(data_rsp_ut(msg)); // start second UT function
    :
  }
  :
}
```

Figure 21. Binding several functions to PTCs.

In the code example in Figure 21, the specific function is bound to the specific component with the keyword **start (**in the test cases *tc_001()* and *tc_002()***)**. Several functions can be bound to the same component in the test case, but only after the one previously bound has ended its execution. This is checked using the keyword **done**.

Upper and Lower Tester on MTC

In Figure 22, the functionality of the UT and LT is implemented in the same function that runs on the MTC. The MTC is mapped to the test system interface. This approach is useful when the test system is not distributed on component or on later testing levels or when the functionalities of the UT and LT are simple or closely tied together. Observing the upper and lower service boundary of the SUT is realized using a single function running on MTC as illustrated in the Figure 22. Once again, using a single function requires that when the function is reused in integration testing, some of its actions may have to be blocked.

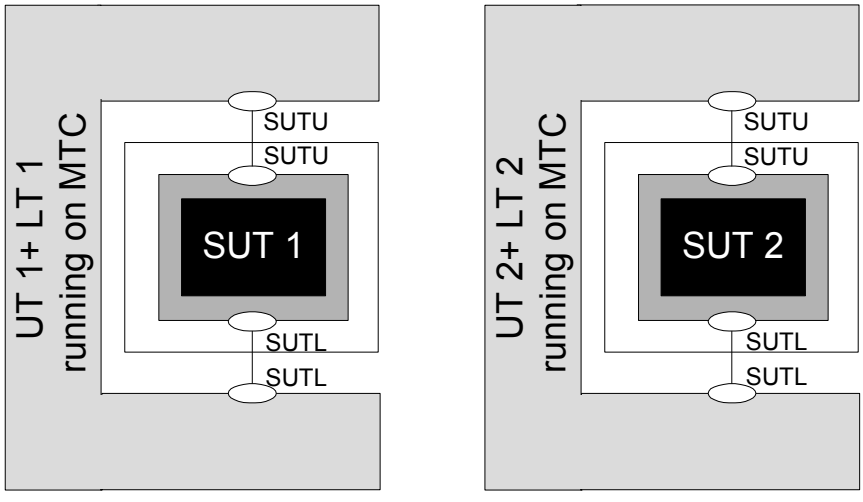


Figure 22. Test system configuration where UT and LT are running on the MTC.

The code example in Figure 23 illustrates how the functionality of the upper and lower tester could be implemented using functions that run on the MTC. In the test cases, the specific functions are called one after the other to test the functions of the SUT.

```
module MySUT1_tests {
  :
  function data_req(template t_msg a_msg) runs on MainTC {
     // data_req functionality related to UT1 and LT1
  }
  function data_ind(template t_msg a_msg) runs on MainTC {
     // data_ind functionality related to UT1 and LT1
  }
  function data_rsp(template t_msg a_msg) runs on MainTC {
     // data_rsp functionality related to UT1 and LT1
  }
  :
  testcase tc_001() runs on MainTC system TSI {
    :
    map(mtc:SUTU, system:SUTU); // map MTC to TSI
    map(mtc:SUTL, system:SUTL); // map MTC to TSI
    data_req(msg);              // first UT1 + LT1 function
    data_ind(msg);              // second UT1 + LT1 function
    :
  }
  testcase tc_002() runs on MainTC system TSI {
    :
    map(mtc:SUTU, system:SUTU); // map MTC to TSI
    map(mtc:SUTL, system:SUTL); // map MTC to TSI
    data_ind(msg);              // second UT1 + LT1 function
    data_rsp(msg);              // third UT1 + LT1 function
    :
  }
  :
}
```

Figure 23. Running several functions on the MTC.

Integrated System

The test system configuration presented in Figure 24 illustrates how the reusable parts used in component testing (illustrated in Figure 20 and Figure 22) could be reused in testing the integrated system. As mentioned earlier, this type of configuration is useful when the test system is not distributed.

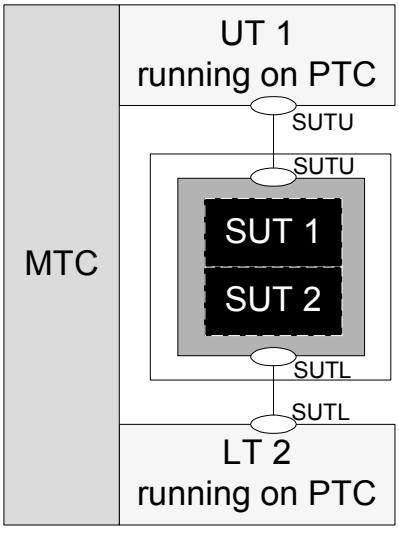


Figure 24. Test system configuration where UT 1 and LT 2 are reused in integration testing.

The code example in Figure 25 illustrates how the code presented in Figure 21 is reused. In this example everything from modules *MySUT1_tests* and *MySUT2_tests* is imported and the test system is configured as illustrated in Figure 24. A new test case is created by reusing the functions and binding them to specific PTCs.

```
module MySUT3_tests {
  import from MySUT1_tests all; // importing everything
  import from MySUT2_tests all; // importing everything
  :
  function Configure(TSI tsi_system) runs on MainTC {
    : // create and map testers and save references
  }
  :
  testcase tc_001() runs on MainTC system TSI {
    CreateAndConfigure(system);
    UT1.start(data_req_ut(msg)); // start first UT1 function
    LT2.start(data_req_lt(msg)); // start first LT2 function
    LT2.done;
    LT2.start(data_ind_lt(msg)); // start second LT2 function
    :
  }
  :
}
```

Figure 25. Reusing the UT1 and LT1 behavior.

Reusing the code presented in Figure 23 is very similar. However, blocking of undesired actions, as previously discussed, has to be handled. Compared to the approaches presented in Guideline 1, this guideline does not support distribution of test components. However, the amount of communication ports decreases, thus decreasing the complexity of the test system and execution, which in turn increases understandability. Nevertheless, this guideline, just as Guideline 1, is also applicable to all viewpoints, and its utilization should especially be considered when testing layered architectures or protocols. Horizontal and historical reuse should be taken into account if the components under test are reusable and their interfaces are not likely to change between products of the same family or product generations.

6.3.3 Guideline 3. Use Preambles, Bodies and Postambles

In component testing, a group of test cases may drive the IUT to the same specific test state in order to test it by using different test bodies. In integration testing, the successful execution of the test body demands that the integrated components are in specific states. These observations obviously favor the use of generic shared preamble that could be used in all the test cases of component and integration testing related to a specific state.

Using the same test body in different testing states is an obvious approach to reusing tests, since it is the body that verifies the test purpose and the verdict of the test. Regardless of the testing level, the IUT should conform to the requirements specification, which in black box testing is the basis of the test cases and their respective purposes. The use of a shared test body is also reasonable when the IUT is driven to different test states (even on the same testing level) and its behavior is observed when executing the same body. An example of this is use of the same input sequences in different states. However, in some cases, e.g. when testing safety critical or very expensive systems, it is preferable to verify that the state of the IUT is “correct” (i.e. valid or invalid) before execution of the test body in order to avoid any costly mistakes.

After the test body is executed a postamble drives the IUT to the desired stable state. This state could be the one where the execution of preamble began or some other that is more convenient for the next test case. Nevertheless, postamble, like

preamble, is usually a very static set of test steps, so its reuse potential is high between testing levels.In addition to the mentioned pros of reusing preambles, bodies and postambles, testers have the freedom to combine them in a variety of ways, thus making it possible to create various state combinations or achieve better coverage of testing. The code example in Figure 26 illustrates the reuse of a preamble and a postamble.

```
module Component_tests_CUT1 {
  :
  testcase tc_state2_001() runs on Connector system CTS {
    map(mtc:PCO1, system:PCO1);
    CUT1_to_state2();      // preamble
    SendMSG();             // body
    TearDown();            // postamble
  }
  testcase tc_state2_002() runs on Connector system CTS {
    map(mtc:PCO1, system:PCO1);
    CUT1_to_state2();      // same preamble
    SendData();            // body
    TearDown();            // same postamble
  }
  : // other test cases etc.
}
module IntegrationTests {
  import from Component_tests_CUT1 { function CUT1_to_state2 }
  import from Component_tests_CUT1 { function TearDown }
  :
  testcase interOp_001() runs on Connector system CTS {
    :
    CUT1_to_state2();      // preamble of CUT1 reused
    :                      // possible preamble of CUT2
    SendSignal();          // body
    :
    TearDown();            // postamble reused
    :                      // possible postamble of CUT2
  }
  :
}
```

Figure 26. Reusing preamble and postamble.

In the code example in Figure 26, the preamble, body and postamble are implemented as functions. The same preamble and postamble are used in test cases *tc_state2_001()* and *tc_state2_002()*. However, this is not reuse, but more like following good coding conventions. Instead, using the same preamble and postamble in integration test case *interOp_001()* is reuse, because it takes place between testing levels, not between test cases on the same level.

This guideline is especially applicable to the vertical viewpoint. Dividing the test case into preamble, body and postamble may not be the best choice for a test case structure if the horizontal or historical viewpoint is the main focus of reuse.

6.3.4 Guideline 4. Implement Test Cases Using High Level Functions

All the previous guidelines have utilized functions in some form. This guideline encourages use of functions in the manner that Smith [50] has found to be useful. In Smith's study, tests are assembled using high-level operations that can be combined in a variety of ways to implement tests quickly and efficiently [50]. However, operations are not always static between products of the same family or between product generations. In order to cope with this issue, we need a certain level of generality and variability.

In order to improve the horizontal or historical reuse of test cases, test cases should be compiled by using functions with descriptive and generic names. The goal is to make test cases reusable by hiding the possible variations in implementations and interfaces inside different function implementations. For instance, a web mail service provider may offer its services for use with a personal computer or with a mobile phone. In this case, there are common and specific parts for different terminals. The specific parts are the actual variation points of the test case.

Table 3 gives a short example of a use case in a fictitious web mail service. This use case creates a base for the high-level test case and different function implementations presented in the code example in Figure 27.

Table 3. Use case description.

Use Case Description – Log in and read mail using secure connection	
Introduction	User logs in to read his/her e-mail
Flow of events	1. User connects to server 2. Verify the username 3. Verify the password 4. Fetch the users mail 5. …

```
module CommonModule_Webmail_TestCases {
  :
  function VerifyUserName() runs on MyTester {
    : // VerifyUserName is the same for both implementations
  }
  function VerifyPassword() runs on MyTester {
    : // VerifyPassword is the same for both implementations
  }
  testcase login_and_read_mail() runs on MyTester {
    :
    CreateConnection(); // Variation point
    VerifyUserName();   // Common part
    VerifyPassword();   // Common part
    FetchMail();        // Variation point
    :
  }
  :
}
module SpecificModule_PC {
  :
  function CreateConnection() runs on MyTester {
    :  // CreateConnection for pc connection (modem, ISDN, ADSL, etc.)
  }
  function FetchMail() runs on MyTester {
    :  // FetchMail for pc connection (e.g. fetch all at once)
  }
  :
}
module SpecificModule_MobilePhone {
  :
  function CreateConnection() runs on MyTester {
    :  // CreateConnection for mobile phone connection (GRPS, HSCSD)
  }
  function FetchMail() runs on MyTester {
    :  // FetchMail for mobile phone connection (e.g. fetch and show
    :  // the first mail and the rest as a background process)
  }
  :
}
module TCs_For_PC {
  import from CommonModule_Webmail_TCs all; // import common part
  import from SpecificModule_PC all;        // selecting pc connection
  :
control {
  execute(login_and_read_mail());           // executing the test case
  :
}
}
```

Figure 27. Test case implemented using functions with descriptive names.

The code example in Figure 27 presents a test case (*login_and_read_mail()*) that is constructed by using functions which correspond to those points presented in the use case description of Table 3. The first and the last functions depend on the selected connection, whereas the two in the middle are common to both connections. The generic naming of the functions and test components (*MyTester*) makes it possible to execute test cases without any changes, after the necessary function implementations have been imported.

This guideline is applicable to all viewpoints, but especially to the horizontal and historical viewpoint where implementations (i.e. interfaces) are likely to change. Perhaps the most value can be obtained when testing products of the same product family, since there is a large set of specified features or functionalities that can be either common or product-specific.

6.3.5 Guideline 5. Parameterize Test Behavior

So far all the guidelines have dealt very little with variations. One of the most powerful variation mechanisms in creating small variations is parameterization. In fact, sometimes it is possible to cover a large set of test purposes by using only small variations, e.g. using boundary and near boundary values as parameter values.

TTCN-3 offers dynamic value parameterization of *function, signature, altstep, template and testcase* [4]. Using parameterization appropriately makes it possible to create more adaptable functions, signatures etc., thus enhancing reusability. The code example in Figure 28 illustrates how test behavior (i.e functions *CreateConnection()* and *SendData()*) is parameterized with test data that is imported from a specific module. Test behavior can also be parameterized using constants, variables or templates that are either global or tied to a specific test component.

Parameterization itself does not necessarily mean reuse. One has to apply parameterization in a way that promotes reuse. For instance, in vertical reuse, component tests could test the component extensively using boundary and near boundary values as parameters, whereas in integration testing, parameters could be altered to suit the needs of interoperability tests. In horizontal and historical reuse, different products or product releases may support different data values or data

ranges that could be handled using parameterization. It should also be noted that it is preferable, in most cases, to separate test data into its own module, as illustrated in the code example in Figure 28, and avoid hard-coded values when appropriate.

```
module Component_Testing_Data { // Test data for component testing
  var integer state1 := 10;
  var integer state2 := 11;
  :
  var bitstring data1 := '01101'B;
  var bitstring data2 := '01110'B;
  :
}
module Integration_Testing_Data { // Test data for integration testing
  var integer state1 := 1;
  :
}
module CUT1_Testcases {
  import from Component_Testing_Data all;
  :
  function CreateConnection(integer parameter) runs on Connector {
    // function behavior according to parameter value
  }
  function SendData(bitstring parameter) runs on Connector {
    // function behavior according to parameter value
  }
  :
  testcase tc_001(integer a_state, bitstring a_data) runs on Connector
  system CTS {
    :
    CreateConnection(a_state); // preamble is parameterized
    SendData(a_data);          // body is parameterized
    SetPassAndUnmap();         // postamble is not parameterized
  }
  :
control { // In the control part, the real values for parameters are given
  execute(tc_001(state1, data1));
  execute(tc_001(state2, data1));
  execute(tc_001(state1, data2));
  execute(tc_001(state2, data2));
  :
}
}
```

Figure 28. Test behavior parameterized.

This guideline is applicable to all viewpoints. Parameterization offers such a powerful and simple mechanism for reuse that it should be practiced regardless of viewpoint. Separating test data and avoiding hard-coded values should also be applied in all viewpoints.

6.3.6 Guideline 6. Use Selection Structures to Alternate Test Behavior and Execution

This guideline has a strong relation to the previous one. To increase the reusability of test cases, if-else structures should be used to offer optional test behavior as illustrated in the code example in Figure 29. In the example, the parameter *a_type* is used to select between two alternate behaviors. The purpose of the test case (*tc_001()*) is to drive the implementation into the same specific state and to perform different types of testing depending on the value *a_type*.

```
module CUT1_Tests {
  // enumerated type for testing types
  type enumerated t_type {FUNCTIONAL, LOAD, ...};
  :
  testcase tc_001(template testing_type a_type, integer NumberOfPTCs)
  runs on MainTC system CTS {
    : // same preamble for all testing types
    if (a_type == FUNCTIONAL) { // functional testing
      var CUT_Tester tester := CUT_Tester.create;
      map(tester:SUTU, system:SUTU);       // map tester to TSI
      map(tester:SUTL, system:SUTL);       // map tester to TSI
      connect(mtc:TP, tester:TP);          // connect tester to MTC
      tester.start(Test_Behavior());       // bind test behavior to PTC
      TP.call(inst_01:{}, nowait);         // MTC gives instructions
      :                                    // and controls PTC
      tester.done;
      :
    }
    else if (a_type == LOAD) { // load testing
      var integer i;
      var CUT_Tester c_tester[NumberOfPTCs]; // creating tester array
      for (i := 0; i<NumberOfPTCs; i := i + 1) {
        tester[i] := Upper_Tester.create;  // create instance
        map(tester[i]:SUTU, system:SUTU);  // map instance to TSI
        map(tester[i]:SUTL, system:SUTL);  // map instance to TSI
        connect(mtc:TP, tester[i]:TP);     // connect instance to MTC
        tester[i].start(Test_Behavior());  // bind test behavior to
        :                                  // instances
      }
      for (i := 0; i<NumberOfPTCs; i := i + 1) {
        TP.call(inst_01:{}, nowait) to tester[i]; // instructions
        :                                         // to instances
      }
      all component.done; // all instances have ended
    }
    : // possibly other testing types
  }
  :
}
```

Figure 29. Creating alternate test behavior by using if-else structures.

The previous example illustrated how the definitions part of a TTCN-3 module, a test case in particular, could be structured to offer alternate test behavior. However, the control part of a module can also be structured using if-else statements to create alternate test execution structures. The code example in Figure 30 illustrates a selection structure implemented using a module parameter and if-else structures. In this example, the module parameter *feature_group* is used to select the group of test cases to be executed.

Combining parameterization and selection structures as illustrated in the examples provides multiple reuse possibilities. The drawback is that these structures increase the size and complexity of the code thus making it harder to maintain. Nevertheless, using selection structures appropriately is recommendable in all the viewpoints.

```
module CUT1_Features {
  // module parameter with a default value
  modulepar {integer feature_group}
  :
control {
  : // using module parameters to alternate control structures
  if (feature_group == 1) {
    execute(cut1_tc_group1_001());
    execute(cut1_tc_group1_002());
    :
  }
  else if (feature_group == 2) {
    execute(tc_group2_001());
    execute(tc_group2_002());
    :
  }
  : // other groups
}
}
```

Figure 30. Creating alternate test execution by using if-else structures.

6.3.7 Guideline 7. Use Common Types and Template Modification

In some cases it is possible and preferable to use shared TTCN-3 type or data definitions for all communication (i.e. the same structured types for different templates or template modification). An example of this is when all the messages used in the system are identical or similar in structure, as illustrated in

Figure 31 [52]. Small variations can be carried out using e.g. template parameterization and/or modification and optional fields. Modification should be favored over parameterization, because it provides a later design decision which is commonly considered to be a major enabler for reuse. In parame-terization the parameter is bound to a specific field whereas any field can be replaced when using modification.

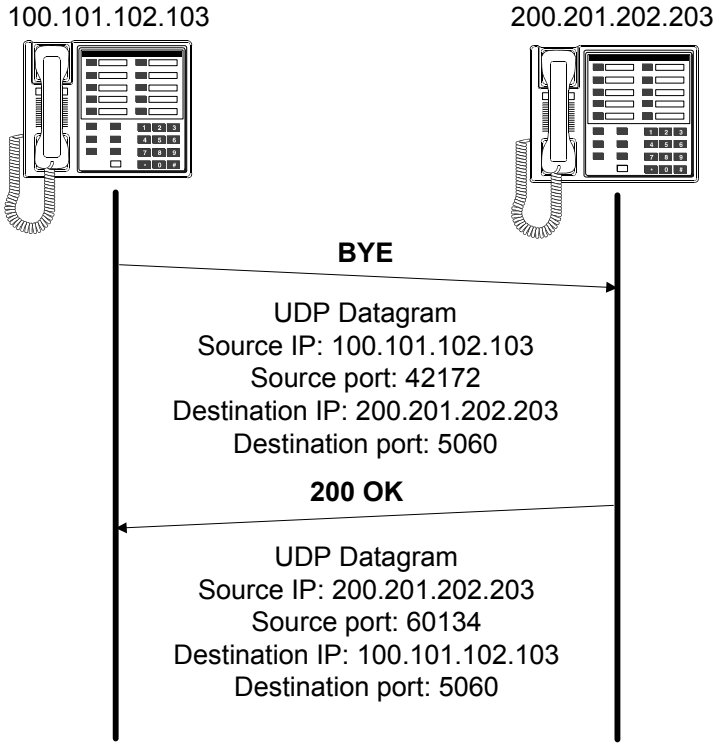


Figure 31. Ending an SIP session.

Common Type Definitions

Figure 31 presents a simple example of ending a SIP session [52 p. 31]. A quick assessment of the figure reveals that the message structures of the BYE and 200 OK are identical. Hence, the same structured type could be used when creating templates for these two messages as illustrated in the code example in Figure 32.

The code example in Figure 32 presents a truncated type definition for a SIP message (*SIPMessageType*) that can be used in different template definitions.

Using a common type definition for all the message templates (i.e. different SIP messages) is not reuse, but can be compared to a function call. However, the type definition and the templates can be reused when testing different SIP applications, e.g. user agents or servers. In addition, this approach decreases the maintenance effort, as is typical of test reuse.

```
module SIP_Messages {
  :
  type record SIPMessageType {
    : // headers via, call-id, cseq, content-length, optional headers etc.
    charstring source,
    charstring sourceport,
    charstring destination,
    charstring destination_port
  }
  template SIPMessageType BYEMessage := {
    :
    source := "100.101.102.103",
    sourceport := "42172",
    destination := "200.201.202.203",
    destination_port := "5060"
  }
  template SIPMessageType OK_200_Message := {
    :
    source := "200.201.202.203",
    sourceport := "60134",
    destination := "100.101.102.103",
    destination_port:= "5060"
  }
  :
}
module SIP_UserAgent_TCs {
  import from SIP_Messages all; // reuse SIP message types & templates
  :
}
module SIP_Server_TCs {
  import from SIP_Messages all; // reuse SIP message types & templates
  :
}
```

Figure 32. Reusing a structured type and templates.

In some cases, combining structured types may present a reuse possibility as, for example, in a protocol stack where messages are "extended" on each layer by adding simple or complex headers. The code example in Figure 33 illustrates the matter. In the example, the type definition of the top layer message (*TOPLayerMessage*) is used as a part of the middle layer message (*MIDDLELayerMessage*), which in turn is used as a part of the bottom layer

message (*BOTTOMLayerMessage)*. Hence, if there are any changes in the message structures in the top or middle layer, they will only affect the code in the top or middle layer modules. Thus, the changes can be made in a concentrated manner, which decreases the maintenance effort.

```
module TOPLayer {
  type record TOPLayerMessage { // top layer message definition
    charstring field1,
    integer field2
  }
  :
}
module MIDDLELayer {
  import from TOPLayer all;
  type record MIDDLELayerMessage { // middle layer message definition
    TOPLayerMessage field1,        // reuse top layer msg
    hexstring field2,
    bitstring field3
  }
  :
}
module BOTTOMLayer {
  import from TOPLayer all;
  import from MIDDLELayer all;
  type record BOTTOMLayerMessage {  // bottom layer message definition
    MIDDLELayerMessage field1,      // reuse top and middle layer msg
    octetstring field2,
    float field3
  }
  template BOTTOMLayer BOTTOMLayer_Message := {
    field1 := {{"TOPLayer header", 1}, 'AC01B'H, '0011'B},
    field2 := 'AE24'O,
    field3 := 1452.4
  }
  :
}
```

Figure 33. Combining structured types.

Template Modification

The code example in Figure 34 illustrates template modification that is used to provide a simple form of inheritance. In the example, any template field and its corresponding value or matching symbol that is defined in the modified template replaces the one specified in the parent template. Template modification should be used, especially if templates are large but very similar.

```
module IntegrationTests {
  template SIPMessageType BYEMessage := {
    :
    protocol:= "UDP Datagram",
    source:= "100.101.102.103",
    sourceport:= "42172",
    destination:= "200.201.202.203",
    destination_port:= "5060"
  }
  // fields protocol and destination_port are unchanged
  template SIPMessageType OK_200_Message modifies BYEMessage := {
    :
    source:= "200.201.202.203",      // modified
    sourceport:= "60134",            // modified
    destination:= "100.101.102.103"  // modified
  }
}
```

Figure 34. Template modification.

Reusing and combining structured types and using template modification is applicable to all viewpoints. Perhaps the greatest savings are found in areas of communication and protocol testing where messages are similar or at least the basic structure is the same.

6.3.8 Guideline 8. Use Wildcards

The use of wildcards (? and *) prevents overspecialization of any parameterizable language element, making them more generic and thus more reusable. Wildcards can be used *instead* of values or *inside* values, in received operations only (i.e. **receive**, **getcall**, **getreply** and **catch** operations) [4].

Early in test development, it is not always necessary to know or predict which concrete values are correct for received messages, e.g. the functionality of the IUT is not exactly known, so its produced outputs are vague. Sometimes it is only necessary to confirm that the IUT is stabile and a message or a call comes across. For example, in integration testing it is necessary to guarantee basic interoperation (e.g. a message or call comes across) whereas system testing takes interoperation to more specific level (e.g. a message or call comes across with the right values). The code example in Figure 35 illustrates the use of wildcards. In this example, the template *CCMessageSend* has a specific values set, whereas

the template *CCMessageReceive* uses wildcards, making it possible to use the same template for multiple receive operations.

```
module IntegrationTests {
  :
  type record CCMessageType {
    charstring field1,
    boolean field2 optional,
    integer(3) field3
  }
  template CCMessageType CCMessageSend(boolean value2) := {
    field1 := "Begin",
    field2 := value2,        // specific value from parameter
    field3 := {128, 4, 8}
  }
  template CCMessageType CCMessageReceive := {
    field1 := "*",           // any charstring
    field2 := *,             // true or false
    field3 := {128, ?, 8}    // any integer as second value
  }
  :
    PCO1.send(CCMessageSend(true)); // every value is determined
    PCO1.receive(CCMessageReceive); // some values are be undetermined
    :
    PCO1.send(CCMessageSend(false));
    PCO1.receive(CCMessageReceive); // same template for receive operations
    :
}
module SystemTests {
  import from IntegrationTests { type CCMessageType }
  :
  template CCMessageType CCMessageReceive := {
    field1 := "Begin",
    field2 := true,
    field3 := {128, 4, 8}
  }
  :
    PCO1.receive(CCMessageBegin:{"Begin", true, MyIntegers(3)});
    PCO1.receive(CCMessageReceive);
    :
}
```

Figure 35. Use of wildcards and reuse of structured type.

This guideline is applicable to all viewpoints and perhaps (as with Guideline 7) the best profits could be found in areas of communication and protocol testing.

6.3.9 Guideline 9. Modularize Tests According to Components

Test suite structuring means modularization of all the elements found in the test suite. Good modular structures support reusability and maintainability of tests [49]. Therefore, as far as vertical reuse is concerned, it should be considered whether dividing test cases into TTCN-3 modules according to the component that they test should be applied. However, depending on the size, variability and complexity of the CUT, the size of the module may grow to an undesiredable level. In this case, additional criteria for test case division should be considered. Preambles, bodies and postambles should be placed in their own module according to CUT as well. Thus, they are separated from any specific testing level and can be imported and reused in another testing level or type of testing. Furthermore, it can be stated that everything in the definitions part of the TTCN-3 module that can be used in testing another component or that can be reused on another testing level should be placed in specific modules or at least grouped within one module.

Within the TTCN-3 module, grouping the definitions enables their reuse for similar tests. Test cases can also be grouped based on the testing type or level where they are thought to be of the most use and then imported later on. However, importing definition groups separately may be considered laborious and the real selection of test cases happens in the control part, which cannot be imported. Therefore, we can conclude that in most cases, grouping enhances readability more than reusability. The code example in Figure 36 illustrates how the definitions part could be divided into various groups.

```
module CUT1_Testcases {
  group Configurations {/*...*/} // components, ports
  group Declarations {/*...*/}   // timers, signatures, templates,…
  group Behavior {
    testcase CUT_DRC_001() /*...*/
    testcase CUT_DRC_002() /*...*/
    :
    group LoadTests { // LoadTesting
      testcase CUT1_DRC_INT_001() /*...*/
      testcase CUT1_DRC_INT_002() /*...*/
    }  // LoadTests
  } // Behavior
}
```

Figure 36. Dividing the definitions part into different modules and grouping of test cases.

When proceeding to integration testing, we can either import the grouped definitions of one module or import everything from the necessary modules. The latter option is an easier and less error-prone approach. The code example in Figure 37 illustrates the use of import clause and the "selection" of test cases in the control part.

```
module IntegrationTests {
  // importing only the specific groups of CUT1_Testcases
  import from CUT1_Testcases { group Configurations }
  import from CUT1_Testcases { group Declarations }
  import from CUT1_Testcases { group Behavior }
  : // importing everything from specific modules
  import from CUT2_Testcases all;
  import from CUT2_Configurations all;
  :  // The "selection" of test cases is in the control part
control {
  execute(CUT1_DRC_INT_001());
  execute(CUT2_DRC_INT_001());
  :
}
}
```

Figure 37. Importing (reusing) specific parts or all parts.

As is clearly observable from the presented examples and the previous paragraphs, this guideline is best suited for the vertical viewpoint.

6.3.10 Guideline 10. Modularize Tests According to Features

This guideline can be seen as an alternative to the previous guideline. As far as horizontal or historical reuse is concerned, it should be considered whether placing all the test cases in a module according to the feature they test should be applied. However, this can possibly divide the test cases of one CUT into more than one module and may also explode the amount of modules. This guideline may be most applicable when feature-based software development and testing is applied as studied by Korhonen et al.

As with the previous guideline, everything in the definitions part that can be used in testing another feature or that can be reused in testing another product or product release should be placed in specific modules or at least grouped within one module. Collecting the test cases according to the feature they test, does not

provide a clear image of what they actually test. Meaning that it may be difficult to perceive the IUT for a certain test case just by looking at that test case. Therefore, grouping could be applied if the feature embodies test cases for various components. This is illustrated in the code example in Figure 38.

```
module FeatureONEInProductA {       // feature
  :
  group CUT1_TCs{       // test cases for CUT1
    testcase CUT1connect01...
    testcase CUT1connect02...
    :
  }
  group CUT2_TCs{       // test cases for CUT2
    testcase CUT2connect01...
    testcase CUT2connect02...
    :
  }
:
}
module CControlForProductB {
  import from FeatureONEInProductA { group CUT1_TCs }
  import from FeatureONEInProductA { group CUT2_TCs }
  import from FeatureFOURInProductB all;
  :
control {
  execute(CUT1connect01...);
  :
  execute(CUT2connect01...);
  :
}
}
```

Figure 38. Reusing some of the tests found in products A and B.

As noted in the previous paragraphs and is apparent from the presented example, this guideline is best applied to horizontal and historical viewpoints and grouping merely enhances the possibility to apply it to vertical reuse as well.

7. Case Study: Vertical Reuse in Protocol Testing

This chapter presents a case study where the guidelines presented in chapter 6 are used in implementing a selection of reusable TTCN-3 tests for two protocols. The selection of tests is made on the basis of reuse potential, which is discussed later on. The purpose of the tests created *for* reuse is to validate conformity of the two protocols against respective protocol standards. Test reusability is studied when integration tests are created *with* reuse, based on the reusable material.

In this study, the tests are not executed on a real test system that would provide information about the maturity of the protocol implementations and validity of the tests. Clearly, test execution would provide valuable information, however, it has little significance when evaluating the applicability of the presented guidelines and the level of reuse. This case study was performed in cooperation with the VTT Technical Research Centre of Finland and NetHawk.

7.1 Introduction

The two protocols chosen for this case study are the Service Specific Connection Oriented Protocol (SSCOP) [53] and the Service Specific Coordination Function for support of signalling at the User Network Interface (SSCF-UNI), depicted in Figure 39 [54]. SSCOP and SSCF are successive layers in the signalling ATM Adaptation Layer (SAAL) that is used for reliable signalling between ATM endpoints. In fact, SAAL is a combination of two sublayers: a common part and a service specific part. The SSCOP and SSCF-UNI are located in the service specific part also known as the Service Specific Convergence Sublayer (SSCS). [53]

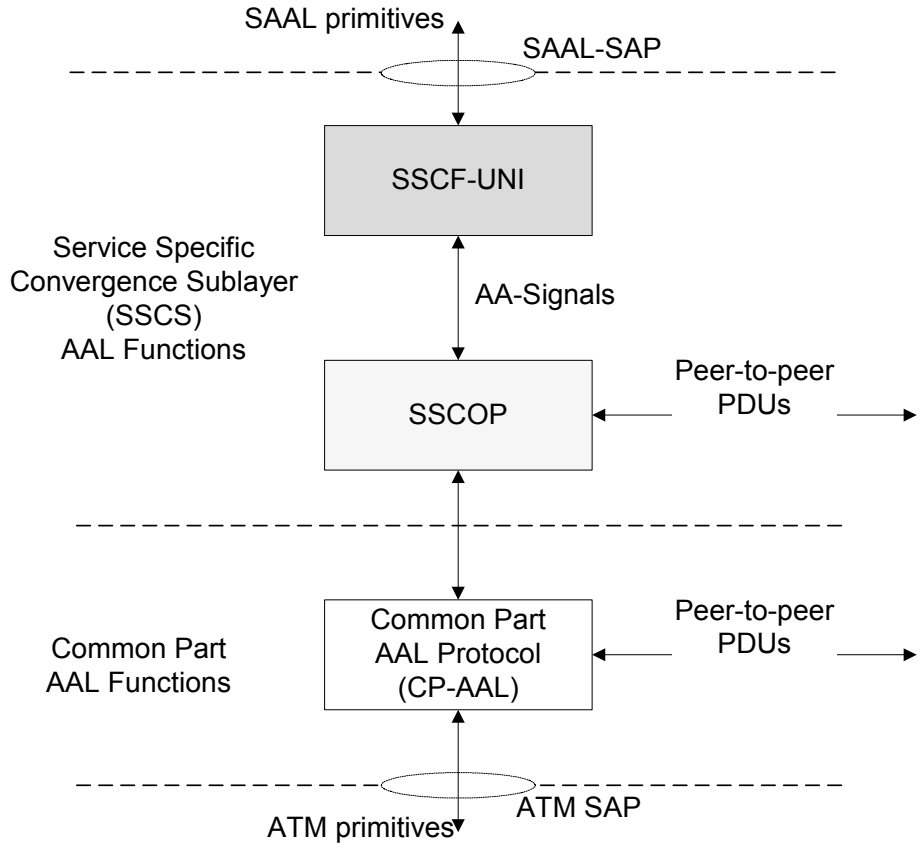


Figure 39. AAL protocol stack.

SAAL at the UNI provides the following services [54]:

- Unacknowledged transfer of data on point-to-point and point-to-multipoint ATM connections
- Assured transfer of data on point-to-point ATM connections
- Transparency of transferred data (i.e. no restrictions to content, format or coding)
- Means to establish and release SAAL connections for assured data transfer.

The Service Specific Connection Oriented Protocol (SSCOP) is a peer-to-peer protocol providing generic reliable data transfer service for different AAL interfaces. The protocol has ten possible states, and it communicates with SSCF-

UNI, system management layer and a peer entity using AA signals and PDUs. SSCOP performs the following functions [53 p. 4]:

- Sequence integrity of the transmitted user data
- Assured and unassured transfer of user data
- Error detection and recovery from errors in the operation of the protocol
- Reporting errors to layer management
- Error correction using selective retransmission
- Flow control (i.e. rate of transmission)
- Connection control (i.e. connection establishment, release and resynchronization)
- Connection maintenance in case of a prolonged absence of data transfer
- Local data retrieval by the SSCOP user
- Status information exchange between peer entities.

The Service Specific Coordination Function for support of signalling at the User Network Interface (SSCF-UNI) maps the services provided by the SSCOP to the needs of the SAAL user. It communicates with SSCOP, the system management layer and the SCCOP user using AA signals and AAL primitives. [54]

7.2 Planning and Preparation

Due to the time limitations and protocol relations, it was decided that the vertical reuse approach (i.e. reuse between testing levels) would be most suitable for the case study. The two sides of reuse were identified. Development *for* reuse consists of designing and implementing reusable tests for SSCOP and SSCF-UNI. Development *with* reuse means reusing the tests in designing and implementing integration tests.

Development *for* reuse began by getting familiar with the recommendations of the two protocols provided by ITU-T. Based on the recommendations, test purposes for the two protocols were identified and documented. Development

with reuse began by examining the reuse potential of the documented test purposes, so that only those test purposes with real reuse potential would end up being implemented. The reuse potential of component tests was identified based on the compound state transition table (known from this point onward as the transition table) presented in the recommendation of the SSCF-UNI. The transition table defines the possible state combinations of SSCF-UNI and SSCOP [54 p. 11].

It was decided that only the possible state combinations and use of valid messages as events would provide sufficient scope to study reuse of tests. However, one "impossible" state combination was added later on, merely out of curiosity, as it had virtually no effect on the results of this study. Figure 40 illustrates the process of creating the test purposes for component (i.e. *for* reuse) and integration (i.e. *with* reuse) testing. The numbers inside the brackets indicate the number of documented test purposes.

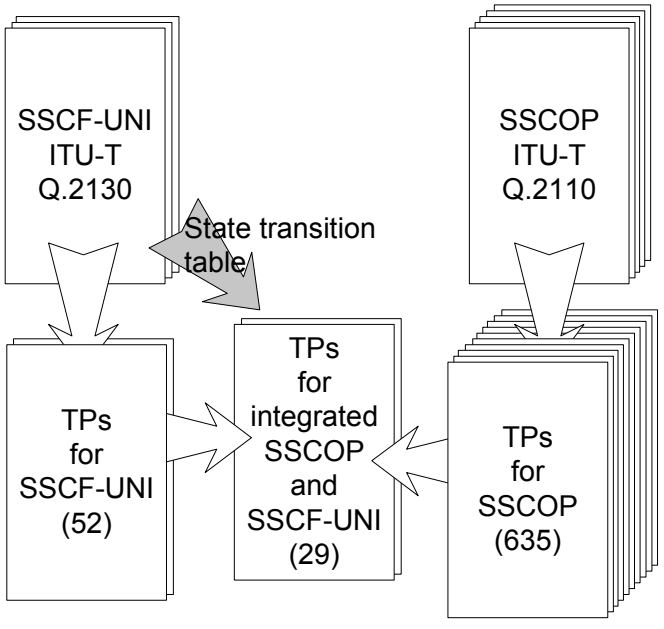


Figure 40. Creation of test purposes.

Limiting the scope to only possible states and valid messages meant that less than half of the test purposes for SSCF-UNI and only a fraction of the test purposes for SSCOP were to be implemented as test cases. However, the effort

used in creating the test purposes that were not implemented was not wasted, for it was used in evaluating the reuse potential of tests outside the scope of this thesis.

7.3 Design and Specification

After the test purposes were documented and the reuse potential was identified, the selected test purposes were to be refined into reusable component tests. As mentioned in the preceding section the transition table was used in identifying the necessary test purposes for integration testing. In addition, it was also used in selecting the component test purposes, those with real reuse potential in the given scope, to be implemented.

Table 4 presents a fraction of the transition table. The compound states are ordered pairs P/Q where P is the state of SSCF-UNI and Q is the state of SSCOP. Some of the events identified as illegal and associated with a compound state could be the result of collisions at the boundary between SSCOP and SSCF-UNI. [54 p. 11] Illegal events were not in the scope of this study.

Table 4. Fraction of the compound state transition table.

Compound state / Event	1/1	2/2	4/10	3/4	2/5
AAL-ESTABLISH request {Parameter Data} (Note 6)	AA-ESTABLISH request {BR := Yes SSCOP-UU := Parameter Data} (Note 6) State 2/2	Illegal	AA-RESYNC request {SSCOP-UU := Parameter Data} (Note 6) State 2/5	AA-ESTABLISH request {BR := Yes, SSCOP-UU := Parameter Data} (Note 6) State 2/2	Illegal

Figure 41 illustrates the meaning of the events listed in the transition table. If the event is AAL primitive it means that the SSCF-UNI receives it from the upper layer, maps it into an AA signal and sends it to the SSCOP that then sends an appropriate PDU. If the event is an AA signal it means that the SSCOP has

received a PDU from its peer and sent an appropriate AA signal to SSCF-UNI, which maps it into AAL primitive.

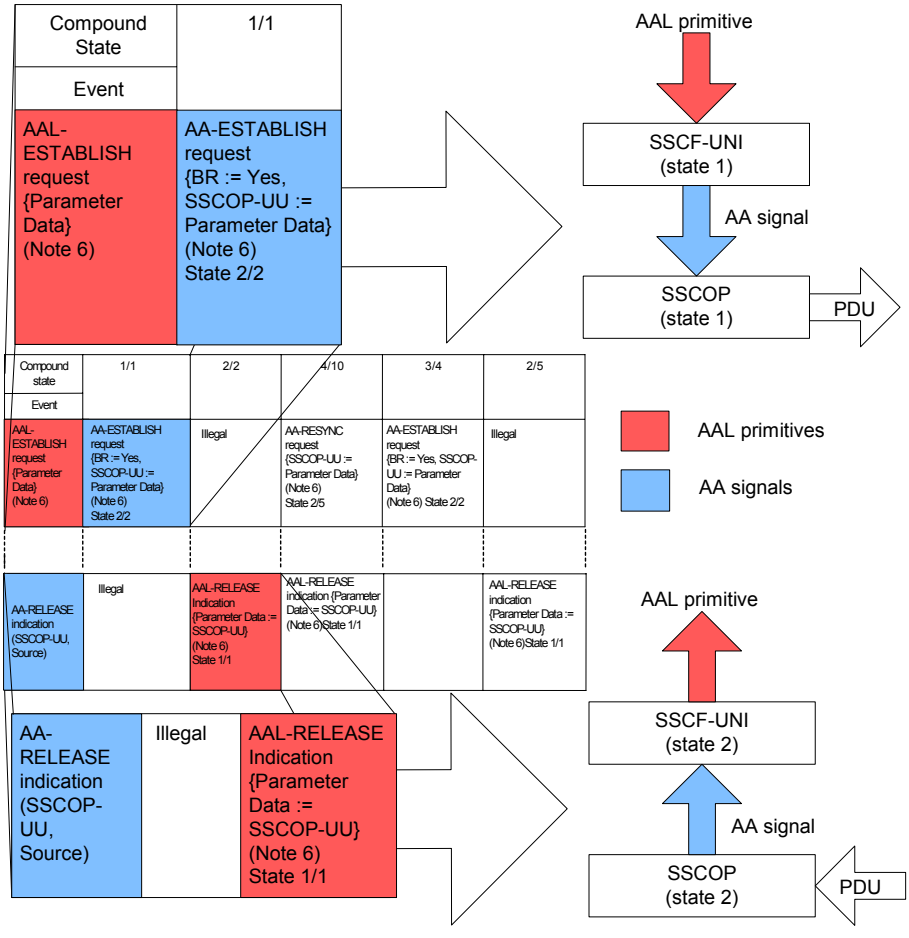


Figure 41. Events for the protocols based on the transition table.

In component testing, the test system is naturally responsible for sending all the primitives, signals and PDUs. In integration testing, however, the signals between SSCF-UNI and SSCOP are sent by either one of the protocols depending on the triggering event. This has to be taken into consideration when designing component tests, so that reusing testware in integration testing is as effortless as possible.

Figure 42 depicts how the component test cases were designed. The signals in the figure are sent by the test system in component testing. The yellow circles in the figure represent a mechanism that was used to block the test system from sending AA signals in integration testing. Blocking the test system from sending a signal corresponds to the blocking briefly discussed in guidelines 1 and 2.

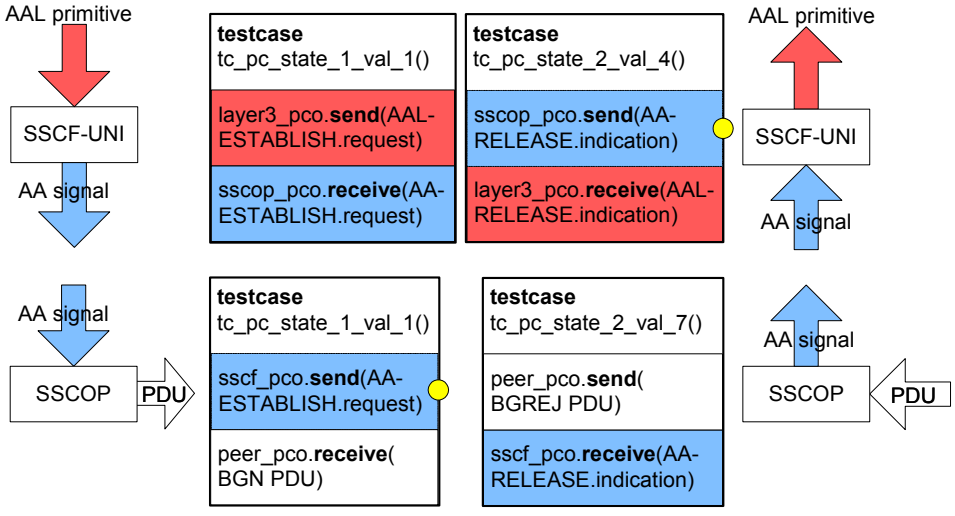


Figure 42. Test case designs for component tests (i.e. for *reuse).*

Figure 43 depicts how the integration test cases were designed. Signals marked with yellow circles are now sent by the protocols, not by the test system. However, the blocked signals are still received by the test system. This is discussed more thoroughly in the next section that describes test implementation.

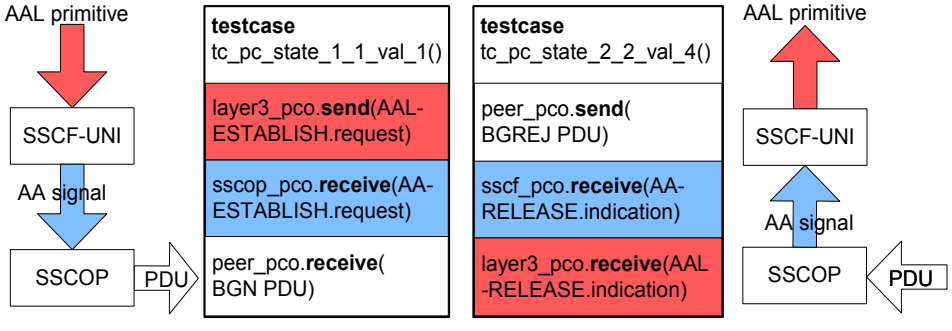


Figure 43. Test case designs for integration tests (i.e. with *reuse).*

7.4 Implementation

During the implementation phase, it was necessary to cut a few corners. The actual implementations of the two protocols were still under development at the time of this case study. This meant that only the TE of the whole TTCN-3 test system (Figure 10) was implemented. Nevertheless, this had only a minor effect in the scope of this study.

Lack of implementation-specific information about the used messages, (primitives, signals and PDUs) resulted in incomplete type definitions that lowered the level of reuse. However, as the tests were not to be executed due to unavailable protocol implementations and incomplete test systems, the control parts of TTCN-3 modules were not implemented, thus increasing the level of reuse (control parts could not have been reused). All in all, it can be stated that despite these limitations, a more than adequate framework for the study of test reuse and use of guidelines was available.

The test systems in the following figures (Figure 44 and Figure 45) depict systems with all the necessary parts for test execution (depicted in Figure 10), although their implementation was left for future work at this stage.

It was decided that there was no need for test system distribution in this study. Therefore, test systems for component and integration testing were constructed according to guideline 2. In component testing, the functionalities of the upper and lower tester were very simple and realized using a single function running on the MTC as illustrated in Figure 44. In the figure, the squares of red, blue and white are the ports for various protocol instances.

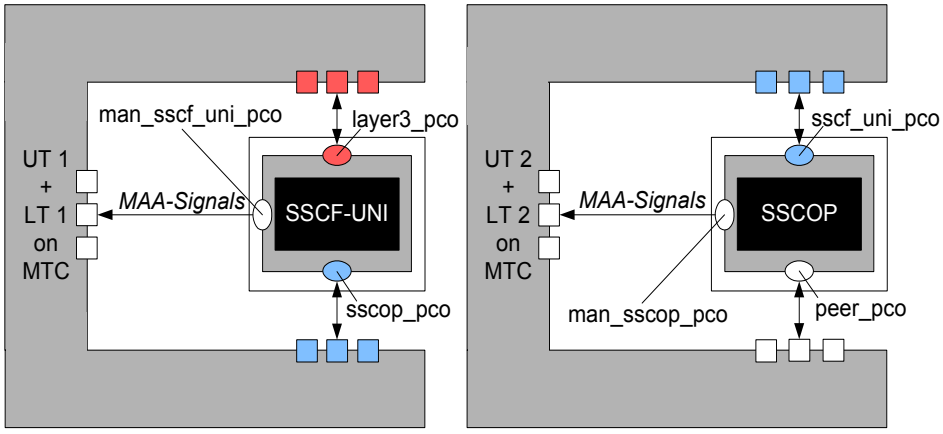


Figure 44. Test systems for component testing of SSCF-UNI and SSCOP.

In integration testing, the upper and lower testers were divided into PTCs as illustrated in Figure 45. The ports for AA signals (*sscop_pco* and *sscf_uni_pco*) are used for checking that the protocol responsible for sending an AA signal to the other protocol actually sends it. Hence, these ports provide intermediate results helpful in interpreting failed or erroneous test cases.

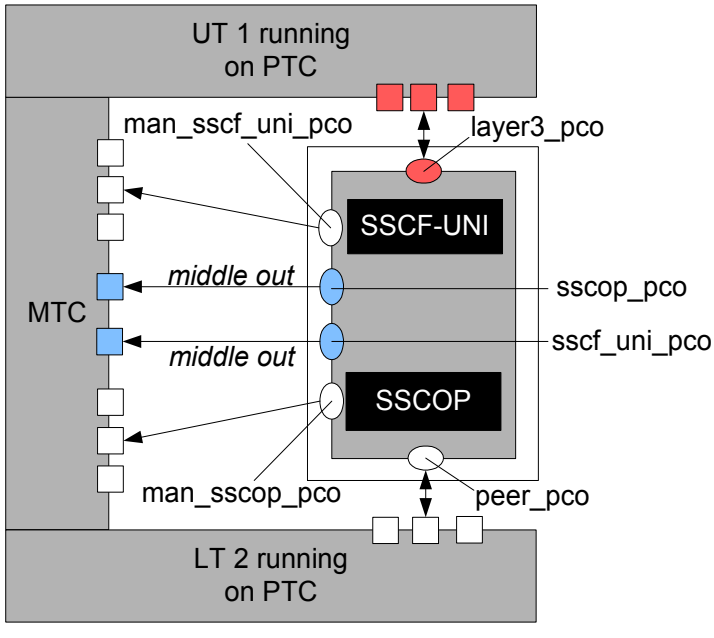


Figure 45. Test system for integration testing.

```
module sscf_uni_lib {
  :
  type component sscf_uni_tester {
    port layer3_sscf_uni_pco layer3_pco[MAX_INSTANCES];
    port sscf_uni_sscop_pco sscop_pco[MAX_INSTANCES];
    port sscs_management_sscf_uni_pco man_sscf_uni_pco[MAX_INSTANCES];
    var t_role role := INITIATOR; // component variable
    timer T1 := 20.0;
  }
  function set_sscf_uni_role(in t_role a_role) runs on sscf_uni_tester {
    role := a_role;
  }
  :
  function request_initialization_indication_sscop(integer a_connection_id,
  template t_sscop_uu a_sscop_uu) runs on sscf_uni_tester {
    :
    if(role == INITIATOR) {
      sscop_pco[a_connection_id].send(aa_est_ind);
    }
    :
    [] layer3_pco[a_connection_id].receive(aal_est_ind) {
      :
    }
    :
  } // end of function request_initialization_indication_sscop
  :
}
module sscf_uni_testcases {
  import from sscf_uni_lib all;
  :
  testcase tc_pc_state_1_val_5() runs on sscf_uni_tester system sscf_uni_tsi {
    : // mappings, variables, etc.
    request_initialization_indication_sscop(connection, sscop_uu);
    : // unmapping
  }
  :
}
module sscf_uni_sscop_integration_testcases {
  import from sscf_uni_lib all;
  :
  testcase tc_pc_state_1_1_val_3() runs on sscf_uni_sscop_tester
  system sscf_uni_sscop_tsi {
    :       // mappings, variables, etc.
    c_upper.start(set_sscf_uni_role(RECEIVER)); // set block
    c_upper.done;
    :
    c_upper.start(request_initialization_indication_sscop(connection,
    sscop_uu)); // reusing the body
    : // unmapping
  }
  :
}
```

Figure 46. Reusing the body.

Using a single function (as noted in guideline 2 and in section 7.3) meant that a specific blocking mechanism had to be used to alternate test behavior so that tests could be reused in integration testing. Test behavior was alternated by changing the value of a test component variable and by using a selection structure as discussed in guidelines 5 and 6, respectively. Test behavior was encapsulated in a reusable test body according to guideline 3. Preambles were also used as recommended in guideline 3, however, their potential for reuse was considerably lower than those of test bodies, and in most cases the effort of reusing them was higher than creating a specific solution for integration testing. The code fractions of the actual implementation in Figure 46 illuminate the matter.

In the code fraction in Figure 46, a test component type named *sscf_uni_tester* is defined in the module *sscf_uni_lib*. The type definition is used for the MTC in component testing and the PTC (UT 1) in integration testing. The function *set_sscf_uni_role()* is used to change the value of the component variable in order to alternate test behavior defined in the function *request_initialization_indication_sscop()*. In test case *tc_pc_state_1_val_5()*, the function is used with the default value of the test component variable (*INITIATOR*), whereas in test case *tc_pc_state_1_1_val_3()*, the default value of the variable is changed so that the test system is blocked from sending the AA signal.

Common type definitions were used as a base for template instantiations for various primitives, signals and PDUs as recommended in guideline 7. Use of common type definitions is illustrated n the code fraction in Figure 47.

Guideline 9 was used to modularize tests. Test cases for SSCF-UNI and SSCOP and integrated protocols were placed in their respective modules. Preambles and test bodies for the protocols were placed in two modules. Type definitions for signals and primitives were placed in their respective modules and common type definitions for the two protocols in one module.

```
module sscf_uni_sscop { // definitions for the interface in between
  :
  type enumerated t_sscf_uni_sscop_signal_type {
    AA_EST_REQ, AA_EST_IND, ... , MAA_UNITDATA_REQ, MAA_UNITDATA_IND
  };
  :
  // AA-ESTABLISH type (request, response, indication, confirmation)
  type record t_aa_est {
    t_sscf_uni_sscop_signal_type   aa_signal,
    t_sscop_uu                     sscop_uu,
    t_br                           br          optional
  }
  :
}
module sscf_uni_lib {
  import from sscf_uni_sscop all;
  :
  function request_initialization_indication_sscop(integer a_connection_id,
  template t_sscop_uu a_sscop_uu) runs on sscf_uni_tester {
    // templates for AA-ESTABLISH.indication and AA-ESTABLISH.response
    var t_aa_est aa_est_ind := {AA_EST_IND, valueof(a_sscop_uu), omit};
    var t_aa_est aa_est_rsp := {AA_EST_RSP, SSCOP_NULL, YES};
    :
  }
  :
}
```

Figure 47. Use of common type definition for AA signals.

7.5 Analysis of Results

As discussed in section 7.4, a couple of factors set limitations to the test implementation. However, from the beginning it was clear that the grounds for good results in reuse levels were evident. As stated in section 5.1.1, vertical reuse has three virtues: narrow domain, low variability of problems and static interfaces. In this case, narrow domain was comprised of component and integration tests for the two protocols. The problems were the conformance requirements stated for the two protocols in their respective recommendations, and there was no variability in them. This means that the protocols should conform against given recommendations regardless of the context that they are used in. Finally, even though protocol implementations were unavailable, it was assumed with great confidence that their interfaces would be static.

7.5.1 Cost of Development For Reuse

The Relative Cost of Writing Reusable Software (RCWR) presented in section 3.6 describes the extra effort that is needed to make reusable software compared to software that is meant for one time use only. In this case study, the extra effort was caused by

- analysis of reuse potential, and
- designing and implementing the blocking mechanism discussed in sections 7.3 and 7.4.

Usually the extra effort includes such activities as

- analysis of potential reusers,
- extra documentation for reusers,
- reuser support in case of problems, and
- maintenance of the reusable components.

Therefore, based on the two lists, it is clear that the value of RCWR should be considerably lower than those usually presented in industrial cases. Furthermore, it should be noted that those industrial values for RCWR presented in section 3.6 are all based on reuse of software, not testware. What really made it difficult to estimate the actual value of RCWR was the lack of experience in creating corresponding tests without reuse. Nevertheless, the value of RCWR was estimated to be as low as 1.1, which is significantly low when compared to the RCWR value 1.5 recommended by Poulin and Caruso [35]. The low value of RCWR was influenced by the following factors:

- the creator and reuser were the same person,
- well bounded and limited scope of reuse, and
- simple solutions for reuse, based on previously created guidelines.

7.5.2 Cost of Development With Reuse

The Relative Cost of Reusing Software (RCR) presented in section 3.6 describes the effort of reusing software compared to effort of creating it from start to finish. The effort of reusing in this case study was caused by

- importing the necessary definitions, and
- reusing preambles and bodies by means of function calls and the use of the implemented blocking mechanism.

Usually the effort of reusing is caused by such activities as

- identifying the parts that can be created *with* reuse,
- searching for candidate components,
- evaluating candidate components, and
- investigating and adapting the best component.

RCR value was estimated to be as low as 0.05. Similarly to the estimated value of RCWR, the estimated value for the RCR is considerably smaller than those presented in industrial cases or recommended by Poulin and Caruso (0.2) [35]. However, such a low value is justified based on the same factors that influenced the low value of RCWR.

In addition, as Bardo et al. have noted, in a very homogenous environment the values of RCR and RCWR can be very low compared to general industrial values [38].

7.5.3 Level of Reuse

Reuse percent was studied in component and in integration testing. In component testing, the reuse percent indicates the level of reusable testware created *for* reuse, whereas in integration testing the reuse percent indicates the level of reusable testware created *with* reuse. Obviously the number of lines created *for* reuse in component testing is the same as the number of lines reused in integration testing. The values for reuse metrics presented in section 3.6 are as follows:

- RSI = 845
- SSI (component testing) = 653
- SSI (integration testing) = 528.

This results in a total value of 2026 lines of source code (i.e. source instructions). Reuse percents for component and integration testware based on the equation (1) in section 3.6 are

$$\text{Reuse Percent (component testware)} \approx 56\%$$

$$\text{Reuse Percent (integration testware)} \approx 62\%.$$

These results indicate that 56 percent of the implemented testware in component testing was created *for* reuse and 62 percent in integration testing *with* reuse. However, if all the test purposes with valid messages for SSCF-UNI and SSCOP had been implemented, the level of reuse in component testware would have been significantly lower.

If the 2026 lines of code would have been produced without reuse (WR), then according to the equation (6) in section 3.6, the total cost without reuse is

$$\text{WR} = 2026 \times NCC.$$

The total development costs of *for* reuse, based on the equation (7) in section 3.6 and RCWR values of 1.5 (recommended by Poulin and Caruso [35]) and 1.1 (estimated value), are:

$$\text{Total cost } \textit{for} \text{ reuse (RCWR} = 1.5) \approx 2598 \times NCC$$

$$\text{Total cost } \textit{for} \text{ reuse (RCWR} = 1.1) \approx 2141 \times NCC.$$

The total development costs of *with* reuse, based on the equation (8) in section 3.6 and RCR values of 0.2 (recommended by Poulin and Caruso [35]) and 0.05 (estimated value), are:

$$\text{Total cost } \textit{with} \text{ reuse (RCR} = 0.2) \approx 1029 \times NCC$$

$$\text{Total cost } \textit{with} \text{ reuse } (\text{RCR} = 0.1) \approx 842 \times NCC.$$

By comparing the Total cost *for* reuse and WR, we can determine the ratio describing how much more it costs to create reusable code compared to code created for one time use only:

$$(\text{RCWR} = 1.5) \rightarrow \frac{2598 \times NCC}{2026 \times NCC} \approx 1.28$$

$$(\text{RCWR} = 1.1) \rightarrow \frac{2141 \times NCC}{2026 \times NCC} \approx 1.06.$$

By comparing the Total cost *with* reuse and WR we can determine the ratio describing how much less it costs to create reusable code compared to code created for one time use only:

$$(\text{RCR} = 0.2) \rightarrow \frac{1029 \times NCC}{2026 \times NCC} \approx 0.51$$

$$(\text{RCR} = 0.05) \rightarrow \frac{842 \times NCC}{2026 \times NCC} \approx 0.42.$$

All in all, the results of this case study are very encouraging. In particular, the low values of RCR and RCWR (even though they are estimated values) indicate that test reuse has lots of potential. However, even when considerably higher values of RCR and RCWR were used, the calculated total costs were still highly comparable and incentive. From a subjective point of view, test reuse seems to provide high productivity gains with very little extra effort.

7.5.4 Use of Guidelines

When evaluating the use of guidelines it should be noted that the guidelines were not created specifically for the case study, but to support the creation of reusable TTCN-3 tests in general. This means that the guidelines were not designed specifically for conformance testing, but for all types of testing in general.

However, during the case study, refinements to the guidelines were made based on the experiences of applying them into practice. The fact that some guidelines were not used in the case study does not mean that they are unhelpful or ineffective to use, but that they simply were not as applicable in this case study as those that were used. In addition, few of the guidelines were exclusionary, meaning that if a selection was made to apply one of the guidelines then some other was excluded.

As noted in section 7.5.1, one of the reasons for a low RCWR value in the case study was the use of predefined guidelines. Design and specification of reusable TTCN-3 code was considerably easier since the reusability issues had been contemplated beforehand. If the order had been reversed so that the case study had been completed before creating any guidelines for reusable TTCN-3 code, the amount and coverage of the guidelines would have been significantly smaller.

Nevertheless, the case study had its own impact on guidelines as well, especially on guidelines 1 and 2 dealing with the test system architecture. The impact of architecture on code reusability had previously been issued in only one guideline (guideline 1). During the case study, new aspects of the architectures relevance to reusability were discovered that led to the splitting of guideline 1 into two guidelines (guidelines 1 and 2). During the implementation phase, a few tips and tricks were discovered that led to updating and refining some of the guidelines. Table 5 summarizes the use of guidelines and their refinements.

Table 5. Use of guidelines in the case study.

Guideline	**Used** (Yes / No)	**Excludes** (Guideline)	**Refinements based on case study** (M=Major, m=minor, n=none)
1	No	2	M
2	Yes	1	M
3	Yes	-	n
4	No	-	n
5	Yes	-	m
6	Yes	-	n
7	Yes	-	n
8	No	-	n
9	Yes	10	m
10	No	9	n

When the results of the case study were presented at a meeting among project partners, they raised the question of creating more specific guidelines for protocol and conformance testing. In addition, there was a keen interest in applying test reuse in domain than the one presented in this work. This will hopefully lead to future work in the field of TTCN-3 test reuse.

8. Discussion

This chapter discusses how the objectives set for this work were reached, the importance of the work and the prospects for future work.

The first objective was to develop guidelines for reusable TTCN-3 code supported by literature studies of software reuse and testing, and studies of test reuse done in the past and especially in the TT-Medal project. This objective was successfully reached and ten guidelines for reusable TTCN-3 code were introduced in chapter 6. These guidelines are based on the techniques known from software reuse, TTCN-3 test system characteristics, test suite and test case structures and the language features of TTCN-3.

The second objective was to apply predefined guidelines in a case study where reusable testware was created with the help of the guidelines. This objective was reached as well, as six out of ten guidelines were used in the case study. However, two guidelines out of the six that were applied excluded the use of two others. Hence, it can be stated that six out of eight were actually applied. Application of the guidelines in practice also provided valuable feedback that, in some cases, resulted in updates and refinements to the guidelines. The use of guidelines also had a positive impact on the measured levels and profits of test reuse.

The third objective was to gain experience of TTCN-3 test reuse. This objective was also achieved during the case study, and the experiences were very encouraging, whether measured using mathematical equations or based on personal experience.

The importance of the work to the software industry and science community are arguable. Today, the pressures on software testing are increasing constantly and new ways to test more efficiently and effectively are required, not only by the industrial partners of the TT-Medal project, but all over. Therefore, the results of this thesis will be published as presentations in the meetings among the TT-Medal partners and in the form of conference papers in scientific forums. In addition, it is my firm belief that the concepts of test reuse could be applied in other test scripting languages as well, as already seen in some of the past studies, thus widening the scope of interest for this work. This thesis will also be published in the form of a VTT Publication.

It is important to note that the concept of test reuse is not as well-known as that of software reuse. In addition, as a programming language TTCN-3 is still very young and relatively unknown. Therefore, plenty of work still lies ahead to promote the application of TTCN-3 in testing software systems. However, the work will be easier when the use of TTCN-3 can be backed up with successful studies such as the one conducted in this work.

Some future work on test reuse has already been agreed on. The results of the case study have been presented to the project partners to promote the possible future cooperation on test reuse. In fact, some preliminary actions have already been agreed upon to study test reuse in a domain than the one presented in this thesis. In addition, there have been preliminary discussions on making reusability guidelines specifically for protocol and conformance testing. It would also be interesting to study the reuse of TTCN-3 tests more thoroughly from the horizontal and historical viewpoints.

9. Summary

The purpose of this thesis was to promote the reuse of TTCN-3 test scripts by creating guidelines for reusable TTCN-3 code that were applied in practice in an industrial case study.

First, an introduction to the topic and the motivation and scope of the work was presented. Then, the principles of software testing and software reuse were presented, after which a short introduction of TTCN-3 was given. Test reuse was presented based on the work done in the TT-Medal project and earlier research experience was briefly presented and compared to this work.

After laying out the basis, the actual work was presented. The guidelines, which were based on software reuse techniques, TTCN-3 test system and language characteristics, and test suite and test case structures, were presented. The guidelines were applied in practice in a case study where their applicability was determined and the level and profits of test reuse were measured. More than half of the guidelines were applied and they supported the low costs of reuse. All in all, the reuse levels and the cost estimations obtained in the case study were very encouraging and they will have a positive impact when future actions are decided.

References

[1] Graham, D., Herzlich, P. & Morelli, C. (1995) CAST report, Computer Aided Software Testing. 3rd. ed. London, UK: Cambridge Market Intelligence. 327 p.

[2] NIST National Institute of Standards and Technology (2002). The Economic Impacts on Inadequate Infrastructure of Software Testing. U.S. Department of Commerce, Technology and Administration, Planning Report 02-03, May, 2002. 309 p.

[3] Shea, B. (25.10.2004) Software Testing Gets New Respect. URL: http://www.informationweek.com/793/testing.htm

[4] 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, FR: European Telecommunications Standards Institute. 178 p.

[5] Tests & Testing Methodologies with Advanced Languages (22.10.2004). URL: http://www.tt-medal.org/index.htm

[6] Myers, G.J. (1979) The Art of Software Testing. New York, NY, USA: John Wiley & Sons, Inc. 177 p.

[7] Hetzel, B. (1988) The Complete Guide to Software Testing. Wellesley, MA, USA: QED Information Sciences. 284 p.

[8] Kit, E. (1995) Software Testing in the Real World: Improving the Process. Harlow, Essex, UK: Addison–Wesley. 252 p.

[9] Ould, M.A. & Unwin, C. (1988) Testing in software development. Cambridge, UK: Press Syndicate of the University of Cambridge. 124 p.

[10] Fewster, M. & Graham, D. (1999) Software Test Automation: Effective use of test execution tools. London, UK: ACM Press. 574 p.

[11] Koomen, T. & Pol, M. (1999) Test Process Improvement: A practical step-by-step guide to structured testing. Harlow, Essex, UK: Addison-Wesley. 215 p.

[12] Jorgensen, P.C. (1995) Software Testing: A Craftsman's Approach. Boca Raton, FL, USA: CRC Press. 254 p.

[13] Korhonen, J., Salmela, M. & Kalaoja, J. (2000) The reuse of tests for configured software products. Espoo: VTT Publications 406. 98 p. + app. 28 p. http://www.vtt.fi/inf/pdf/publications/2000/P406.

[14] Latvakoski, J. (1997) Integration test automation of embedded communication software. Espoo: VTT Publications 318. 124 p.

[15] Patton, R. (2001) Software Testing. Indianapolis, IN, USA: SAMS. 389 p.

[16] Binder, R.V. (1999) Testing Object-Oriented Systems: Models, Patterns and Tools. Reading, MA, USA: Addison–Wesley. 1191 p.

[17] Beizer, B. (1990) Software Testing Techniques. New York, NY, USA: Van Nostrand Reinhold. 550 p.

[18] IEEE Std 601.12-1990 (1990). IEEE Standard Glossary of Software Engineering Terminology. New York, NY, USA: The Institute of Electrical and Electronics Engineers, Inc. 84 p.

[19] ISO/IEC 9646-1 (1994). Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1: General concepts. Genève, CH: IEC. 46 p.

[20] Sarikaya, B. (1993) Principles of Protocol Engineering and Conformance Testing. New York, NY, USA: Ellis Horwood. 502 p.

[21] Schieferdecker, I. & Grabowski, J. (2000) Conformance Testing with TTCN. Telektronikk, 96 (4), pp. 85–95. ISSN 0085-7130

[22] Mäntyniemi, A. & Mäki-Asiala, P. (2004) Improving Efficiency of Testing with Test Reuse: Development of Reusable Test Assets. In: Proceedings of the First International Workshop on Quality Assurance in Reuse Contexts, August 30th – September 2nd, Boston, MA, USA. Pp. 11–18.

[23] NATO Communications and Information Systems Agency (1991). Standard for Management of a Reusable Software Component Library. Brussels, BE: NATO Communications and Information Systems Agency, Vol. 2. 65 p.

[24] Jacobson, I., Griss, M. & Johnsson, P. (1997) Software Reuse – Architecture, Process and Organisation for Business Success. New York, NY, USA: Harlow: Addison–Wesley. 497 p.

[25] McClure, C.L. (1997) Software Reuse Techniques: Adding Reuse to the System Development Process. Upper Saddle River, NJ, USA: Prentice Hall, cop. 350 p.

[26] IEEE Std 1517-1999 (1999). IEEE Standard for Information Technology – Software Life Cycle Processes – Reuse Processes. New York, NY, USA: The Institute of Electrical and Electronics Engineers, Inc. 43 p.

[27] Glass. R.L. (1998) Reuse: What's Wrong with This Picture? Software, IEEE, Vol. 15, Iss. 2, March/April, pp. 57–59.

[28] Karlsson, E.-A. (1995) Software Reuse: A Holistic Approach. New York, NY, USA: John Wiley & Sons, Inc. 510 p.

[29] Glass, R.L. (17.8.2004) What's Wrong with Software Reuse?
URL: www.stickyminds.com/se/S2731.asp.

[30] Morisio, M., Michel, E. & Tully, C. (2002) Success and Failure Factors in Software Reuse. IEEE Transactions on Software Engineering, Vol. 28, Iss. 4, pp. 340–357.

[31] Crnkovic, I. (2001) Component-based Software Engineering – New Challenges in Software Development. Sofware Focus, Vol. 2, Iss. 4, pp. 127–133.

[32] Szyperski, C. (1997) Component Software, Beyond Object Oriented Programming. Harlow, Essex, UK: Addison–Wesley Longman Limited. 411 p.

[33] D'Souza, D.F. & Wills, A.C. (1998) Objects, Components, and Frameworks with UML, The Catalysis Approach. Reading, MA, USA: Addison–Wesley Longman, Inc. 785 p.

[34] Möller, K.H. & Paulish, D.J. (1993) Software metrics: a practitioner's guide to improved product development. London, UK: Chapman & Hall Computing. 257 p.

[35] Poulin, J.S. & Caruso, J.M. (1993) A Reuse Metrics and Return on Investment Model. In: Advances in Software Reuse: Proceedings of the Second International Workshop on Software Reusability, Lucca, IT, 24–26 March. Pp. 152–166.

[36] Favaro, J. (1991) What price reusability? A case study. ACM SIGAda Ada Letters, Vol. 11, Iss. 3, pp. 115–124.

[37] Lim, W.C. (1994) Effects of reuse on quality, productivity, and economics. IEEE Software, Vol. 11, Iss. 5, pp. 23–30.

[38] Bardo, T., Elliot, D., Krysak, T., Morgan, M., Shuey, R. & Tracz, W. (6.10.2004) CORE: A Product Line Success Story. URL: http://www.stsc.hill.af.mil/crosstalk/1996/03/Core.asp.

[39] Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A. & Willcock, C. (2003) An Introduction to the Testing and Test Control Notation (TTCN-3). Computer Networks: The International Journal of Computer and Telecommunications Networking, Vol. 42, Iss. 3, pp. 375–403.

[40] Schieferdecker, I. & Vassiliou-Gioles, T. (2003) Realizing distributed TTCN-3 test systems with TCI. In: Proceedings of the 15th IFIP International Conference on Testing Communicating Systems, TestCom 2003, May 26–28, Cannes, FR. Pp. 95–109.

[41] Schieferdecker, I. (2004) TTCN-3 Tutorial. Test & Testing Methodologies with Advanced Languages Seminar, 26th of March, Oulu, FI.

[42] Dai, Z.R., Grabowski J. & Neukirchen H. (2002) Timed TTCN-3 – A Real-Time Extension For TTCN-3. In: Proceedings of the IFIP 14th International Conference on Testing of Communicating Systems, TestCom 2002, March, 19–22, 2002, Berlin, DE: Kluwer Academic Publishers. Pp. 407–424.

[43] Pulkkinen, P. (2004) Mapping C++ Data Types into a Test Specification Language. University of Oulu, Department of Electrical and Information Engineering, Oulu, FI. Master's Thesis. 83 p.

[44] 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, FR: European Telecommunications Standards Institute. 55 p.

[45] ETSI ES 201 873-6. (2003). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI). Sophia Antipolis Cedex, FR: European Telecommunications Standards Institute. 106 p.

[46] Mäki-Asiala, P., Kärki, M., Mäntyniemi, A., Lehtonen, D., Schieferdecker, I. & Vouffo, A. (2004) Requirements of Reusable TTCN-3 Tests (1.0). Technical report in project: Test & Testing Methodologies with Advanced Languages (TT-Medal), Oulu, FI. 74 p.

[47] Ruuska, P. & Kärki, M. (2004) TTCN-3 Language Characteristics in Producing Reusable Test Software. In: Proceedings of the 8th International Conference on Software Reuse: Methods, Techniques and Tools: ICSR 2004, Madrid, ES, July 5–9. Pp. 49–58.

[48] Gao, J.Z. (1999) Testing Component-Based Software, STARWEST'99, San Jose, CA, USA, June. 19 p.

[49] ETSI Technical Report 190 (1995). Methods for Testing and Specification (MTS); Partial and multi-part Abstract Test Suites (ATS); Rules for the context-dependent reuse of ATSs. Sophia Antipolis Cedex, FR: European Telecommunications Standards Institute. 56 p.

[50] Smith, E.G. (2001) Designing Reusable Test Automation “The Sequencer”. In: Software Testing Analysis & Review STARWEST 2001, October 29th – November 2nd, San Jose, CA, USA. 12 p.

[51] Hörnstein, J. & Edler, H. (2002) Test Reuse in CBSE Using Built-in Tests. In: Proceedings of the Workshop on Component-based Software Engineering, Composing systems from components. Lund, SE. Pp. 11–14.

[52] Johnston, A.B. (2001) SIP: understanding the session initiation protocol. Boston, MA, USA: Artech House. 201 p.

[53] ITU-T Q.2110 (1994). B-ISDN ATM Adaptation Layer – Service Specific Connection Oriented Protocol (SSCOP). Geneva, CH: International Telecommunication Union. 99 p.

[54] ITU-T Q.2130 (1994). B-ISDN SIGNALLING ATM Adaptation Layer – Service Specific Coordination Function (SSCF) for support of signalling at the user network interface (SSFC at UNI). Geneva, CH: International Telecommunication Union. 58 p.

Published by

Series title, number and report code of publication

VTT Publications 557
VTT–PUBS–557

Author(s)
Mäki-Asiala, Pekka

Title

Reuse of TTCN-3 code

Abstract

Today, the growing size and complexity of software along with decreasing development times causes tremendous challenges to software testing. This has driven the whole software industry to seek new ways to test more efficiently and effectively.

Software reuse has been practiced for decades and successful industrial studies have demonstrated such profits as increased productivity and quality as well as decreased development times and costs. This raises the question of whether software reuse could be applied to a testing context as well.

This work studies the reuse of tests that are created with a new test specification and implementation language TTCN-3 (Testing and Test Control Notation). In order to apply reuse into a testing context, a set of guidelines for reusable TTCN-3 code is presented. These guidelines are based on the techniques familiar from software reuse, TTCN-3 test system and language characteristics, and on some of the specifics of software testing. Applicability of the guidelines, and the level and profits of TTCN-3 test reuse are determined in a case study. The case study plainly demonstrates that the majority of the guidelines were successfully applied and that they had a positive impact on measured levels and profits of reuse. The overall results, experiences and impressions of TTCN-3 test reuse during this work were very encouraging and will hopefully lead to future projects in areas of test reuse.

Keywords
software testing, software reuse, test reuse

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

ISBN			Project number
951–38–6431–6 (soft back ed.) 951–38–6432–4 (URL:http://www.vtt.fi/inf/pdf/)			E3SU00131
Date	Language	Pages	Price
January 2005	English, Finnish abstr.	112 p.	C
Name of project		Commissioned by	
Series title and ISSN VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)		Sold by VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 20 722 4404 Fax +358 20 722 4374	

Julkaisija

Julkaisun sarja, numero ja raporttikoodi

VTT Publications 557
VTT–PUBS–557

Tekijä(t)
Mäki-Asiala, Pekka

Nimeke

TTCN-3-koodin uudelleenkäyttö

Tiivistelmä

Ohjelmistojen koon ja kompleksisuuden kasvaminen ja samanaikainen kehitys-ajan lyhentyminen aiheuttavat ohjelmistotestaukselle suuria haasteita. Tämä suuntaus on pakottanut ohjelmistoteollisuuden etsimään uusia keinoja testauksen tehostamiseksi.

Ohjelmistojen uudelleenkäyttöä on harjoitettu vuosikymmenien ajan. Uudelleenkäytön menestyksekkään soveltamisen on huomattu tarjoavan merkittäviä etuja, kuten tuottavuuden ja laadun parantumista, kehityskulujen vähenemistä ja kehitysajan lyhentymistä. Tämä herättää kysymyksen uudelleenkäytön soveltamisesta myös ohjelmistotestauksessa.

Tässä työssä tutkittiin TTCN-3-kielellä luotujen testien uudelleenkäyttöä. Tätä varten luotiin erityiset testien uudelleenkäyttöä edistävät ohjeet, jotka pohjautuvat tunnettuihin ohjelmistojen uudelleenkäyttötekniikoihin, TTCN-3-testijärjestelmän ja -kielen ominaisuuksiin ja ohjelmistotestauksen erityispiirteisiin. Ohjeiden soveltuvuutta ja testien uudelleenkäyttöä arvioitiin tapaustutkimuksessa, joka osoitti ohjeiden hyödyllisyyden saavutetuissa tuloksissa. Yleisvaikutelma tuloksista ja kokemuksista oli rohkaiseva, mikä toivottavasti heijastuu tulevaisuuteen testien uudelleenkäyttöä tutkivina jatkohankkeina.

Avainsanat
software testing, software reuse, test reuse

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

ISBN	Projektinumero
951–38–6431–6 (nid.) 951–38–6432–4 (URL: http://www.vtt.fi/inf/pdf/)	E3SU00131

Julkaisuaika	Kieli	Sivuja	Hinta
Tammikuu 2005	Englanti, suom. tiiv.	112 s.	C

Projektin nimi	Toimeksiantaja(t)

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. 020 722 4404 Faksi 020 722 4374

This work studies the reuse of tests that are created with a new test specification and implementation language TTCN-3 (Testing and Test Control Notation). In order to apply reuse into a testing context, a set of guidelines for reusable TTCN-3 code is presented. These guidelines are based on the techniques familiar from software reuse, TTCN-3 test system and language characteristics, and on some of the specifics of software testing. Applicability of the guidelines, and the level and profits of TTCN-3 test reuse are determined in a case study. The case study plainly demonstrates that the majority of the guidelines were successfully applied and that they had a positive impact on measured levels and profits of reuse. The overall results, experiences and impressions of TTCN-3 test reuse during this work were very encouraging and will hopefully lead to future projects in areas of test reuse.

Tätä julkaisua myy

VTT TIETOPALVELU
PL 2000
02044 VTT
Puh. 020 722 4404
Faksi 020 722 4374

Denna publikation säljs av

VTT INFORMATIONSTJÄNST
PB 2000
02044 VTT
Tel. 020 722 4404
Fax 020 722 4374

This publication is available from

VTT INFORMATION SERVICE
P.O.Box 2000
FIN–02044 VTT, Finland
Phone internat. +358 20 722 4404
Fax +358 20 722 4374

ISBN 951–38–6431–6 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6432–4 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)

Pekka Mäki-Asiala