LAPPEENRANTA UNIVERSITY OF TECHNOLOGY
FACULTY OF TECHNOLOGY MANAGEMENT
DEPARTMENT OF INFORMATION TECHNOLOGY

Visualization of communication system testing logs

The topic of the Master of Science Thesis has been accepted by the head of the Department of Information Technology on October 21st, 2009.

Examiners: Professor Joni-Kristian Kämäräinen
M.Sc. Vesa-Matti Puro

Supervisor: Professor Joni-Kristian Kämäräinen

Lappeenranta, May 25th, 2010

Sampo Ahokas
Koverinkatu 3 B 9
53810 LAPPEENRANTA
sampo.ahokas@iki.fi

ABSTRACT

Lappeenranta University of Technology
Faculty of Technology Management
Department of Information Technology

Sampo Ahokas

Visualization of communication system testing logs

Master of Science Thesis

2010

90 pages, 38 figures and 8 tables.

Examiners: Professor Joni-Kristian Kämäräinen
M.Sc. Vesa-Matti Puro

Keywords: TTCN-3, testing, visualization, log analysis

During the past decades testing has matured from ad-hoc activity into being an integral part of the development process. The benefits of testing are obvious for modern communication systems, which operate in heterogeneous environments amongst devices from various manufacturers. The increased demand for testing also creates demand for tools and technologies that support and automate testing activities. This thesis discusses applicability of visualization techniques in the result analysis part of the testing process.

Particularly, the primary focus of this work is visualization of test execution logs produced by a TTCN-3 test system. TTCN-3 is an internationally standardized test specification and implementation language. The TTCN-3 standard suite includes specification of a test logging interface and a graphical presentation format, but no immediate relationship between them. This thesis presents a technique for mapping the log events to the graphical presentation format along with a concrete implementation, which is integrated with the Eclipse Platform and the OpenTTCN Tester toolchain.

Results of this work indicate that for majority of the log events, a visual representation may be derived from the TTCN-3 standard suite. The remaining events were analysed and three categories relevant in either log analysis or implementation of the visualization tool were identified: events indicating insertion of something into the incoming queue of a port, events indicating a mismatch and events describing the control flow during the execution. Applicability of the results is limited into the domain of TTCN-3, but the developed mapping and the implementation may be utilized with any TTCN-3 tool that is able to produce the execution log in the standardized XML format.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknillistaloudellinen tiedekunta
Tietotekniikan osasto

Sampo Ahokas

Tietoliikennejärjestelmän testauksessa tuotetun tapahtumarekisterin visualisointi

Diplomityö

2010

90 sivua, 38 kuvaa ja 8 taulukkoa.

Tarkastajat: Professori Joni-Kristian Kämäräinen
DI Vesa-Matti Puro

Hakusanat: TTCN-3, testaus, visualisointi, tapahtumarekisterin analysointi
Keywords: TTCN-3, testing, visualization, log analysis

Testaus on vakiinnuttanut asemansa olennaisena osana tietoteknisten järjestelmien elinkaarta aina määrittelystä alkaen. Testauksen yleistyminen on luonut kysyntää myös testausprosessia tukeville ja automatisoiville teknologioille sekä työkaluille. Tämä diplomityö käsittelee visualisoinnin hyödyntämistä testitulosten analysointivaiheessa.

Erityisesti tässä työssä tarkastellaan TTCN-3-testijärjestelmän tuottaman lokin (tapahtumarekisteri) visualisointia. TTCN-3 on standardoitu testitapausten määrittely- ja toteutuskieli. TTCN-3-standardisarja määrittelee rajapinnan, jonka kautta testijärjestelmä tarjoaa lokitapahtumat sekä graafisen esitysmuodon lähdekoodille, mutta suoraa vastaavuutta yksittäisten lokitapahtumien ja graafiseen esitysmuodon välillä ei ole määritelty. Tämä työ esittää menetelmän lokitapahtumien muuntamiseksi graafiseen esitysmuotoon. Työssä esitellään myös ohjelmistokomponentti, joka toteuttaa TTCN-3-lokien visualisoinnin integroituna OpenTTCN Tester-työkaluun ja Eclipse-ympäristöön.

Työn tulokset osoittavat, että TTCN-3-standardisarjasta löytyy sopiva graafinen esitys suurimmalle osalle lokitapahtumista. Jäljelle jäävät lokitapahtumat analysoitiin ja niistä tunnistettiin kolme testitulosten analysoinnin tai visualisointityökalun kannalta merkityksellistä ryhmää: portin viestijonoon lisäyksestä kertovat tapahtumat, niin kutsutut mismatch-tapahtumat sekä suorituksen etenemisestä kertovat tapahtumat. Tulosten sovellettavuus rajoittuu TTCN-3-ympäristöön, mutta tuloksena syntynyt menetelmä sekä työkalu soveltuvat käytettäväksi kaikkien standardinmukaista XML-muotoista TTCN-3-testauslokia tuottavien työkalujen kanssa.

PREFACE

This work was carried out late 2009 and early 2010 at OpenTTCN Ltd, a software company located next to the Lappeenranta University of Technology. I wish to thank Vesa-Matti Puro and OpenTTCN for offering me the chance to write this thesis and for providing a pleasant and flexible working environment during the past few years.

Sincere thanks also to the thesis supervisor and examiner Joni Kämäräinen for the excellent guidance since the day one of this project. I feel that the personal feedback along with the short but concise *Joni's HowTo for writing MSc/BSc thesis* greatly helped me in getting to the right track.

Additional thanks to my colleagues Vesa-Matti Puro and Alexey Mednonogov for their comments and proof-reading my thesis.

I want to also thank my parents, siblings, parents-in-law and brother-in-law for their support and always encouraging me to finish these studies. Finally, a big and warm hug to my loving and caring wife Anni for... well, everything.

Lappeenranta, May 25th, 2010

Sampo Ahokas

Contents

ABBREVIATIONS

CORBA	Common Object Request Broker Architecture
CTMF	OSI Conformance Testing Methodology and Framework
EMF	Eclipse Modeling Framework
ETSI	European Telecommunications Standards Institute
GEF	Eclipse Graphical Editing Framework
GFT	TTCN-3 Graphical Presentation Format
IDE	Integrated Development Environment
IDL	Interface Definition Language
ISO	International Organization for Standardization
MSC	Message Sequence Chart
OSI	Open Systems Interconnection
SIP	Session Initiation Protocol
SUT	System Under Test
TCI	TTCN-3 Control Interface
TCI-TL	TTCN-3 Control Interface - Test Logging Interface
TCP/IP	Transmission Control Protocol / Internet Protocol
TE	TTCN-3 Executable
TRI	TTCN-3 Runtime Interface
TSI	Test System Interface
TTCN	Tree and Tabular Control Notation
TTCN-3	Testing And Test Control Notation version 3
UDP	User Datagram Protocol
XML	eXtensible Markup Language

1 INTRODUCTION

1.1 Background and motivation

These days various communicating devices are a natural part of everyday life. Mobile phones, satellite navigators, computers and other similar electronic devices are the most obvious ones, but the list goes far beyond that. For instance, modern automobiles contain a number of highly sophisticated interconnected electronic devices. Common to all interacting communication systems is that the communication must occur according to a set of well-defined rules. These rules are referred to as a communication *protocol*.

In the real world, simply having a common protocol is not yet a guarantee of interoperability. Implementing a complex communication system is not a trivial task, which leaves plenty of room for errors. *Testing* is one way to detect potential errors in an implementation. The activity of testing an implementation against its specification is called *conformance testing*. However, it is important to note that testing cannot guarantee conformance to a specification since it detects errors rather than their absence.

Looking back in time, the need for standardizing the rules of interaction between heterogeneous interconnected systems was recognized by the International Organization for Standardization (ISO) in 1977. The work on Open Systems Interconnection (OSI) was started [88], and that initiative has resulted to a number of published standards. In the scope of this thesis, one of the standards is particularly interesting. ISO 9646: *Conformance Testing Methodology and Framework* [43] was originally developed to provide a framework for testing OSI systems, but the conformance testing concepts presented there have proven to be robust and have thus been adapted to testing other kinds of communication systems [63].

Testing is a complex process in its own right. Let's assume we have a system that implements a standard protocol and a standardized set of tests is available for the protocol. Without relation to any particular means of testing, we can identify at least the following practical activities that are required in order to test the system:

1. Making the tests executable.
2. Executing the tests against the *System Under Test (SUT)*.

3. Analysing the test results.

The events detected during the execution are recorded to a test execution log. In the context of conformance testing, ISO 9646 defines *conformance log* as a "human-readable record of information produced as a result of a test campaign, which is sufficient to record the observed test outcomes and verify the assignment of test results". A scenario where all tests resulted in a *pass* verdict is often not particularly interesting. After all, the system under test appears to be fine. Otherwise, if any of the tests resulted in a non-pass verdict, user attention is typically required.

Especially if the system is complex and involves multiple interacting components, pinpointing the cause of the problem may not be straightforward. From the perspective of testing tools, this leads to demand for intuitive log presentation formats that aid the user in the various tasks related to analysing the log. Utilizing *visualization* techniques in presentation of the log data is one of the options. In this thesis I take a look at the topic of applying visualization to log data generated by testing of communication systems.

In terms of technologies available for test implementation, the options are endless. Often the preferred choice is a product specifically designed for testing, but in some cases a trivial program written in traditional programming languages, such as C or Python, may be just the right tool. ISO 9646 also defines a test notation called *Tree and Tabular Control Notation (TTCN)* for use in the specification of OSI conformance test suites. Like the conformance testing principles from the standard, also the TTCN notation was considered useful and was adopted to uses beyond testing of OSI systems. However in its original form TTCN was deemed to be too restrictive and complex, which led to a redesigned successor, the *Testing and Test Control Notation (TTCN-3)*, being developed by the European Telecommunication Standards Institute (ETSI) [34].

The practical part of the thesis focuses on TTCN-3 *Graphical Presentation Format (GFT)* [22] and using it for visualization of test execution logs. On concrete level this thesis introduces a prototype of the *Reprise* log analysis and visualization tool.

1.2 Structure of the thesis

After the introduction in Chapter 1, this thesis is structured as follows:

Chapter 2 takes a look at communication systems and related standards such as Open Systems Interconnection. The chapter introduces the concept of layered protocol architecture, around which most of the communication systems are built.

Chapter 3 presents an overview of testing in general and discusses the location of this work and protocol conformance testing in the broader context of software testing.

Chapter 4 gives a brief introduction to information visualization and the motivation behind it.

Chapter 5 introduces the TTCN-3 testing language, which is the technology of choice in the practical part of this thesis.

Chapter 6 discusses the task of constructing the graphical representation based on data received through the TTCN-3 Test Logging interface.

Chapter 7 describes Reprise, the TTCN-3 log analysis and visualization tool developed as a part of this thesis work.

Chapter 8 presents conclusions and ideas for future work.

2 COMMUNICATION SYSTEMS

2.1 What is a communication system?

The ultimate practical goal of this work is to build a software tool for visualizing logs produced by testing of communication systems. Obviously, understanding and knowledge of the problem domain has a vital role in the success of any software project. Thus we begin by describing what is considered to be a communication system in the scope of this work, followed by introduction of some methods for modeling and abstracting communication systems.

To formally define a communication system we could go back to 1920s and take a look at Ralph Hartley's work on *Transmission of Information* [37]. The article presents a theory of information as a measurable quantity based on physical instead of psychological considerations. By eliminating the psychological factors and measuring information in terms of purely physical quantities, the task of transmitting the information over some physical communication channel becomes much more manageable.

Building on the work of Hartley and Nyquist [55], some decades later in 1948 Claude Shannon published a very influential article: *A Mathematical Theory of Communication* [70]. The paper defines the problem of communication as follows:

> *"The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point."*

Indeed, communication is all about transmitting a message to another location. Shannon's work laid out the foundation of all modern communication systems. Most of the paper focuses on the problem of transmitting information reliably over unreliable, noisy communication channels. While reliability of the physical communication channel is certainly crucial and could be subject of testing itself, in the scope of this work it is assumed to be functioning correctly. However, the article presents two ideas that are relevant in this context. First, it helps us define a communication system: the five essential parts of a communication system as presented in [70] and illustrated in Figure 1 are:

1. Information source. Produces a message or a sequence of messages to be transmitted to the receiver.

2. Transmitter. Operates on the message to produce a signal suitable for transmission over the channel.

3. Channel. Merely the medium used for transmitting the signal. For example a pair of wires, a band of radio frequencies or a beam of light.

4. Receiver. Reconstructs the message from the signal. Effectively inverse of what the transmitter does.

5. Destination. Recipient of the message.

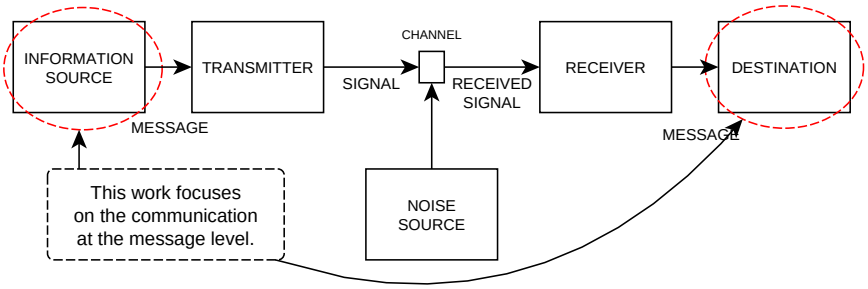


Figure 1: Schematic model of a general communication system

Furthermore, Shannon's article introduced the binary digit, briefly the *bit*, as the fundamental unit of information. Despite not being always obvious to the end-user, this fact still holds true for practically every piece of digital information that exists today. A bit has two possible states. The states may be represented for example by two distinct voltage, current or light intensity levels. In computing, bits are typically presented as numerical digits with possible values being 0 and 1. Additionally bits may be interpreted as logical values true/false or yes/no. Dealing with individual bits is a common task in the type of testing discussed in this thesis, but we never go past the boundary to the physical representation of the bits, due to reasons given in Section 2.2.2.

For this thesis, the definition of a communication system has been adopted from Coulouris [14], who defines a distributed system as *"a system in which hardware and software components located at networked computers communicate and coordinate their actions only by passing messages"*. However, in our case, the networked computers are not necessary physical computers – instead, the key concept is that various components *communicate by passing messages*. If the components are running on the same computer as separate processes, they must still communicate through the network interface. They may not, for example, share data stored in memory.

2.2 Open Systems Interconnection – OSI

2.2.1 Background

The introduction in Chapter 1 briefly mentioned Open Systems Interconnection (OSI) as an effort to standardize networking. More precisely, the objective of OSI is to define a set of standards to enable real open systems to cooperate [42].

An "open system" is a system that communicates with its peers in conformance to all applicable OSI protocol standards. System being open does not imply any particular technology or implementation; it just refers to recognition and support of the applicable standards. The only aspect OSI is thus concerned with is the interconnection of systems. Figure 2 illustrates the fundamental concept on an abstract level: physical media provides the means for connecting several open systems to each other.

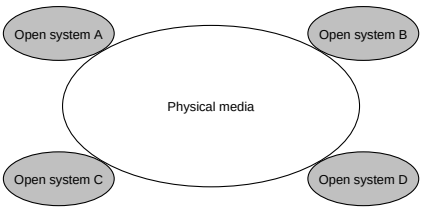


Figure 2: Open systems connected by physical media [42]

OSI was a joint project of the Telecommunication Standardization Sector of International Telecommunication Union (ITU-T) and the International Organization of Standardization (ISO). The work led to dozens of standards being published under the general title "Information Technology - Open Systems Interconnection".

In the context of this thesis, the OSI standard suite contains two relevant concepts. The roots of the TTCN-3 testing language lie in the OSI Conformance Testing Methodology and Framework (CTMF) [43], and the OSI Reference Model [42] is used to describe and abstract communication systems. The TTCN-3 language will be further introduced in Chapter 5, and the next section describes the reference model.

2.2.2 Reference Model

The OSI reference model provides a way to divide a system into smaller parts. The model consists of seven layers, typically illustrated in a vertical sequence as shown in Figure 3.

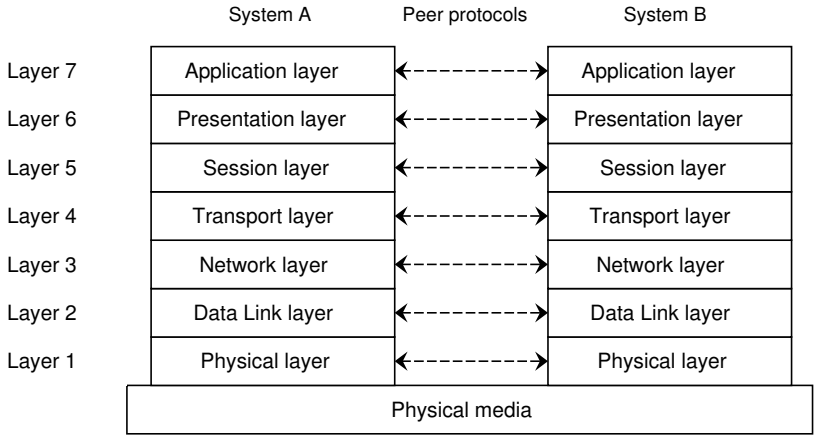


Figure 3: The OSI reference model

The fundamental idea of layering is to divide the problem of running a distributed application into smaller pieces. Layers add functionality in an incremental fashion so that each layer adds value to the set of services provided by the lower layers. Independence of each layer is ensured by defining merely services provided by the layer, without forcing any particular method for performing the services. This allows modifications to the internal implementation of each layer while still offering the same services to the higher layer. This technique can be viewed to be similar to the concepts of programming where a module only publishes its interface to the users, keeping its implementation hidden behind the interface. A brief overview of the layers is given below, starting from the topmost layer [88].

- *Application layer:* This layer is located directly next to the end user and it provides means for distributed applications to communicate with each other. Applications are the ultimate sources and receivers of data that is being exchanged through the network. The remaining layers exist only to support the application layer in the task of enabling intercommunication between applications.
- *Presentation layer:* This layer may offer services that support the application layer

in interpretation of the exchanged data. Practical experience has shown that this layer is of very little use to most applications [74].

- *Session layer:* This layer supports the cooperation between presentation entities. Along with the presentation layer, this layer has not been found to be useful in practice [74].
- *Transport layer:* This layer provides transparent data transfer between session entities of different systems. In other words its task is to hide the complexity of reliable and cost-effective data transfer.
- *Network layer:* Network layer is responsible for exchange of network service data units between transport entities over a network connection. Effectively it relieves the transport layer of routing and switching considerations.
- *Data link layer:* Provides functional and procedural means to connect network entities by establishing, maintaining and releasing data links.
- *Physical layer:* This layer consists of the mechanical, electrical, functional and procedural characteristics for managing physical connections (data circuits) between data link entities.

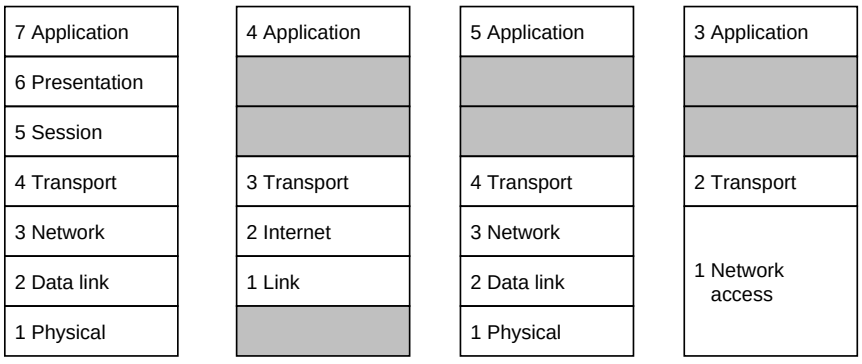
The OSI Conformance Testing Methodology and Framework (CTMF) is applicable to protocols on OSI layers from 2 to 7. Testing of the physical layer protocols would involve either generating or measuring physical signals, which is not fully addressed by the CTMF [44]. Consequently, the kind of testing discussed in this thesis is generally applicable to protocols located above the physical layer in the OSI model. TTCN-3 test systems and their relation to the OSI model are further discussed in Section 5.6.

2.2.3 OSI in a broader context

OSI has also been subject to quite a bit of criticism. One of the objectives OSI had was to define protocols for the entire protocol stack (all of the layers). However, the protocols developed as a part of the OSI work never became popular. The failure has been attributed to reasons such as bad timing, too complex technology, poor implementations and even politics [63, 74]. In contrast, the OSI reference model has proven to be exceptionally useful for discussing networks [74].

Looking at the OSI reference model in the context of Internet and its protocol suite Transmission Control Protocol / Internet Protocol (TCP/IP) some fundamental similarities may be identified. Both are based on the concept of a stack of independent protocols and the functionality of the layers is similar. In both models, the lower layers form a transport provider and the upper layers are application-oriented. However, also differences exist. The OSI model has been carefully designed to make a distinction between the concepts of services, interface and protocol. The service definition tells what the layer does, interface tells how to access the layer and the layer itself selects the protocols it uses to provide the services it is responsible for. In other words, the OSI model makes a clear distinguishion between interface and implementation and the model was engineered before the protocols even existed. Conversely, in the case of TCP/IP the protocols were first and the model was effectively developed as a description of the existing protocols. This makes the TCP/IP model poorly suited for describing other protocol stacks while OSI excels at the task [74].

When modeling networks, a common trend seems to be to view the OSI model somewhat too complex. Particularly, the session and presentation layers are typically omitted. Figure 4 shows a comparison of various networking models found in literature. Although the figure displays the (approximately) equivalent layers vertically aligned between models, it is important to note that the models are not designed to be compatible or to be directly compared.



(a) OSI **(b)** RFC 1122 **(c)** Hybrid [74] **(d)** Simplified [72]

Figure 4: Approximate comparison of layers between models

The full OSI model with seven layers is shown first as Figure 4a. Figure 4b illustrates the TCP/IP model defined in [11]. As can be seen, the TCP/IP model is ignorant of any physical layer considerations. Tanenbaum's book [74] utilizes the OSI model but focuses primarily on TCP/IP and related protocols, and thus presents a hybrid model

better suited for that purpose. This hybrid model, as illustrated in Figure 4c, adds a physical layer to TCP/IP and removes the presentation and session layers from OSI.

Finally, a simplified three layer model from [72] is shown in Figure 4d. This model is based on the view of communication involving three agents: applications, computers and networks. Applications execute on computers and computers are connected to networks. Transferring data from application A to application B first involves transmission of the data from one computer to another computer and then getting it to the receiving application within the computer.

2.3 A simple protocol architecture

From the implementation perspective, each protocol in a layered protocol architecture typically inserts a control header to the data before passing it forward. The header carries control information, such as the address of the recipient. Likewise, each peer protocol at the receiving system removes the respective header and delivers the remaining payload to the layer above it. This is repeated in each layer, all the way up to the receiving application.

Figure 5 illustrates the concept of a protocol architecture in the simplified three layer presented by Stallings [72]. Three protocols are involved in this scenario: an application protocol, a transport protocol and a network procotol.

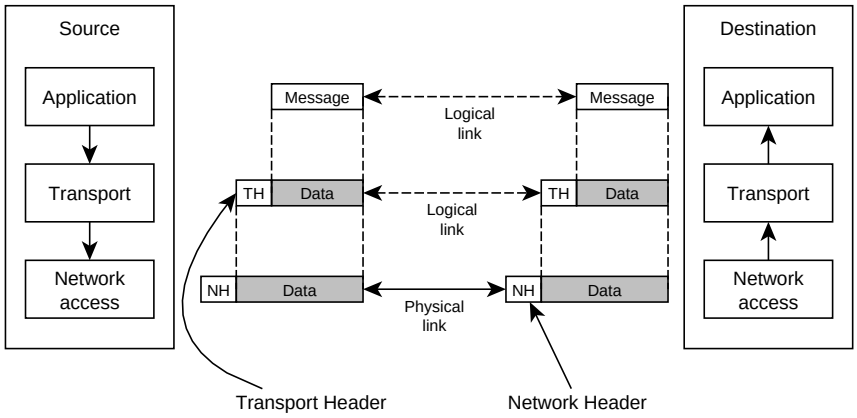


Figure 5: Protocol architecture

The combination of data from the above layer and the control information is referred

to as a protocol data unit (PDU). In this example, as the application sends the message, it is first packaged inside a transport PDU. The transport PDU is then handed to the network protocol, which in turn adds its own header before the physical transmission of the packet. On the destination system the original message travels up through the protocol stack. Each layer will extract the relevant header and pass the message forward based on the control information.

2.4 Real-world example: Session Initiation Protocol

Code examples from the Session Initiation Protocol (SIP) conformance test suite are shown later in this thesis. SIP is an application layer signaling protocol designed for controlling realtime communications on the Internet. The core protocol itself is defined in RFC3261 [67], but the full specification is split to a number of documents. The "Hitchiker's Guide to Session Initiation Protocol" [66] provides an overview of the big picture. SIP works independently of underlying transport and it may run on top of several different transport protocols, including unreliable datagram procols such as the User Datagram Protocol (UDP).

Arguably SIP is an excellent candidate to be used as an example in this context due to several reasons. First, the real-world interoperability leaves currently a lot to be desired. Interoperability issues are common due to lack of clarity in the specification, implementation of a deprecated version of the specification and incomplete or incorrect implementations [62]. This certainly indicates demand for more thorough testing. Furthermore, a conformance test suite written in TTCN-3 language has been published by ETSI [25]. Finally, various free and open implementations, such as Kamailio [50], are available. As a consequence, SIP is well suited for experimenting and learning the use of TTCN-3 in a real-world scenario. Constructing an automated SIP test system in TTCN-3 only requires a TTCN-3 tool and writing an *adapter* for connecting the test system to the system under test – everything else is freely available. TTCN-3 test system construction is further discussed in Section 5.3.

3 TESTING

3.1 What is testing?

The concept of testing was introduced in Section 1.1 by describing a very specific case: protocol conformance testing. However, for instance by performing a literature search on the keyword "testing", one can quickly conclude that testing is a broad term which is used to refer to a wide range of activities with different objectives. We now take a step back and look at testing in the broader context of software testing.

Testing may be performed on various levels ranging from *unit testing* of small units of code to customer *acceptance testing* of a large information system. Similarly, the objective may vary: for example, the test could aim to expose deviations from user requirements, assess conformance to a specification or to evaluate robustness in load conditions. Bertolino [8] suggests that a common denominator is the abstract view that testing always consists of the following two steps:

1. Observing a sample of executions.
2. Giving a verdict over them.

By comparison, Myers defines testing as *"the process of executing a program with the intent of finding errors"* [54]. This definition brings testing closer to programming; after all, most people involved with software have also formed a mental model of testing, largely due to the fact that testing is as old as coding and has been always present in a form or another. Yet, these mental models may differ significantly between individuals, and this has resulted in confused communication among parties involved in a software project. Starting from the 1950s, several periods can be distinguished based on the most influential testing model of the era. In the earliest, debugging-oriented period testing was simply seen as a part of the activities involved with getting the bugs fixed and the application running. During the decades the level of professionalism associated with software testing has increased significantly and testing has been moved towards the beginning of the project life cycle so that it is seen as an integral part of the development activities from the very beginning [32].

The concept of *Validation, Verification and Testing* (VV&T) [1] refers to software quality on a more general level. Validation is concerned about correctness of the software

with respect to user needs and requirements as a whole. On the other hand, verification focuses on a particular phase of the software development life cycle. As a rule of thumb, validation answers the question "Are we building the right product?" and verification answers the question "Are we building the product right?". The verification process provides *objective* evidence, while the evidence obtained during the validation process may not necessarily be as easily measured [40]. Consequently, automatic testing, including the kind of testing discussed in this thesis, belongs to the verification activities.

To summarise, it appears that testing is definitely a part of each software project in a way or another. Simultaneously, testing is still often found to be difficult and confusing part of the picture. In fact this is not a huge surprise considering that due to the growth of testing being split to various periods with unique characteristics, a typical developer not only views testing differently than a typical manager, but the mental models also differ between developers from different generations. The view taken in this thesis is discussed in the following section.

3.2 Protocol conformance testing in a more general context

Research on protocol conformance testing, originally pushed by the pressure of enabling communication, has traditionally been able to proceed on its seemingly privileged trail largely due to the nature of communication protocols. A protocol must have a precise state-based specification of desired behaviour and protocols are typically reactive. This has allowed early development of advanced testing methods. Now, because communication protocols are implemented either as software only or as a combination of hardware and software, it has been suggested that the protocol testing principles have much broader applicability in software testing [9]. This close relation between protocol conformance testing and software testing is also supported by [8]. Earlier protocols were simpler, but with the growth of complexity in the protocols, the gap between software testing and protocol testing has been vanishing recently.

The OSI Protocol Conformance Testing Methodology and Framework (CTMF) has stood the test of time and has remained stable due to the standardization efforts. On the other hand, the common understanding of testing and its terminology has been continuously evolving, leading to confusion and misunderstandings. Therefore, in this thesis, testing theory and terminology are adopted from the stable and mature OSI CTMF [43] and the TTCN-3 standard suite [26].

3.3 Testing strategies: white and black boxes

Testing literature often discusses the box approach to testing. One way to categorize testing is to separate the testing strategies to black-box testing and white-box testing. Unlike many other definitions in the field of testing, these definitions are generally well agreed upon in literature. The descriptions below have not been specifically adopted from any single source.

Black-box testing treats the system under test as a black box, and the tests are designed solely based on the external specification of the system. The test developer should not know, or need to know, anything about the internal implementation of the box. Testing concentrates on finding scenarios where the system under test does not behave according to its specifications. Black-box testing may also be referred to as data-driven or input/output-driven testing.

White-box testing relies on the knowledge of the internals of the system. Test data is derived by examining the program logic. White-box testing may also be called logic-driven testing, glass box testing or structural testing.

The OSI protocol conformance testing methodology is defined as a black-box testing architecture. Tests are developed based on the definition of behaviour required from the system under test. Therefore, this thesis will focus solely on the black-box testing approach.

3.4 Testing process

The ISO 29119 Software Testing Standard is an ongoing work on standardizing software testing. Aim of the project is to "produce one international standard for software testing that covers the entire software testing lifecycle, throughout the analysis, design, development and maintenance of any software product or a system" [45]. Part two [46] describes a generic testing process model that can be used within any software development and testing lifecycle. The test process model presented by the standard is spread around multiple layers. Each of the layers contains a varying number of test processes. Figure 6 illustrates these layers and processes as well as the position of this work in the overall context.

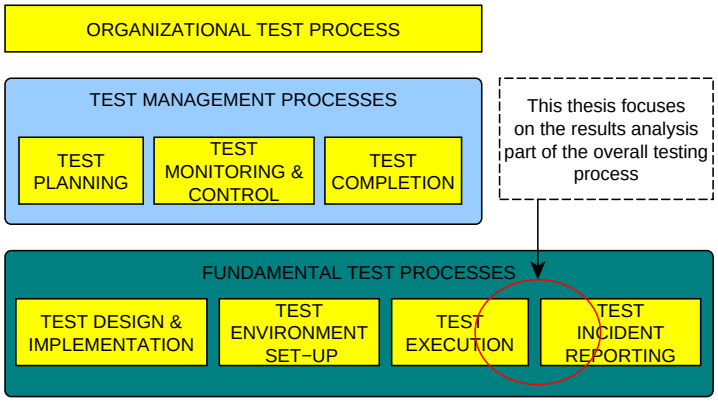


Figure 6: Testing process [46]

Organizational test processes deal with developing and managing test specifications on the organizational level. Common to these processes is that they do not contain project-specific details. By comparison, the test management processes are applicable at the level of an individual project. However, the most interesting processes in the scope of this work are the fundamental test processes, as illustrated in Figure 7.

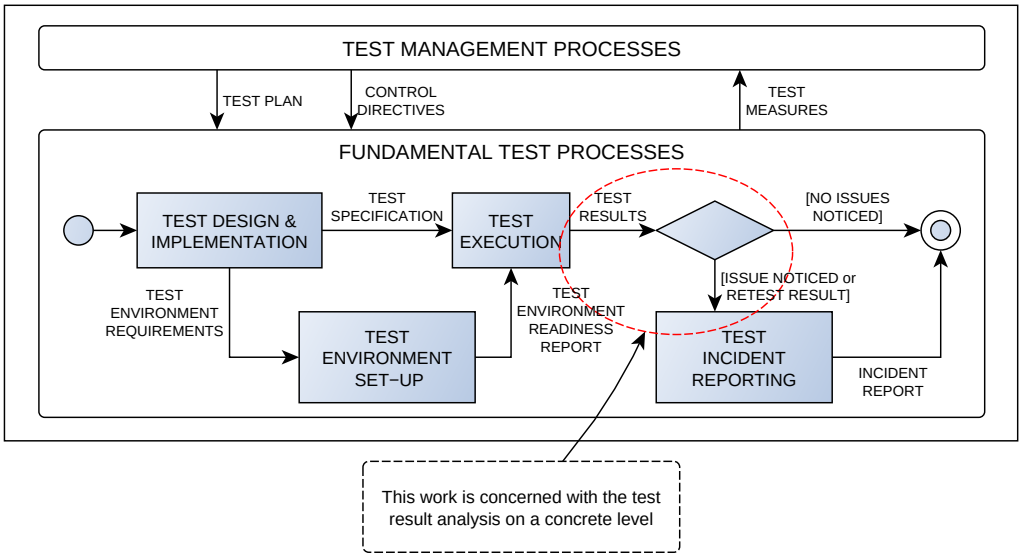


Figure 7: Fundamental testing processes [46]

3.5 Concept of test suite

A test suite, as defined in ISO 9646 [44], is built up in nested hierarchical fashion from a number of identified test units. Figure 8 shows the structure of test suite in a tree format.

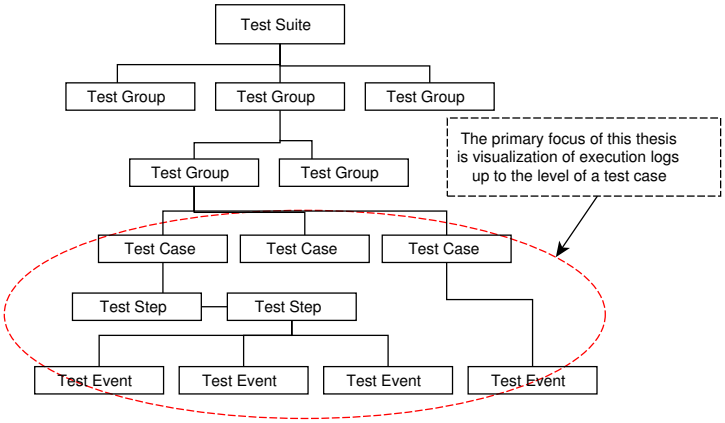


Figure 8: Testing suite structure [51]

The smallest indivisible unit is the *test event*. Typically an event is the transmission or reception of a message. Next incremental unit up from independent events is the *test step*, which is comprised of a number of test events. Defining a test step precisely is difficult, as it is simply a set of test events that form an identifiable subdivision of a test case. An analogy to programming languages would be a subroutine.

Test case is the fundamental building block of a *test suite*, always identified by a test purpose. Test case leads to a verdict: pass, fail or inconclusive. Pass verdict is given if the observed test outcome does not give evidence of non-conformance and no invalid test events have been detected. Fail verdict is given when the observed test outcome either demonstrates non-conformance with respect to the conformance requirements or contains at least one invalid test event with respect to the relevant specification. Inconclusive verdict indicates outcome which cannot be presented with either pass or fail verdict. TTCN-3 has also added an error verdict to the list of possible outcomes. A TTCN-3 test system (described later in Section 5.3) is a complex system where internal system errors may also occur. The error verdict is reserved for the test system and it is assigned to a test case if a dynamic test case error occurs during the execution.

Test suite serves as the root of the tree. Test suite can range from a trivial case of a single test case and a single test event to a complex test suite with several *test groups*, each consisting of multiple test cases and so on. The reason for focusing on visualizations

that cover at most the execution of a single test case is obvious based on the initial motivation for this work: to aid the user in pinpointing the location of an error found in testing. Final test verdicts are assigned for test cases, and therefore the test case is always the first subject of investigation.

Besides the above presented division to test steps and further to test events, a test case may also be considered to consist of the following three components [51]:

1. *Preamble.* Responsible for preparing the system under test (SUT) for the current test case. In other words, the preamble sets the SUT into an appropriate starting state.
2. *Test Body.* The Test Events that comprise the test itself.
3. *Postamble.* The SUT may have to be returned to a specific state after the test body has been executed.

These components may be explicitly identified as test steps, but they do not have to be [44]. Often multiple test cases may benefit from using the same preamble or postamble. Additionally, if the end state of the test body is not unique, it has to be checked by a verification step before the postamble. This may add an optional verification step to the end of the test case. Thus, the test case structure may be represented using the state machine shown in Figure 9.

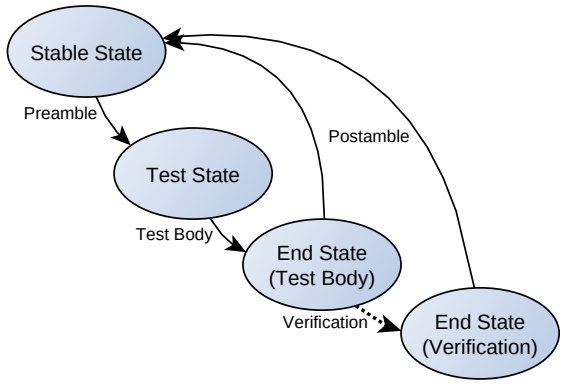


Figure 9: Test case states [68]

3.6 Testing versus debugging

Testing and debugging are sometimes used synonymously. This was particularly true earlier during the debugging-oriented period [32], but even today the distinction is often unclear. It is also common to see the phrase "testing and debugging" used as a single unit. However, first of all testing and debugging have very different objectives: the purpose of testing is to show that a program has bugs while the purpose of debugging is to find the error and correct it. This leads to the logical conclusion that debugging often follows testing [7].

Additionally, differences go beyond the differences in goals and methods, particularly in the psychology. Several key differences are discussed in [7], with the most relevant ones in the scope of this thesis being:

1. Testing starts with known conditions and predictable outcomes. The only unpredictable part is whether the test passes. On the other hand, debugging often starts from unknown initial conditions and the end cannot be predicted.
2. Testing can and should be planned, designed and scheduled. The procedures and duration of debugging cannot be easily estimated or planned.
3. Testing should strive to be predictable, dull, constrained, rigid and inhuman. Debugging demands intuitive leaps, conjectures, experimentation, and freedom.

Testing is just a way to detect the error in a more controlled environment. Otherwise the error would have most likely been discovered in the real world where it might be exponentially more difficult to debug. Therefore, even though we are visualizing the execution traces produced by testing, the goal is to support the debugging effort. After all, testing should be highly predictable and dull; the only question is whether the test passes or not. Presenting the verdict of the test is trivial and does not need sophisticated presentation techniques – the real challenge is to show what led to the particular verdict.

After stepping into the field of debugging, it is important to immediately make a clear distinction between *debuggers* and *trace viewers*, such as our log analysis tool *Reprise* (described in Chapter 7). Varying definitions may be found in literature, but for instance [65] states that "source-level debuggers allow the user to control the execution of the program, setting breakpoints and exploring values as needed".

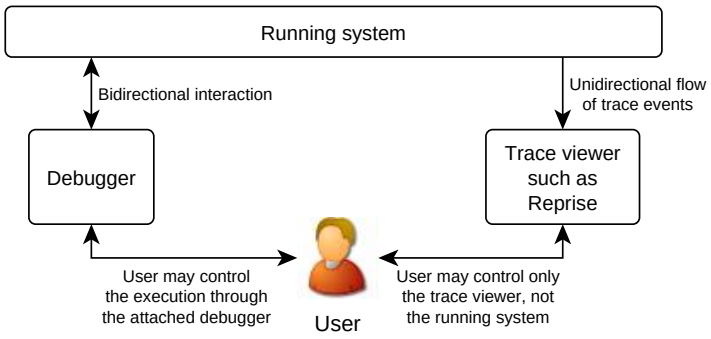


Figure 10: Debugger versus trace viewer

In the scope of this work, the possibility of *controlling the execution* is considered to be the fundamental difference, as illustrated in Figure 10. By attaching a debugger to the running system, user would be able to affect the execution itself. By using an interactive trace viewer, user is only able to affect the representation of the traces. Therefore, although this thesis discusses test log visualization as technique to support the debugging effort, the debugging effort is never considered to affect the test execution. This view supports the idea of the actual testing being carried out by a third-party testing laboratory as presented in the OSI conformance testing methodology. In that kind of scenario, the testing laboratory would execute the tests and deliver all the traces to be analysed by their client.

4 VISUALIZATION

4.1 Motivation

Before proceeding further into the visualization ideas and techniques, it is important to consider the motivation for doing so. In fact, Fekete et al. point out that challenges do exist in both recognizing and communicating the value of the field of information visualization even on a broader scale [27]. This could be explained by considering the probably most widely accepted definition of visualization [13]:

> *"The use of computer-supported, interactive visual representations of data to amplify cognition."*

The last three words of this definition describe the ultimate purpose of visualization: to amplify cognition. Now, cognition is not easily and precisely measurable, and this may cause confusion in assessing the value and motivation of visualization [27]. The discussion about background and motivation of this work in Section 1.1 suggested using visualization techniques (as opposed to text) to support the user during log analysis. "Why visualization?", one could ask.

Larkin and Simon discuss the comparison of representations and selection of the "better" representation as follows. First, two representations of the same data are said to be informationally equivalent if all of the information in the one is also inferable from the other and vice versa. Furthermore, when two representations are informationally equivalent, their computational efficiency depends on the information processing operators that act on them [52]. When considering humans as information processors, knowledge of the human brain and eyes may be exploited to develop representations which are well suited for the human perception system. Understanding of the rules which guide the perception of patterns and motion may be utilized to understand why certain visualizations work and others do not [17]. In other words, understanding of how human visual cognition works makes it possible to represent the information in a format that humans may process more efficiently – and this gives us the ultimate motivation for developing visualization techniques.

4.2 Interactive visualization process

An important but sometimes overlooked part in development of visualization techniques is surprisingly the user. Simply coming up with a graphic technique that is able to display all the data is not necessarily the technique that is most adequate from the viewpoint of the user. Users are an integral part of the visualization process, and this is especially true in an interactive visualization system, as illustrated in Figure 11.

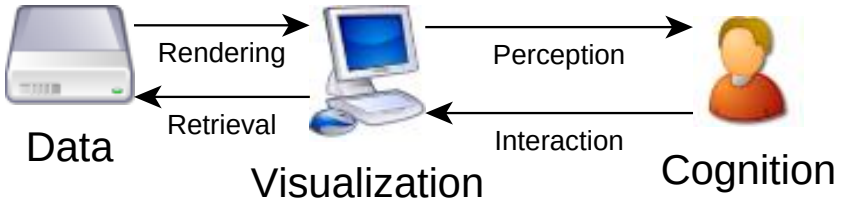


Figure 11: The visualization process [79]

As can be seen in the figure, visualization is not a one-way process of generating pictures. The visualization system is responsible for retrieving and rendering a representation of the data while being controlled by user interactions. In other words, two main components may be identified at the above visualization system: *representation* and *interaction*. The roots of representation component lie in the field of computer graphics and it is concerned with the mapping from data to representation and rendering the representation on the display. On the other hand, the interaction component handles the dialog between the user and the system and its theory comes from the field of human-computer interaction (HCI). It has been argued that visualization research has been primarily focused on the representation while ignoring the needs of the end user and interactivity [87, 79].

4.3 Importance of the domain

In addition to recognizing the importance of the interaction with the user, it is also important to note that in this context the user is not stereotypical "dumb" computer user who just "doesn't get it", with "it" referring to the amazing representation technique developed by the visualization researcher. Instead, both the visualization researcher and the user are experts in their respective fields. Still, this leaves two gaps that must be bridged in order to develop an effective visualization tool: a knowledge gap and an interest gap. The domain expert (user) often has a deep knowledge of his field and he needs

an useful tool. On the other hand, the visualization researcher knows about graphics and human-computer interaction, but does not necessarily understand the problem domain. This forms the knowledge gap. Additionally, the visualization researcher is naturally interested in developing novel methods and techniques that are interesting in the field of visualization, but may not be what the domain expert needs, which is referred to as the interest gap. These gaps are illustrated in Figure 12 [82].

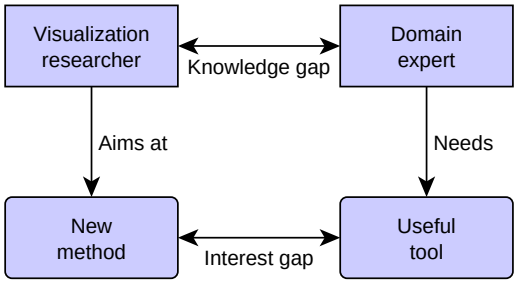


Figure 12: Gaps between the visualization researcher and the domain expert

Wijk suggests using user-centered design approach as a way to bridge these gaps. The end users must be given extensive attention during each stage of the design process. Prototyping and constant feedback is important, as users are much better at stating what they do and what they do not like, than describing what they do exactly want. Regardless, bridging the gaps is never easy. Much effort may be needed to bridge the knowledge gap, without a guarantee that the interest gap is ever bridged [82].

In the context of visualizing test execution logs, it might be safe to assume that the knowledge gap is relatively narrow. Fully understanding the inner workings of complex communication systems certainly require extensive application-specific knowledge, but the fundamental concepts of message exchange are most likely well recognized by all software engineers. Therefore, developing a practical and useful log visualization tool should be a reasonable goal as long as effort is put to eliminating the interest gap. Particularly, the user interface and usability aspects may not be neglected. The next section introduces one starting point for designing advanced graphical user interfaces.

4.4 Visual Information Seeking Mantra

Retrieving the desired piece of information becomes increasingly more difficult as the information volume grows. This creates demand for interactive visualization tools and

techniques that allow easy navigation in large datasets. Shneiderman presents his visual design guideline, the *Visual Information Seeking Mantra*, as follows [71]:

> *Overview first, zoom and filter, then details-on-demand*
>
> *Overview first, zoom and filter, then details-on-demand*
>
> ...

In the paper published back in 1996 [71] the mantra is repeated ten times, and each repetition is said to represent one project in which he found himself rediscovering this principle. It might be safe to assume that following this trend, in a paper published today the mantra would be repeated over several pages. This mantra refers to the fact that the user expects to be first presented an overview of the data, followed by the possibility to navigate in the data by zooming and filtering. Fine details are only displayed on demand. The seven fundamental tasks that users may want to perform on the data are summarized by the paper as follows:

1. *Overview:* Gain an overview of the entire collection.
2. *Zoom:* Zoom in on items of interest.
3. *Filter:* Filter out uninteresting items.
4. *Details-on-demand:* Select an item or group and get details when needed.
5. *Relate:* View relationship among items.
6. *History:* Keep a history of action to support undo, replay and progressive refinement.
7. *Extract:* Allow extraction of sub-collections and the query parameters.

Section 7.5 describes how support for these tasks is implemented in the visualization tool that is being developed as a part of this thesis work.

5 TTCN-3

5.1 History

The roots of *Testing and Test Control Notation version 3* (TTCN-3) test specification language can be traced back to the OSI Conformance Testing Methodology and Framework (CTMF). As suggested by the name, the language is currently in its third edition. This section provides brief history of the language.

Work on *Tree and Tabular Combined Notation* (TTCN) began in 1983 as a part of the CTMF work, and the first edition of TTCN was standardized almost ten years later in 1992. TTCN has been widely used for describing protocol conformance test suites, primarily in standardization organizations. The main characteristics of TTCN that contributed to its wide acceptance are stated to be [47]:

- It provides easy and natural means to describe all possible scenarios of stimulus and the resulting reactions between the tester and the System Under Test (SUT).
- The verdict system provides means for judging whether the test result agrees with the test purpose.
- The language contains appropriate mechanism for describing constraints on received messages so that conformance to the specification can be mechanically evaluated.

The first edition of TTCN lacked one particular feature that was soon found to be important: concurrent behaviour within the tester, the SUT, or between them. In addition to extending the language with a concurrency mechanism, other features such as the concept of module and package were introduced to better support reusability and encapsulation. This extended TTCN was adopted as TTCN edition 2 (TTCN-2) in 1998. After that, some defects have been corrected and a revised version, sometimes referred to as TTCN-2++, was published in 2002 [47].

Despite the improvements and having been successfully adopted for applications beside the OSI protocol conformance testing, TTCN and TTCN-2 were still considered to suffer from the strong ties to OSI. To be truly suitable for multiple application areas, TTCN-2

is too restrictive and it includes concepts and application-specific semantic rules which have a meaning only in OSI conformance testing. In 1998, along with the decision to correct the defects in TTCN-2, work on a redesigned third edition was started. The primary goal of the redesign was to modernize TTCN and to widen its application area [34].

The most visible difference between TTCN-2 and TTCN-3 is that while TTCN-2 used a tabular notation as the main description notation, the TTCN-3 Core Language looks and feels like many common programming languages such as C++ or Java. To reflect the new nature of TTCN, also the acronym of TTCN was changed to refer to *Testing and Test Control Notation* instead of the previous Tree and Tabular Combined Notation [47].

The TTCN-3 standard suite consists of nine parts and various extensions. The language has been under active development ever since the work started and as of today, February 7th 2010, the most recent officially published version is 4.1.1 (TTCN-3: 2009). However, it is worth mentioning that the Graphical Presentation Format (part 3), which is is extensively utilized in this thesis, has not been updated since version 3.2.1 (TTCN-3: 2007) [26].

5.2 Overview

TTCN-3 is applicable to specification of all types of reactive system tests over different communication interfaces. Primary presentation format is the TTCN-3 Core Language that provides interfaces to different data description languages. Additionally, the TTCN-3 standard suite defines alternative presentation formats: The tabular presentation format (TFT) [21] is similar in appearance and functionality to TTCN-2 while the graphical presentation format (GFT) is based on a subset of Message Sequence Charts (MSC) with test specific extensions. This overview is illustrated in Figure 13.

The figure also highlights the focus area of this thesis: utilizing TTCN-3 GFT and other visualization means for execution log presentation. Idea of the presentation formats is that they can be converted into the TTCN-3 Core Language notation while preserving their meaning. However, in the context of this thesis, GFT is only considered as a format for log presentation; we are not concerned about using it for test specification and thus converting it to TTCN-3 core language presentation.

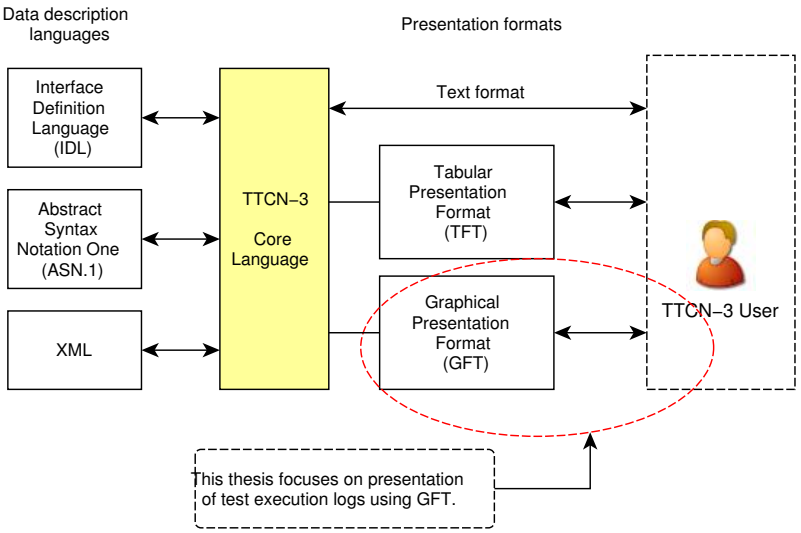


Figure 13: TTCN-3 Core Language and its relation to data description languages and the presentation formats

5.3 Test System

Like code written in any programming language, TTCN-3 code is not executable on its own. The code needs to be either interpreted or compiled into executable format of some kind. A collection of tests written in TTCN-3 is called a *test suite*, or *abstract test suite*. In this case abstract refers to the fact that the test suite lacks any implementation-specific information. For example, when messages are sent in TTCN-3 code, the code does not state how the messages are sent in the real world. As we recall from Section 2.1, the message exchange between the tester and the SUT will eventually happen by transmitting bits over some transmission media. This abstraction is useful as it separates the test case behaviour and system-specific details, but in order to end up with real-world executable tests, we need to combine the abstract TTCN-3 test suite with appropriate tools [83].

In the OSI conformance testing methodology the realization of executable tests is called *means of testing* (MOT), defined as "the combination of equipment and procedures that can perform the derivation, selection, parameterization and execution of test cases, in conformance with a reference standardized ATS and can produce a conformance log" [43]. In the TTCN-3 terminology, the complete system that provides means for execution of the test suite is referred to as a *TTCN-3 Test System*.

A test system can be thought of conceptually as a set of interacting entities where each entity corresponds to a particular aspect of functionality in a test system implementation [23]. Figure 14 shows the structure and the entities in a TTCN-3 test system as well as the area where the work of this thesis is located.

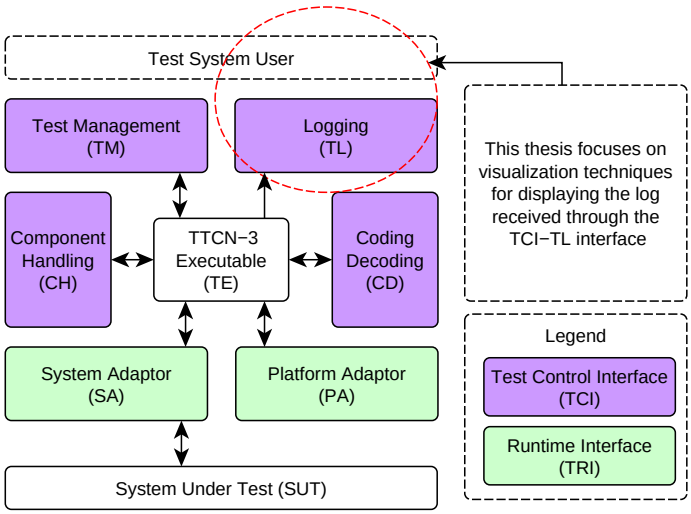


Figure 14: Structure of a TTCN-3 Test System

Here we encounter yet another abstraction: the concrete means for interconnecting the test system entities are specific to the implementation of the TTCN-3 tool in question. The standard suite [26] defines the interfaces as a set of operations independent of target language. Official language mappings are available for Java, ANSI C and C++ , but also proprietary support for other programming languages or environments, such as the Microsoft .NET Framework have been developed [2]. In practice, the TTCN-3 tool vendor makes available a TTCN-3 compiler or interpreter and a software development kit (SDK), which is used to implement the application-specific functionality, such as the system adaptor and the encoder/decoder. The vendor-provided entities are then either linked or otherwise combined with the user-implemented entities and together they form a complete, fully functional test system.

TTCN-3 Executable (TE) is the heart of the test system. It deals with the interpretation and execution of TTCN-3 code. In a concrete test system implementation it corresponds either to the executable code produced by a TTCN-3 compiler or a TTCN-3 interpreter. TE may be executed in a centralized or distributed manner. TE is connected to rest of the system through two major interfaces: TTCN-3 Runtime Interface (TRI) and TTCN-3 Control Interface (TCI), defined in the respective standards [23, 24]. TRI provides adaptation for timing and communication of a test system to a particular processing platform

and the system under test. TCI provides adaptation for management, test component handling and encoding/decoding. Brief description of each of the entities as defined in the respective standards is:

- System Adaptor (SA): Adapts message- and procedure-based communication of the TTCN-3 test system with the SUT to the particular execution platform. It is responsible for propagating sent messages from the TE to the SUT and for notifying the TE of any messages received from the SUT.
- Platform Adaptor (PA): Implements TTCN-3 external functions and provides a TTCN-3 test system with a single notion of time.
- Test Management (TM): Responsible for the overall management of a test system. Execution starts with the TM, which is also responsible for propagating module parameters if necessary. Typically also the user interface.
- Coding and Decoding (CD): Responsible for the external encoding and decoding of TTCN-3 values into sequences of bits suitable to be sent to the System Under Test.
- Component Handling (CH): Implements communication between test system entities that are distributed into several nodes.
- Test Logging (TL): The TL entity performs test event logging and presentation to the test system user. Includes information such as test component creation, startup and termination, data sent to/from the SUT, timers and so on.

5.4 Core Language

The TTCN-3 Core Language is the primary notation for TTCN-3 test specifications and it is stated to serve three purposes [20]. First, it is a generalized text-based test language in its own right. It also acts as a standardized interchange format of TTCN-3 test suites between TTCN-3 tools. Finally, it serves as the semantic basis for various presentation formats. Among the presentation formats, its unique characteristic is that the core language may be used independently of the other presentation formats while the currently specified tabular and graphical formats may be only used on the basis of the core language.

In addition to the basic programming language constructs such as data types, variables, expressions, assignments, loops and functions, the language includes various features that have been specifically designed for testing. These features separate the TTCN-3 core language from the general purpose programming languages. The statements defined for the language may be categorized as follows [33]:

- Basic program statements. The common denominator for programming languages.
- Configuration operations. These are concerned with setting up and controlling test components and connections.
- Communication operations. Used for communication between and amongst test components and the system under test.
- Behaviour statements and operations. Used to define alternative, interleaved and default test behaviour.

Listing 1 shows the typical, albeit simplified, structure of a TTCN-3 test suite. However, the example is still far more complex than the most trivial "hello world" program would be. The main building blocks of a test case (preamble, test body and postamble) are separated with comments. In the scope of this thesis we are primarily interested in the communication that happens inside the test body.

Listing 1: Hello test suite

```
module Hello {

  type port HelloPortType message { inout charstring; };
  type component HelloComponent { port HelloPortType helloPort; };
  type component HelloSystemInterface { port HelloPortType helloPort; };

  template charstring a_Hello := "Hello world!";

  testcase TC_hello() runs on HelloComponent system HelloSystemInterface {

    // Preamble
    map(mtc:helloPort, system:helloPort);

    // Test body; The primary interest area of this thesis.
    helloPort.send(a_Hello);
    helloPort.receive(a_Hello);
    setverdict(pass);

    // Postamble
    unmap(mtc:helloPort, system:helloPort);
```

```
21   };
22
23   control {
24     execute(TC_hello());
25   }
26 }
```

The above example assumes that the system under test is an echo server that replies to our hello greeting. The execution in fact starts from the *control part*, which is shown on lines 23-25. The control part is the entry point for executing a TTCN-3 test suite and an analogy in the C language would be the `main()` function. The control part typically controls and sequences the execution of test cases.

More relevant in the context of this thesis are the *template* definition on line 7 and the test body on lines 15-17. The template mechanism provides a comfortable way to specify data to be transmitted or received. In this example the template named a_Hello is defined as a charstring with the content of "Hello world!". The send statement on line 15 is straightforward: it sends a message defined by the template to the SUT via helloPort without any conditions.

However, the receive statement is somewhat more complex. The receive statement is executed and the control flow moves past it only after a message that *matches* the given template has been inserted to the input queue of helloPort. In other words, in this concrete case the line 16 is executed after the SUT adapter has enqueued a message with the exact content of charstring "Hello world!". Finally after receiving the message, a pass verdict is assigned to indicate successful test result.

Templates may further be parameterized to make them better adaptable to different testing situations, especially for specifying received data. However these more complex cases are not discussed in detail here because visualizations typically would contain only the name of the template that matched (or did not match), with the data representation being done by utilizing the core language notation. Combined with a SUT adapter that simulates an echo server and executed using OpenTTCN Tester version 3.0.3, the previously shown hello test case results in the textual log shown in Listing 2.

Listing 2: Hello example textual log

```
1 mtc : {01:45:45.338} : // CASE TC_hello STARTED
2 mtc : {01:45:45.374} : map(mtc:helloPort, system:helloPort);
3 mtc : {01:45:45.444} : helloPort.send(charstring a_Hello := "Hello world!");
4 mtc : {01:45:45.444} : helloPort.receive(charstring a_Hello := "Hello world!");
5 mtc : {01:45:45.444} : setverdict(pass);
6 mtc : {01:45:45.481} : unmap(mtc:helloPort, system:helloPort);
```

```
7 mtc : {01:45:45.482} : // CASE TC_hello FINISHED
8 mtc : {01:45:45.482} : // VERDICT TC_hello PASS
```

Obviously the above test case is a simplified example which in fact contains a serious design flaw: the test will deadlock if the system under test does not respond with the expected response (the receive operation never matches). In reality we would like to handle this scenario by assigning a fail verdict, and this is where the concept of alternative behaviour discussed in the next section becomes useful.

5.5 Alternative behaviour

A robust test system is able to handle any unexpected response from the SUT. Instead of deadlocking like the above presented hello example, a proper test suite grafecully handles this kind of scenarios by defining alternative behaviour in addition to the expected behaviour. In general, the two most common alternative scenarios are either receiving an unexpected message or not receiving anything at all. One way to further categorize the unexpected messages is to divide them to three different kinds: something completely unrelated, something that was recognized but is known to be an incorrect response or something that causes the test system to re-enter the wait state.

To allow effectively specifying alternative test behaviour, the unique characteristics of the TTCN-3 language are the *alt statement* and *altsteps*. The alt statement denotes branching of test by defining a set of possible events that are to be matched against a particular snapshot. Altsteps take this concept even further and specify the alternatives in a function-like reusable description. Altsteps are always either explicitly or implicitly inside an alt statement, and because of this their primary benefit is to prevent code duplication by allowing common functionality to be grouped together.

To avoid race conditions, the alternative evaluation has *snapshot* semantics. A snapshot is taken when the execution of alt statement begins and it effectively freezes the system state for the duration of the execution. If none of the alternatives can be chosen, a new snapshot is taken and the evaluation is done again[1]. Figure 15 illustrates the control flow of alt statement and altstep inside it.

Examples of specifying alternatives in TTCN-3 code will be given in the subsequent

[1]This is a simplification. Obviously, a proper TTCN-3 tool implementation will take the next snapshot only after something that might modify the snapshot has been detected.

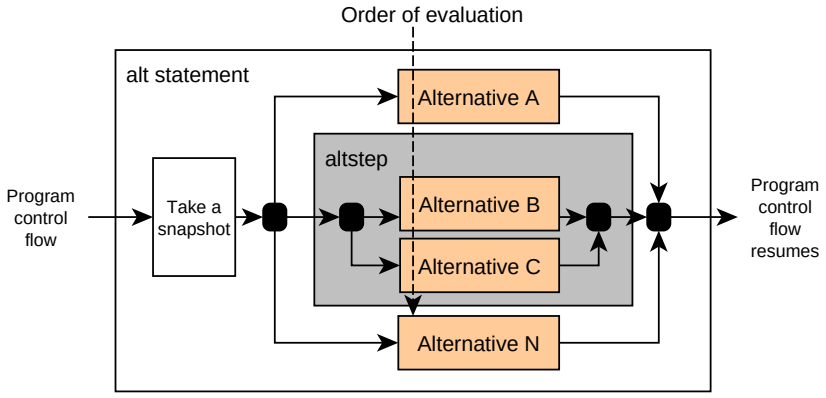


Figure 15: Control flow in alternative behaviour

sections. For example, Section 5.9.1 presents a real-world example from the Session Initiation Protocol (SIP) test suite, and Section 6.7 discusses visualization of log generated by execution of an altstep.

5.6 Test Components, Ports and Test System Interface

Test components, conceptually part of the TTCN-3 executable, are the entities responsible for executing the TTCN-3 code. In programming terms, each test component is typically implemented either as a separate thread or process. A test case defines the behaviour of the main test component (MTC), which is started implicitly by the test system. The main test component may also create and start additional parallel test components (PTC) in order to execute code in a concurrent fashion. Furthermore, PTCs are either of alive or non-alive type. PTCs of non-alive type start their execution only once and die either after executing the last statement or by being stopped explicitly. On the other hand, PTCs of alive type may be started, stopped and restarted later on.

Test components communicate with each other and the system under test via *ports*. A port can be thought of as a first-in-first-out (FIFO) queue in the receive (in) direction. Incoming messages or calls are kept in the queue until being processed by the test component that owns the port. In the send (out) direction, a port is linked directly to the communication partner.

The Test System Interface (TSI) is a special type of abstract component that connects the test system with the system under test. Intercomponent communication happens through

connected ports while the test system interface ports may be *mapped* to test component ports. Furthermore, test system interface ports are implemented by the system adapter (SA). All communication with the SUT goes through the system adapter and test system interface. Figure 16 shows a conceptual view of the entities involved in connecting the TTCN-3 executable with the SUT, including the OSI layered protocol architecture.

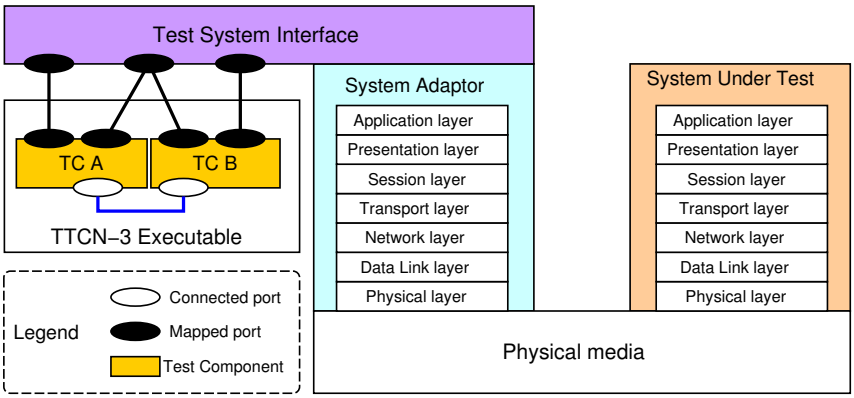


Figure 16: TTCN-3 Executable connected to the System Under Test

5.7 Test Logging Interface

Out of the entities of a TTCN-3 test system, the Test Logging (TL) entity is the most relevant in the scope of this thesis. Also the Reprise visualization tool, which is being developed as a part of this thesis work, includes a TL implementation. A TL entity is connected to the TTCN-3 executable via the TTCN-3 Control Interface (TCI) and more specifically the Test Logging subinterface (TCI-TL) [24].

In the present version of the TTCN-3 standard (4.1.1) the TCI-TL interface contains 105 member operations. The interface is defined using the abstract CORBA Interface Definition Language (IDL) [57]. Additionally, language mappings to ANSI C, Java and C++ programming languages are given. To implement a TL entity in practice, the test system developer is responsible for providing implementation (in the chosen programming language) for an interface which contains the 105 distinct operations. Additionally, the implementation must be registered to the test system by vendor-specific means.

The TE will then provide information about test execution by invoking the relevant operations on the registered client entities. Each operation indicates that a specific event happened – for example, a message was received, a timer expired or the test terminated.

One of the most common events, the unicast send event, is shown as an example. Listing 3 shows the IDL definition with the parameter descriptions from [24] included as comments.

Listing 3: IDL definition of tliMSend_m

```
1  void tliMSend_m(
2   in TString am,              // An additional message.
3   in TInteger ts,             // The time when the event is produced.
4   in TString src,             // The source file of the test specification.
5   in TInteger line,           // The line number where the request is performed.
6   in TriComponentIdType c,    // The component which produces this event.
7   in TriPortIdType at,        // The port via which the message is sent.
8   in TriPortIdType to,        // The port to which the message is sent.
9   in Value msgValue,          // The value to be encoded and sent.
10  in Value addrValue,         // The address value of the destination within the SUT.
11  in TciStatusType encoderFailure, // The failure message which might occur at encoding.
12  in TriMessageType msg,      // The encoded message.
13  in TriAddressType address, // The address of the destination within the SUT.
14  in TriStatusType transmissionFailure)  // The failure message which might occur at
    →transmission.
```

The parameters shown on lines 2-6 are common to all of the events, while the rest are specific to the event in question. The TCI-TL interface and the rest of the events will be further discussed in Chapter 6 along with their mapping to a graphical presentation.

At the implementation level, the TL entity receives information about test execution as a sequence of operation invocations. The subsequent sections in this thesis discuss the TL interface and the operations extensively, but constantly referring to "invocation of operation X" would be cumbersome and awkward. Essentially, each invocation is a notification about an event [2] that happened during the test execution. Consequently, in this document the substantive *log event* has been selected to describe the notifications received through the test logging interface.

5.8 Graphical Presentation Format

Message Sequence Chart is a *"trace language for the specification and description of the communication behaviour of system components and their environment by means of message interchange"* [48]. The TTCN-3 Graphical Presentation Format (GFT) [3] extends MSCs by defining test-specific extensions [22]. GFT provides means for graphical

[2] An event is "a thing that happens or takes place" [59].

[3] The abbreviation does not match the name, for a reason unknown to me. As an interesting detail, [5] shows that the TTCN-3 Graphical Presentation Format was abbreviated as GPF during its development.

presentation of TTCN-3 behaviour definitions. As illustrated in Figure 17, the control part, test cases, functions and altsteps can be mapped onto a corresponding GFT diagram and vice versa, but no graphical presentation is defined for types and data.

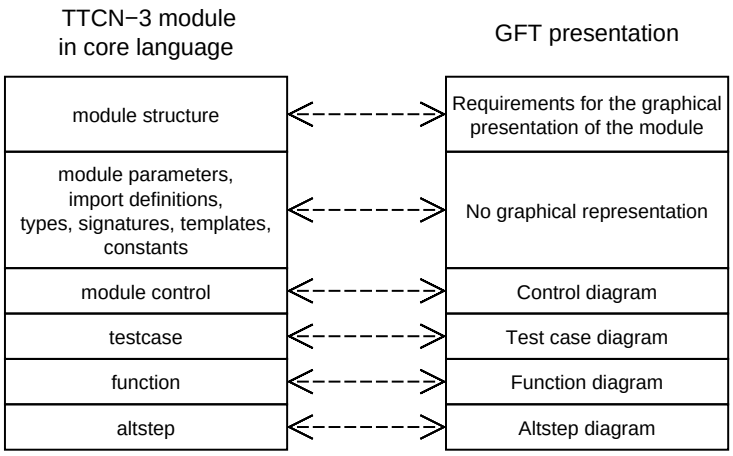
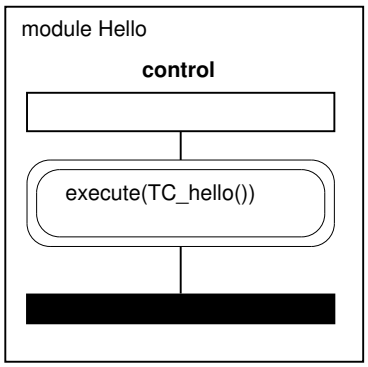


Figure 17: Mapping between TTCN-3 core language and GFT [22]

Being able to review the recorded communication is certainly important in the log analysis, and therefore the lack of data presentation is a concern that must be addressed. The approach taken in this thesis is to keep the complete textual log available in addition to the graphical presentation. Another possibility would be to utilize for example the TTCN-3 Data Presentation Format (DPF) [31], which is an inofficial graphical notation for representing TTCN-3 types and values graphically. The objective of DPF is to fill the current gaps and, together with GFT, form the ultimate graphical representation of TTCN-3.

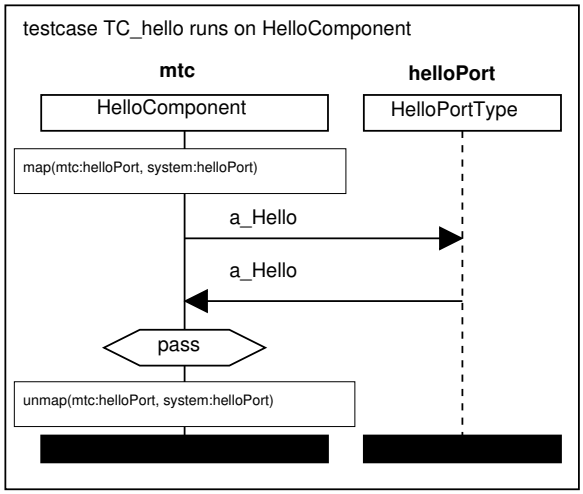
The four diagram types specified in GFT are *control diagram*, *test case diagram*, *function diagram* and *altstep diagram*. Introduction of the diagrams is carried out here by converting the "hello" test suite from Section 5.4 into its graphical presentation. The control part, which is the entry point to the TTCN-3 program, is described using the control diagram shown in Figure 18.

In this particular case the mapping is very straightforward, as can be seen in the figure. The execute statement is used to start test cases from the control part and it is mapped to the execute symbol, which is then attached to instance of the control component. In GFT the execute symbol invokes the test case diagram, which is shown along with the corresponding source code in Figure 19.




```
// TTCN-3 Code, control part

control {

  execute(TC_hello());

}
```

Figure 18: GFT representation of the Hello control part




```
map(mtc:helloPort, system:helloPort);

helloPort.send(a_Hello);

helloPort.receive(a_Hello);

setverdict(pass);

unmap(mtc:helloPort, system:helloPort);
```

Figure 19: GFT representation of the Hello test case

The symbols shown in the diagram have not been introduced yet; instead, the purpose of this diagram is to merely give an example of a GFT test case diagram. More detailed discussion about the various symbols is given throughout Chapter 6, which discusses applicability of the symbols for log representation.

As the examples show, GFT diagrams are constructed by attaching various symbols to the test component and port instances. Each diagram includes a single test component instance and the port instances for each relevant port of the test component. Test case diagram shows the execution of the main test component (MTC), while function diagrams may either represent a function invoked within the MTC or a function passed to a parallel test component. Message exchange and procedure calls are shown between the test component and the port or ports.

Test case diagrams may also invoke *altstep diagrams* and *function diagrams* by using

the reference symbol. This technique originates from the the MSC specification [48], where the High-level MSC (HMSC) is defined as a graphical method for combining a set of MSCs. HMSC is a directed graph which may contain nodes that reference another MSC or HMSC. Figure 20 shows the diagram invocation methods specified in the GFT standard [22].

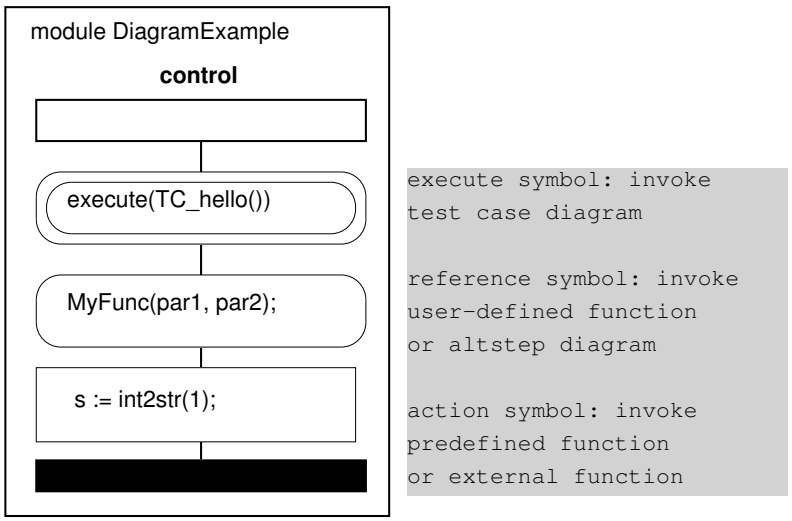


Figure 20: GFT diagram invocations

An additional rule defined in GFT is that if a function is called inside a TTCN-3 construct with an associated GFT symbol, the function invocation is presented as text within the associated symbol. Also, the reference symbol may only be used for presenting invocations of user-defined functions within the current module. External functions or predefined TTCN-3 functions shall be represented using the textual form inside an action symbol. Test case diagram is the primary diagram adopted for log visualization and various examples will be shown throughout the remaining sections of this thesis. Visualization of control part log using the control diagram is briefly discussed in Section 6.8.1 and visualization of an altstep log using the altstep diagram is shown in Section 6.7. As these examples will illustrate, these diagrams have practically no differences for the purposes of log presentation. In each case, the visualization is constructed from a test component instance and the relevant port instances with various symbols attached to them. Therefore, no examples of the function diagram are shown in this thesis, as it is merely a special case of the test case diagram.

5.9 Communication

We are interested in testing of *communication systems*. Therefore, although TTCN-3 contains a number of general purpose language constructs such as loops and functions, we focus on the communication operations. This section introduces the two fundamental communication paradigms defined in TTCN-3: *message-based* and *procedure-based* communication.

5.9.1 Message-based communication

Arguably the most common communication pattern in TTCN-3 test suites is first sending a stimulus to the SUT and then assigning the verdict based on the response (or absence of a response). This real-world example is used to illustrate the key elements of a typical communication sequence. The Listing 4 shows a slightly simplified source code of the test case `SIP_QC_TE_V_002`, taken from the SIP protocol conformance test suite [25].

Listing 4: SIP_QC_TE_V_002

```
1  altstep defaultCCTE() runs on SipComponent {
2    [] any timer.timeout { setverdict (fail); }
3    [] SIPP.receive { setverdict (fail); }
4  }
5
6  testcase SIP_QC_TE_V_002(inout CSeq loc_CSeq_s)
7  runs on SipComponent system SipInterfaces
8  {
9    // Parts of the preamble have been omitted
10   v_Default := activate (defaultCCTE());
11   SIPP.send (OPTIONS_Request_s_1(...)) to v_sent_label;
12
13   TResp.start (PX_TRESP);
14   alt {
15     [] SIPP.receive (Response_200_r_5(...)) sender v_sent_label { setverdict (pass); }
16     [] SIPP.receive (Response_1XX_r_1(...)) { repeat; }
17     [] SIPP.receive (Response_Any_r_1) sender v_sent_label { setverdict (fail); }
18     [] TResp.timeout { setverdict (fail); }
19   }
20   // The postamble has been omitted
21 }
```

The test purpose is defined as “Ensure that the IUT on receipt of an OPTIONS request, sends a Success (200 OK) including the headers From, Call-ID, CSeq and Via headers copy from the OPTIONS request.”. The source code shows the concept of alternative test behaviour discussed in Section 5.5. In the listing, lines 1-4 show the function-like

definition of an altstep, and line 10 activates the altstep as a *default altstep*.

During runtime, the default altsteps are effectively appended to the list of alternatives inside each alt statement. A common application of the default mechanism is to make the system more robust by providing a generic handler for any unexpected behaviour, such as receipt of an unexpected message. In this particular case, receiving a response that matches the `Response_200_r_5` template leads to pass verdict. If the received response matches the `Response_1XX_r_1` template in the second alternative, the entire alt block is repeated. Any other response, including a timeout, causes a fail verdict to be assigned. A GFT representation of this test case is shown in Figure 21.

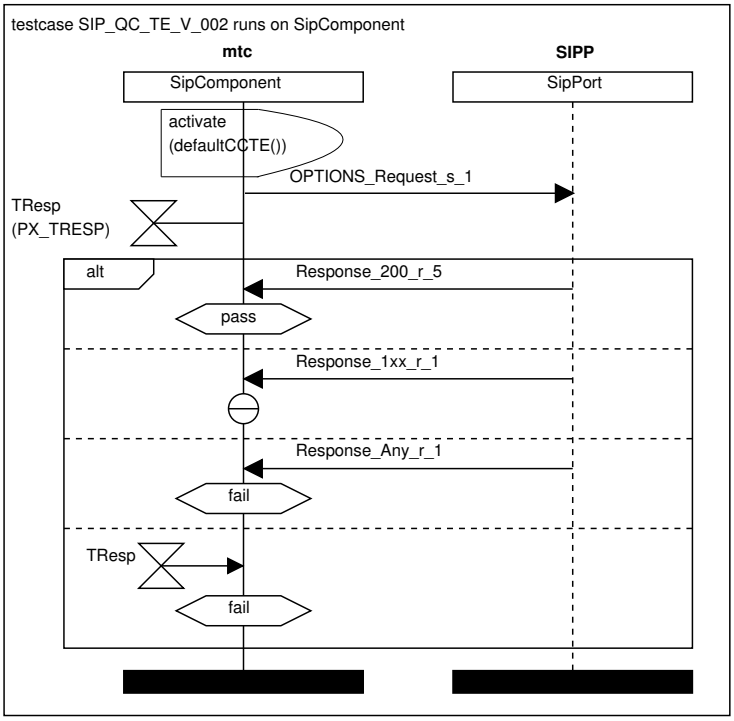


Figure 21: GFT representation of SIP_QC_TE_V_002

5.9.2 Procedure-based communication

Compared to message-based communication, the procedure-based communication model has one fundamental difference: explicit roles of the communication partners. Even in a client/server scenario, message-based communication uses the same primitives regardless of the role of the communication partner. Distinction between clients and servers is defined only by the semantics of the communication protocol in question.

On the other hand, procedure-based communication clearly and explicitly defines the roles of the communication partners. Client always invokes a remote procedure on a server which either returns a reply or, in error scenarios, raises an exception. In TTCN-3 terms, the client performs a *call* and the server responds by either *reply* or *raise*.

Additionally, while message-based communication is based on asynchronous message exchange so that the sender continues its execution immediately after the send operation, procedure-based communication is typically synchronous. The caller blocks until the callee has processed the message and either returned a reply or raised an exception.

5.9.3 Role of the test system

The communication paradigms were introduced above in an abstract manner, ignoring the implementation details. Particularly the TTCN-3 template system excels at providing an abstraction that separates the test suite from the reality: named templates completely hide the complexity of the messages. For example, looking at the source code in the previously shown Listing 4, one can notice that the test case sends and receives only templates, each with a descriptive name. However, as we recall from Section 2.1, the messages will be eventually transmitted as bits and bytes, and the test system must perform the necessary conversion between the transfer syntax and the abstract TTCN-3 value. This section describes the communication on a more concrete level.

The conversion from TTCN-3 value to binary data suitable for transmission over some physical media is referred to as *encoding*, and the reverse operation is called *decoding*. In a TTCN-3 test system the encoding and decoding is delegated to the coding/decoding (CD) entity, and the system adaptor (SA) is responsible for communicating with the system under test (SUT).

Figure 22, produced using the LaTeX MSC package [53], gives a more concrete view of the actions carried out during a common communication pattern: sending a request and receiving a response to it. In TTCN-3 code, this is represented by `send` operation followed by a `receive` operation.

Once execution of the send operation begins, the TE first requests encoding of the abstract value by invoking the `tciEncode` operation at the CD entity, followed by logging of the encode request by invoking `tliEncode` at the TL. Next, the TE requests transmission of the message to the SUT by invoking the `triSend` operation at the SA.

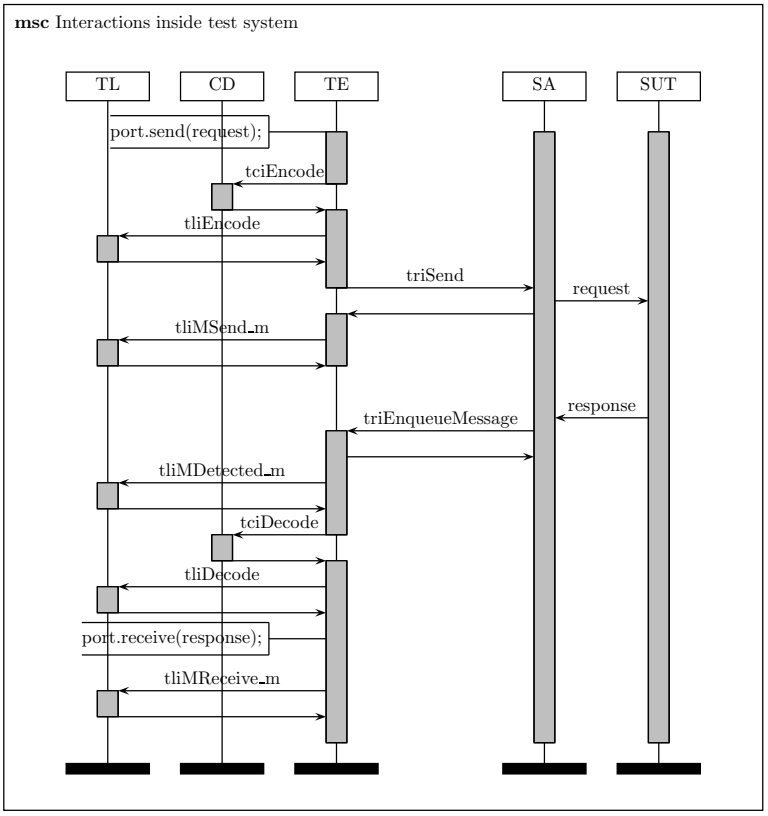


Figure 22: Interactions inside test system during a send/receive pair

Once the send completes, the TE notifies the test logging entity (TL) by invoking the `tliMSend_m` operation.

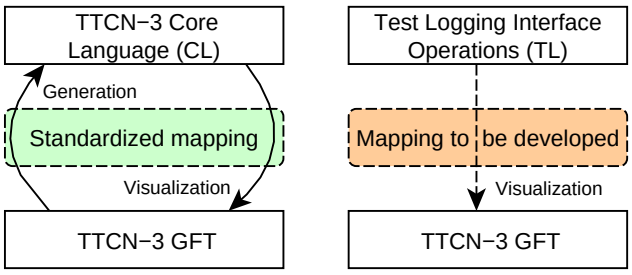
Eventually the SUT processes the request and sends back a response, which is received by the SA. SA relays the message back to the TE by appending it to the incoming queue of the relevant port. This happens by invoking the `triEnqueueMsg` operation on the TE. Before the message may be handled at the TE, the message has to be decoded. The decoding is performed at the CD entity by invoking the `tciDecode` operation, followed by a request to log the decode operation. Finally, the TE determines that the incoming message indeed matched the expected response and executes the `receive` operation, followed by logging of the receive event by invoking the `tliMReceive_m` operation at the TL.

As these interactions illustrate, CD and SA are (at least partially) application-specific due to their responsibility of adapting the test system to a particular system. Conversely, due to abstractions, the TE is always generic in nature. The TL is located somewhere between these two extremes: a generic implementation is often sufficient, but application-specific features may be sometimes desired.

6 ADOPTING TTCN-3 GFT FOR LOG VISUALIZATION

6.1 The objective

The previous chapter gave examples of the standardized mapping from TTCN-3 core language (CL) to the graphical presentation format (GFT). However, as illustrated in Figure 23, our task of using GFT for presenting information received through the test logging interface (TL) is somewhat different.



(a) CL to GFT **(b)** TL to GFT

Figure 23: Mapping of CL to GFT compared to mapping of TL to GFT

While the core language and GFT have been designed so that combined use of generation and visualization allows round-trip engineering, we are only interested in unidirectional case of visualizing TL using GFT. Unfortunately the TCI standard [24] does not define a mapping from the log events to GFT. Therefore, we have to establish the rules for representing the events graphically either by utilizing some related work or by developing the mapping on our own.

This chapter forms the body of the theoretical results of this thesis and is structured as follows. To begin with, related work is discussed in a brief literature review. After that, the solution approach is described first, followed by presentation of my work. Finally, some examples are given in the end of the chapter.

6.2 Related work

GFT is stated to be well suited for visualization of TTCN-3 execution traces on different levels of abstractions ranging from showing only the messages exchanged between the test components and the SUT all the way to showing default activation, timers, and functions [5]. However, the paper does not contain anything concrete that would assist us in this work.

Din et al. have described their implementation of TTCN-3 log visualization [18]. The presentation format is based on a MSC-like sequence diagram with symbols defined by GFT. The system is implemented in a distributed manner with log presentation separated from the test system by using the CORBA [57] technology. TraceViewer [39] has been utilized in the implementation. Graphics are displayed using the Scalable Vector Graphics [85] (SVG) format in a web browser. The technique is also stated to be applicable outside the domain of TTCN-3. In contrast, this thesis presents an approach that is different in two aspects.

First, although both visualizations are based on the symbols adopted from GFT, their work aims at a more generic presentation while the approach presented in this thesis maintains the look and feel of GFT as closely as possible. Regardless, with respect to *practical* usefulness, the difference is most likely subtle and irrelevant. Second, due to being based on SVG, their implementation produces documents that are viewable anywhere in a web browser or other tool capable of displaying SVG graphics, which is an obvious benefit. A detailed description of our implementation will be given in Chapter 7, but the fundamental difference is that our implementation is tightly integrated in the TTCN-3 development and execution environment. This allows offering the user a single environment throughout the full cycle from test development and execution to log analysis. Additionally, interactions with other parts of the test project, such as linking the log event and the respective source code, may easily be introduced if necessary. However, Din et al. mentioned plans to integrate their tool in Eclipse, which makes the tools more alike in this respect.

6.3 The solution approach

The review of related work did not reveal much besides the fact that GFT is considered to be suitable for log presentation and it has been implemented in practice, at least once.

This leaves us where we started: we have to develop our own rules for mapping the log events to GFT.

The log events have been defined in [24] as a seemingly isolated interface: no relationship to the core language [20] or the graphical presentation format [22] have been specified. In many cases, such as the send event, the relationship between these three is obvious. Execution of the `send` operation results in generation of the `tliMSend` event, and the send operation is represented by an arrow symbol in GFT. However, in all of the cases this kind of relationship does not exist or is not as obvious.

This chapter presents my review and classification of the events. The work has been carried out in two steps as described below.

1. Evaluate each of the events and determine whether the above discussed correspondence between the event, a core language construct and a symbol exists.
2. Review the remaining events and categorize them based on their effect in construction of the visualizations.

In other words, the first step is mechanical work that is carried out to filter out the trivial cases where the visualization may be indirectly derived from the standard. The second step focuses on the events whose role is still unclear in the graphical presentation. Overview and my classification of the events is given in Section 6.4. The identified categories are further discussed in Sections 6.5, 6.6 and 6.7.

6.4 Mapping log events to TTCN-3 core language constructs

This section enumerates all of the events defined in the TL interface along with the corresponding TTCN-3 language construct and GFT symbol, where applicable. The standard does not specify any formal categories for the events but to structure the presentation, the events have been first divided into the following four groups.

1. Control flow events. These events describe the control flow of the TTCN-3 program execution.

2. Configuration events. These events provide details about the dynamic configuration of the test system, including timer information.

3. Communication events. These events contain information about the communication that has happened.

4. Miscellaneous events that do not fit into the above categories.

The events are presented in a number of tables. Unless stated otherwise, the graphical symbol shown for an event is the standardized GFT representation of the respective TTCN-3 core language statement/operation. The reader is advised to keep in mind that the objective is to provide a complete listing of the log events and additionally identify the matching TTCN-3 language construct and GFT symbol. Therefore, this section *does not* aim at providing a detailed, thorough and accurate description of the respective TTCN-3 core language constructs or their GFT representation. This section may mention TTCN-3 concepts that have not been introduced in this document. This is a conscious choice that has been made in order to limit the size and the scope of this document. Whenever in doubt, the TTCN-3 standard suite [26] is the authoritative source for the descriptions.

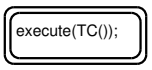
6.4.1 Control flow events

The events listed in Table 1 provide indication about the flow of control in the TTCN-3 program. Many of these events do not have a meaningful presentation as a visual symbol. Instead, the visualization tool may utilize these events for constructing the visual representation.

The events that indicate startup and termination of test case are obviously valuable for the visualization tool. For example, the test case startup indication instructs the tool to instantiate a new diagram. Similarly, any pending actions, such as scaling the diagram to fit the available screen real estate, may be performed upon notification of the test case termination.

Value of the test case verdict is commonly assigned in the selected branch inside an alt statement. Therefore, the control flow indications during evaluation of an alt statement may turn out to be useful during log analysis. Visualization of alternative behaviour and mismatches is further discussed in Section 6.7. The events indicating scope transitions

Table 1: Diagram control events

[1]	Event (tli...)	CL statement	Symbol	Comments
A	`TcExecute`	`execute`		Request to execute a test case from the control part.
B	`TcStart` `CtrlStart`			Test case or control part execution is about to start. Generated before the execution starts, and therefore these events merely indicate an attempt to start the execution.
A	`TcStop` `CtrlStop`	`stop`		Request to stop execution of a test case or control part.
B	`TcStarted`			Test case execution has been started.
B	`TcTerminated` `CtrlTerminated`			Test case or control part execution has finished.

B	`AEnter`
	`ALeave`
	`ANomatch`
	`ARepeat`
	`AWait`
	`ADefaults`
				Control flow during evaluation of an alt statement.

B	`SEnter` `SLeave`			Entering or leaving a scope.

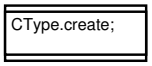
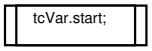
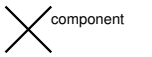
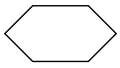
[1] Event classification: See Section 6.4.5

provide fine-grained details about the execution. TTCN-3 language defines seven basic units of scope: module definitions, control part of a module, component types, functions, altsteps, test cases and statement blocks [20]. While an argument could certainly be made for considering these events in the visualization, they have been omitted in the scope of this work. This decision is based on a simple reason: OpenTTCN Tester [58] is the TTCN-3 tool of choice in this thesis, and the present version of the tool does not generate the scope transition events.

6.4.2 Configuration events

The configuration events provide details about the dynamic configuration of the test system. As TTCN-3 code is executed, additional test components may be created and also the communication paths between ports of the system components are dynamically constructed during the execution. Table 2 shows the configuration operations related to test components. Most of these events are generated immediately after execution of a specific TTCN-3 operation, but the table includes a diverse set of events, as explained below.

Table 2: Test component configuration events

[1]	**Event (tli..)**	**CL statement**	**Symbol**	**Comments**
A	`CCreate`	`tc.create`		Test component instantiation.
A	`CStart`	`tc.start`		Start execution of a test component. Also serves as a hyperlink to the function diagram displaying the test component behaviour.
A	`CStop` `CKill`	`tc.stop` `tc.kill`		Request to stop a test component, with the component identifier attached to the stop symbol.
A	`CKilled` `CDone`	`tc.killed` `tc.done`		Matching operations for component events. To be used standalone or inside an alt.
D	`CKilledMismatch` `CDoneMismatch`			See mismatch visualization in Section 6.7.
B	`CTerminated`			Test component has finished its execution.
E	`CRunning` `CAlive`	`tc.running` `tc.alive`		No graphical symbol is defined: these operations evaluate to Boolean expressions and are denoted textually whenever being used.

[1] Event classification: See Section 6.4.5

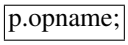
The `create`, `start`, `stop` and `kill` operations indicate direct requests to affect a test component. As we recall from Section 5.6, parallel test components may be of either alive or non-alive type. For components of non-alive type, stop and kill are equivalent: the component always dies once it is stopped. On the other hand, for a component of alive type, the stop operation will only stop execution of the currently assigned behaviour, allowing the component to be re-started later on while the kill operation will terminate the component permanently. The GFT standard does not define a symbol for the kill operation, but due to its close relation to the stop operation, these two are grouped together.

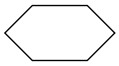
The `done` and `killed` operations are very different in nature: they are used as matching operations in alternatives. They match whenever the component finishes is execution or is killed. If these operations are used alone outside an explicit alternative, the execution will block until the component finishes execution or terminates, respectively. Finally, the `running` and `alive` events indicate execution of statements that may be used to test whether a component is running or alive. These two statements evaluate to boolean expressions, and are of little interest in visualization.

Table 3 shows events that are related to ports and default alternative behaviour. The first set of events (`map` and `connect`) describe the setup of communication channels. The second group contains operations that affect the behaviour of a port. The `clear` operation empties the incoming message queue while the `start` and `stop` operations enable or disable all communication operations on the port. Halting a port is almost equivalent to stopping, but a `halt`ed port still allows the existing messages to be processed from the incoming message queue. The GFT standard does not mention the halt operation, but it is grouped together with the other similar operations.

Table 3: Port and default alternative configuration events

[1]	**Event (tli..)**	**CL statement**	**Symbol**	**Comments**

A	`PConnect`
	`PDisconnect`
	`PMap`
	`PUnmap`
		`p.connect`
		`p.disconnect`
		`p.map`
		`p.unmap`
				Setup communication paths between ports.

A	`PClear`
	`PStart`
	`PStop`
	`PHalt`
		`p.clear`
		`p.start`
		`p.stop`
		`p.halt`
				Manipulate the behaviour of a local port. The operation name is written inside the symbol.

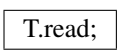
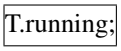
A	`AActivate` `ADeactivate`	`activate` `deactivate`		Puts an altstep into the list of defaults. Graphical presentation is the default symbol with the operation name inside.

[1] Event classification: See Section 6.4.5

The `activate` and `deactivate` operations manipulate the list of default alternatives. When an altstep is activated, it is appended into the list of default alternatives. Deactivation removes the altstep from the list. As we recall from Section 5.5, the default alternatives are typically used to provide handlers for common unexpected behaviour and they are implicitly appended into any alt statement being executed.

Table 4 shows the timer events. Timers are generally used to provide timeouts in scenarios where the system under test does not respond within the expected time frame. Typically the test system vendor provides at least a default linear timer implementation, which follows the real, wall-clock time. Additionally, the platform adaptor may provide a custom timer implementation to provide for example scaled, non-linear or event-driven notion of time.

Table 4: Timer events

[1]	Event (tliT..)	CL statement	Symbol	Comments
A	`Start`	`T.start`		Start a timer. Timer identifier is attached to the symbol.
A	`Stop`	`T.stop`		Timer has been stopped.
A	`Timeout`	`T.timeout`		Timeout has been matched and the relevant statement block will be executed.
A	`Read`	`T.read`		Read elapsed time from a timer.
A	`Running`	`T.running`		Check if a timer is running.
C	`TimeoutDetected`			A timeout has been enqueued from the Platform Adaptor using the `triTimeout` operation. Enqueued events visualization is discussed in Section 6.6.
D	`TimeoutMismatch`			Mismatch visualization is discussed in Section 6.7.

[1] Event classification: See Section 6.4.5

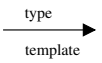
6.4.3 Communication events

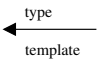
Many communication operations have multiple, effectively similar versions for different scenarios such as various addressing models (unicast, broadcast, multicast) or differentiating between communication with the SUT and communication between peer test components. Distinction between these versions is done by appending a suffix or suffixes to the event name. The `_c` suffix indicates an intercomponent operation via a connected port while the `_m` suffix indicates communication with the system under test via a mapped port. The default addressing model is unicast, which means from one sender to one receiver. The `_MC` suffix indicates multicast communication, meaning from one sender to multiple receivers and the `_BC` suffix indicates broadcast communication, which means from one sender to all receivers.

The communication events have been split into three categories based on the communication paradigm. Table 5 presents the events related to message-based communication. As we recall from Section 5.9, message-based communication does not make a distinction between the roles of the communication partners while procedure-based communication makes the roles explicit. Therefore, for message-based communication only

Table 5: Message-based communication

[1]	**Event (tliM...)**	**CL statement**	**Symbol**	**Comments**

A	`Send_m`
	`Send_m_BC`
	`Send_m_MC`
	`Send_c`
	`Send_c_BC`
	`Send_c_MC`
		`p.send`		Sending of a message via a port. Represented by the arrow symbol drawn from a test component instance to a port instance.

A	`Receive_m` `Receive_c`	`p.receive`		Message has been received (and matched) from a port. Represented by the arrow symbol drawn from a port instance to a test component instance.
C	`Detected_m` `Detected_c`			A message has been appended to the incoming queue of a port. Further discussed in Section 6.6.
D	`Mismatch_m` `Mismatch_c`			See mismatch visualization in Section 6.7.

[1] Event classification: See Section 6.4.5

a single set of events exists, but the procedure-based communication events have been further grouped by the role of the test system.

Table 6 shows the events related to procedure-based communication in a scenario where the test system acts as a client and the system under test as a server. Client invokes operations on the server by using the TTCN-3 `call` statement. The execution of a call statement has special semantics: the activated defaults are not effective. To guard against deadlocks caused by this, an optional guard timer may be enabled by supplying a timeout as a parameter to the call statement. The `CatchTimeout` indicates catching of this kind of timeout.

The test system may also need to take the role of a server, for example when testing a client implementation or when using procedure-based communication for passing data between test components. Compared to the case where the test system operates as a client, all the concepts are the same, only the roles are reversed. The client *calls* an operation on the server, and the server responds either with a *reply* or *raises* an exception. This group of events is shown in Table 7. In fact, the procedure-based communication symbols shown in the tables 6 and 7 contain only a partial representation. GFT specifies that a blocking call is placed inside an inline expression symbol with the possible responses to the blocking call operation, and a subsequent suspension region is drawn on the test component instance that performs the call. An example of inline expression symbol used for representation of an alt statement was shown previously in Figure 21.

Table 6: Procedure-based communication, TE acts as a client

[1]	Event (tliPr...)	CL statement	Symbol	Comments

A	`Call_m`
	`Call_m_BC`
	`Call_m_MC`
	`Call_c`
	`Call_c_BC`
	`Call_c_MC`
		`p.call`	**call** proc → template	A procedure call has been executed on the SUT or on a peer test component.

A	`GetReply_m` `GetReply_c`	`p.getreply`	**getreply** proc ← template	A response to a previous call has been received.
A	`Catch_m` `Catch_c`	`p.catch`	**catch** proc ← template	An exception from a called entity has been caught.
A	`CatchTimeout`	`p.catch (timeout)`		A timeout exception has been caught.

C	`GetReplyDetected_m`
	`GetReplyDetected_c`
	`CatchDetected_m`
	`CatchDetected_c`
	`CatchTimeoutDetected`
				See enqueued events visualization in Section 6.6.

D	`GetReplyMismatch_m`
	`GetReplyMismatch_c`
	`CatchMismatch_m`
	`CatchMismatch_c`
				A queued reply or catching of an exception did not match a template. Further discussed in Section 6.7.

[1] Event classification: See Section 6.4.5

Table 7: Procedure-based communication, TE acts as a server

[1]	Event (tliPr...)	CL statement	Symbol	Comments
A	`GetCall_m` `GetCall_c`	`p.getcall`	**getcall** proc ← template	An enqueued call was succesfully matched against a template and the getcall operation was executed.

A	`Reply_m`
	`Reply_m_BC`
	`Reply_m_MC`
	`Reply_c`
	`Reply_c_BC`
	`Reply_c_MC`
		`p.reply`	**reply** proc → template	A reply (to a getcall) operation has been executed on the SUT or a peer test component.

A	`Raise_m`
	`Raise_m_BC`
	`Raise_m_MC`
	`Raise_c`
	`Raise_c_BC`
	`Raise_c_MC`
		`p.raise`	**raise** proc → template	A raise operation has been executed.

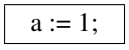
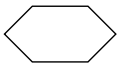
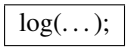
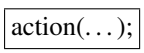
C	`GetCallDetected_m` `GetCallDetected_c`			An incoming call has been inserted into the incoming queue of a port. Further discussed in Section 6.6.
D	`GetCallMismatch_m` `GetCallMismatch_c`			A call that was inserted into the queue did not match a template. Further discussed in Section 6.7.

[1] Event classification: See Section 6.4.5

6.4.4 Miscellaneous events

Table 8 lists the remaining events that did not fit into any of the other categories. Variable assignment or accessing the value of a module parameter are common events during program execution, but are often worth logging only when looking into a very specific piece of code. The `match` operation shown in this table is not to be confused with the matching performed in the alternatives: this match operation compares a value with a template, and it evaluates to a boolean expression. The `tliEncode` and `tliDecode` events indicate conversion between a TTCN-3 value and the transfer syntax. The purpose of the conversion is to support the abstract nature of TTCN-3, and therefore these events are typically not immediately interesting. On the other hand, being able to inspect the transfer syntax alongside with the corresponding TTCN-3 value is beneficial in scenarios such as potential codec malfunction. Regardless, data presentation is out of the scope of this work and consequently these events have been classified as irrelevant. Finally, the `setverdict` event is obviously extremely important in log analysis.

Table 8: Miscellaneous events

[1]	**Event (tli..)**	**CL statement**	**Symbol**	**Comments**
E	`Var`	`a := 1`		Assignment of a value to a variable.
E	`ModulePar`			Accessing the value of a module parameter.
E	`GetVerdict`	`getverdict`		Accessing the value of the local verdict.
A	`SetVerdict`	`setverdict`		Assignment of the local verdict. Value of the verdict is shown inside the symbol.
A	`Log`	`log`		Execution of TTCN-3 log statement, which may be used to log for example free-form text or value of a variable.
A	`Action`	`action`		Request the user to perform an action on the system under test.
E	`Match` `MatchMismatch`	`match`		Succesful or mismatching match operation, which compares a value with a template.
E	`Encode` `Decode`	`encvalue` `decvalue` or implicitly during communication		Request to encode or decode a value at the CD entity.

[1] Event classification: See Section 6.4.5

6.4.5 Classification of the events

The findings indicate that a one-to-one mapping between a TTCN-3 statement, a log event and a GFT symbol exists for majority of the events. However, it turns out that the events that do not fit into this category are certainly not irrelevant for the purposes of visualization. The events were initially divided into groups just to structure the presentation, but in the analysis some slightly different groups were identified. To conclude the results, from the perspective of visualization techniques and the visualization tool, the events may be categorized as follows.

A. Events that indicate execution of a specific TTCN-3 operation and have a standardized graphical presentation. The standard symbol may be adopted for log presentation. However, simply visualizing all of these events is not necessarily the best approach, as it would most likely lead to extremely large diagrams. A proper visualization tool should give the user an option to enable or disable the visualization depending on event type.

B. Events that do not necessarily provide anything immediately interesting to the user, but that do affect the diagram construction and layout by providing information about the control flow to the visualization tool. These events are discussed in Section 6.5.

C. Enqueued events. These events indicate that something was inserted into the incoming queue of a port. This group of events is primarily useful for debugging faulty test cases that do not properly handle all the input from the message queue. Further discussion of these events is given in Section 6.6.

D. Mismatch events. These events are generated whenever something did not match the expected input, and they provide valuable information for identifying the error in the system under test. Additionally, mismatches may reveal errors in the test itself – especially in a test suite that is under development. Further discussion is given in Section 6.7.

E. Events considered to be irrelevant in the scope of this thesis.

6.5 Visualizing diagram transitions

As we recall from Section 5.8, the graphical presentation format specifies four types of diagrams: control part, test case, function and altstep diagram. Diagrams indicate a reference to another diagram with either the execute symbol (special case for execute test case statement) or the reference symbol. Examples of these symbols were shown in Figure 20.

From log visualization perspective, the diagram references may be represented using the respective symbol as a hyperlink, which the user can click to open the referenced diagram. The information necessary to support this feature is provided in the log events selected for group B in our classification. An example of the hyperlink approach in the context of the Hello example will be given in Section 6.8.1.

Nevertheless, the hyperlink approach is not necessarily the most convenient for the user. Schieferdecker et al. [41] describe a technique called HyperMSC, which allows selectively expanding the referenced diagrams inline. This approach could also be combined with the concept of *semantic zooming*. In contrast to the traditional geometric zooming, where zooming affects only the size of the objects, semantic zooming modifies also the appearance of the objects. The appearance is typically dictated by the amount of screen real-estate available to the object. When zoomed out, an object could be presented using a point but as the user zooms closer, the object will gradually contain more and more details [60, 29]. Sedlmair [69] already reported positive experiences from utilizing semantic zooming for displaying message sequence charts.

In scenarios with limited screen real estate available, the semantic zooming approach could also provide a single compact view for browsing the log data: overview could start with a list of test campaign logs recorded in the database. Zooming inside the campaign would show a list of test cases executed during the campaign. Following this pattern, given enough zooming the view would eventually display fine details of an individual log event. Furthermore, [61] proposes various context aids that help the user to keep track of the current position in the information space.

6.6 Visualizing enqueued events

The log events with the `Detected` suffix in their name have been grouped together as *enqueued events*. This name refers to the relevant family of operations in the TRI interface. In order to relay something from the SUT to the TE, the system adaptor invokes an `enqueue` operation.

These events do not yet indicate execution of any TTCN-3 code. Instead, they are generated either by the SA or the TE to indicate that some data has been inserted into the incoming queue of a test component port [4]. It is possible that the queue is never processed in the TTCN-3 code. Therefore, these events do not necessarily have any effect in the execution. On the other hand, these events are certainly important for diagnosing errors, especially in the case of inadequate test suites that do not properly handle the input. The action symbol, attached to the relevant port instance, has been adopted for visualizing these events.

These events are closely coupled with the `tliDecode` event. Before being able to process an incoming message in the TTCN-3 code, the test system must convert the message to the abstract TTCN-3 value by requesting decoding from the CD entity. As a consequence, inserting something into the input queue will always cause a `tliDecode` event to be generated. This is illustrated in the visualization examples presented later in Section 6.8.

6.7 Visualizing alternative behaviour and mismatches

As discussed in Section 5.5, alternatives are used to branch the test behaviour based on detected events, such as receiving of a message or expiration of timeout. During evaluation of an altstep, the TTCN-3 tool traverses the list of alternatives and looks for a branch that may be matched. For example, the receive statement matches if the expected message is found in the incoming queue. Consequently, if the message is not found, the receive event does not match. In other words, a *mismatch* has been detected. Visualization of alternatives and mismatches is discussed by using the simple altstep shown in Listing 5 as an example.

[4] This is my interpretation of the TCI standard [24]. It is not obvious whether these events should be generated already upon insertion of a message into a test system interface port, or once the message reaches a test component.

Listing 5: altstep DemoAlt

```
1 altstep DemoAlt() {
2   [] p.receive(msg_A) { setverdict(pass); }
3   [] p.receive(msg_B) { setverdict(fail, "unexpected message B"); }
4   [] p.receive(msg_C) { repeat; }
5   [] p.receive { setverdict(fail, "unknown message"); }
6   [] T.timeout { setverdict(fail, "response supervising timer timeout"); }
7 }
```

The altstep `DemoAlt` is prepared to handle several scenarios: receiving of the expected message (msg_A), receiving of a known unexpected message (msg_B), receiving a message that causes the alternative to be repeated (msg_C), something completely unrelated or a timeout. Figure 24 shows the equivalent GFT representation.

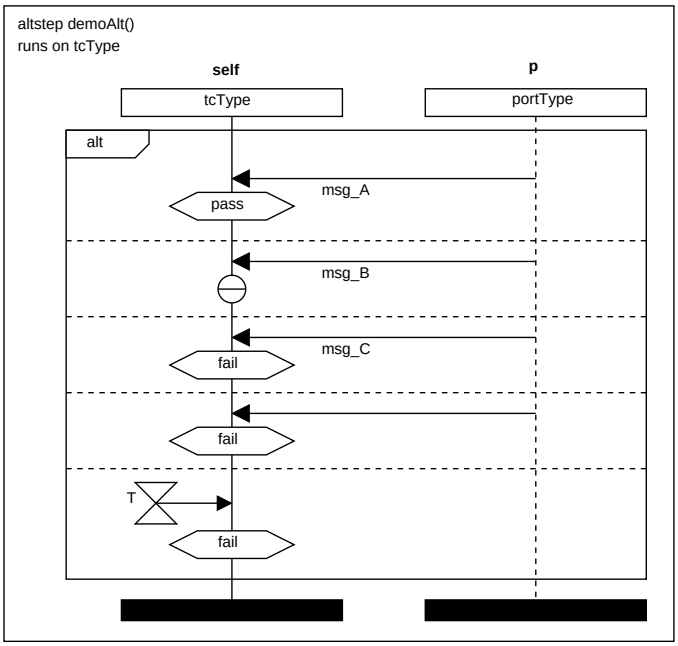


Figure 24: GFT representation of DemoAlt

A scenario where the "receive any message" alternative gets selected is used as an example. In other words, something totally unexpected was received, and this will result in several mismatches. The `tliAEnter` and `tliALeave` events have been utilized for opening and closing the inline expression (alt) region in the diagram and the action symbol is adopted for presenting the mismatch events. Figure 25 shows the resulting diagram.

As precondition we assume that a message exists in the input queue of port `p`, and the TE begins execution of the alternative. The message does not match any of the predefined templates, but finally gets handled by the receive statement that accepts any message.

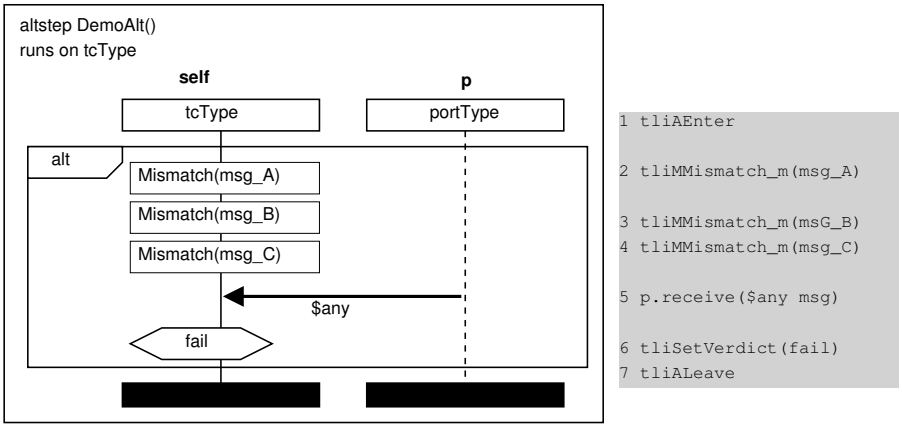


Figure 25: Log of DemoAlt

This presentation technique only shows that the template did not match, but in reality the mismatch events also carry information about the *difference* between the expected and received value. Obviously this is valuable information, but the data presentation goes out the scope of this work.

Particularly with the data presentation omitted, the usefulness of including this kind of mismatch events in the visualization can be questioned. The diagram obviously shows that matching against the templates was attempted, but still tells very little besides the fact that receiving "any message" led to the assignment of fail verdict.

Alternative approach to visualizing the alternative and mismatching behaviour would be to combine the log visualization with the GFT representation (shown in Figure 24) of the source code. Displaying the visualized source code first and then indicating the selected alternative branch would lead to a more powerful presentation due to inclusion of the expected behaviour in the same presentation. It would be immediately obvious to the viewer that msg_A would have led to pass verdict, and that also msg_B would have allowed the test to continue. Unfortunately, constructing this kind of presentation by utilizing only information provided in the log events is impossible.

6.8 Examples

This section gives examples of applying the log visualization to the examples presented in the previous chapters. The first example shows the fundamental concepts, such as diagram generation and references using the simple but fundamentally flawed Hello ex-

ample introduced in Listing 1. The second example shows examples of mismatch and enqueued events visualization using the Session Initiation Protocol test case shown in Listing 4.

6.8.1 Hello

The textual log of a succesful execution was shown in Listing 2. The control part, which is represented using the control diagram, is the main entry point to the execution. If we wish to present the entire log in graphical format, log of the control part is the first view given to the user. In the hello test case there is no control logic other than a single request to execute the test case `TC_hello`. The graphical presentation of the log in fact looks exactly like the graphical presentation of the source code, as illustrated by Figure 26. Color coding has been introduced to indicate the verdict.

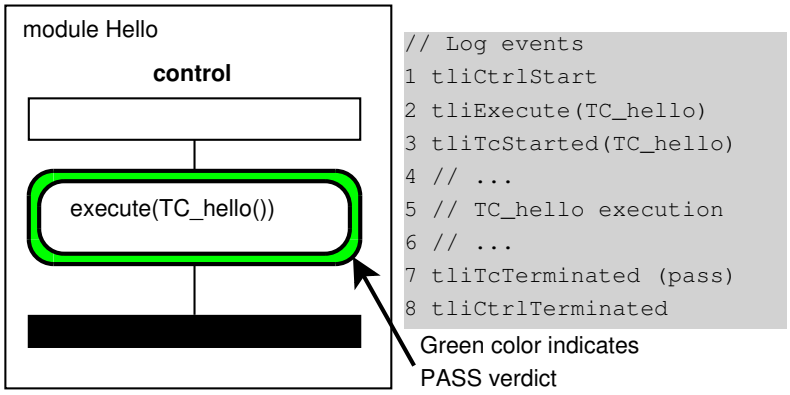
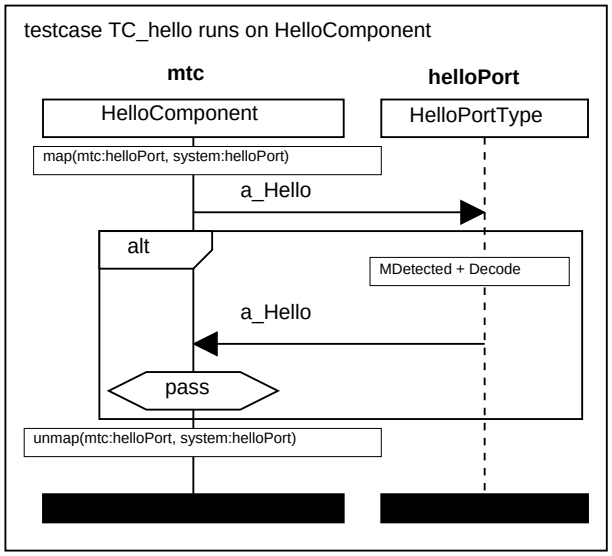


Figure 26: Visualization of control part log

The Visual Information Seeking Mantra introduced in Section 4.4 says *Overview first, zoom and filter, then details-on-demand*. The overview is the entire control part. In this specific case zooming and filtering would add little benefit, but if we consider a test suite with tens of test cases, an argument could be certainly made for providing an option to filter the displayed test cases based on the verdict, for example. This would allow the user to quickly focus his attention on the test cases that did not pass.

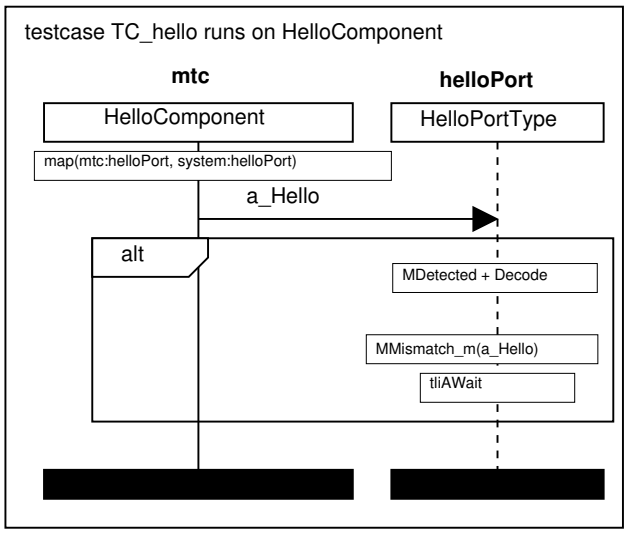
Second part of the mantra, details-on-demand, suggests that the user is able to demand more details. In this example, the log for the test case itself would be a candidate for the details. In practice, clicking on the execute symbol could take the user to the test case diagram shown in Figure 27.




```
// Log events
1 tliTcStart(TC_hello)


2 tliPMap(mtc:helloPort,
          system:helloPort)
3 tliMSend_m(helloPort, a_Hello)
4 tliAEnter
5 tliMDetected_m
6 tliDecode, results in a_Hello

7 tliMReceive_m(helloPort, a_Hello)
8 tliSetVerdict(pass)
9 tliALeave
10 tliPUnmap(mtc:helloPort,
             system:helloPort)

11 tliTcTerminated
```

Figure 27: Visualization of test case log - case pass

This log diagram illustrates also the fact that a stand-alone receive statement is implicitly an alt with only one alternative branch. In this scenario the execution completed as expected. If we consider a case where the response is something else than the expected template, the log would look like shown in Figure 28.




```
// Log events
1 tliTcStart(TC_hello)


2 tliPMap(mtc:helloPort,
          system:helloPort)
3 tliMSend_m(helloPort, a_Hello)
4 tliAEnter
5 tliMDetected_m
6 tliDecode // results in "HI"
7 tliMMismatch_m(a_Hello)
9 tliANomatch
10 tliAWait
// We are prepared to handle only
// a_Hello => DEADLOCK
```

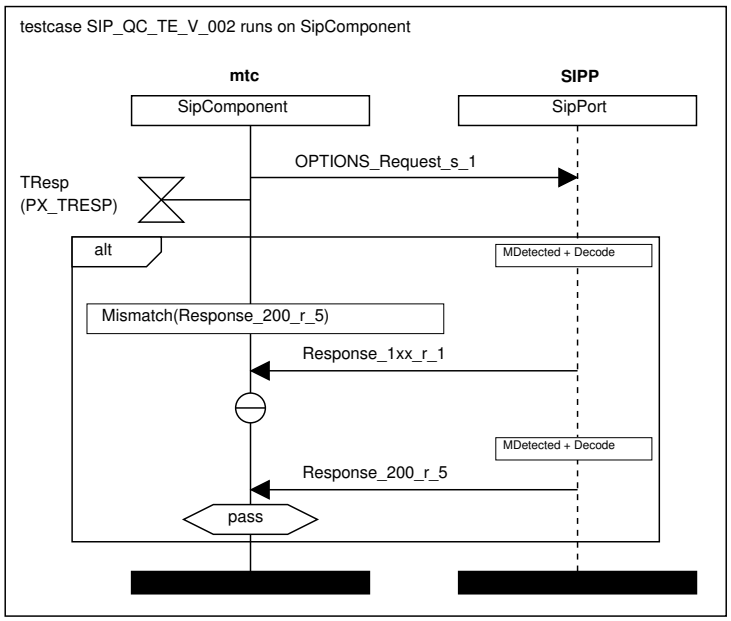
Figure 28: Visualization of test case log - case deadlock

Now the log shows that the `a_Hello` template was sent to the SUT via helloPort, and the implicit alt of the `receive` operation was entered. SUT replied asynchronously with a message that was decoded into "HI", which is not what was expected. Therefore a message mismatch event was generated, followed by mismatch of the entire alternative. Finally, the `tliAWait` indicates that the system is waiting for events for a new snapshot

before re-evaluation of the alt. This is where the system deadlocks, because the test case did not include any handling of unexpected responses.

6.8.2 SIP test suite

In Section 5.9.1 the SIP_QC_TE_V_002 testcase was previously used to demonstrate a typical communication pattern and its representation as a GFT diagram. Figure 29 shows an example of a possible execution log.




```
// Log events
1 tliTcStart(SIP_QC_TE_V_002)

// Preamble omitted

2 tliMSend_m(SipPort, OPTIONS_Req)
3 tliTStart(TResp, PX_TRESP);
4 tliAEnter
5 tliMDetected_m
6 tliDecode => Response_1xx_r_1

7 tliMMismatch(Response_200)

8 tliMReceive_m(Response_1xx)
9 tliARepeat
10 tliMDetected_m
11 tliDecode => Response_200_r_5

12 tliMReceive_m(Responose_200)
13 tliSetVerdict(pass)
14 tliALeave
// Postamble omitted
15 tliTcTerminated
```

Figure 29: Visualization of SIP_QC_TE_V_002 log

This example illustrates the previously discussed enqueued events, mismatch events and the repeat statement in alternatives. The test case is specified to repeat the main alternative if the SUT responds with a message that matches the `Response_1xx_r_1` template. In this example, this particular response was received, followed by a message that matched the expected `Response_200_r_5` template. As a consequence, the test ended with the pass verdict.

7 THE REPRISE VISUALIZATION TOOL

7.1 Environment

In practice software development requires a wide variety of software tools that make possible and support the creation of new software. Integrated Developmet Environments (IDE) aim to make software developers more productive. Because of this, the IDEs keep constantly evolving as new useful features are discovered [65]. This is also true in our case. Reprise, our software tool for viewing and visualizing TTCN-3 execution logs, integrates into a development environment which previously included an editor, a compiler, a virtual machine and other components necessary for test development and execution.

The Eclipse Platform is a cross-platform integrated development environment, published under an open source license and available for download at [76]. Eclipse was created to provide a common platform for diverse applications and to facilitate their integration [16]. To support developing extensions, Eclipse is built using a powerful plug-in architecture. Plug-ins are the smallest unit of Eclipse Platform function that can be developed and delivered separately. In fact, except for a small kernel known as the Platform Runtime, also the Eclipse Platform itself consists of various plug-ins. While a small tool might be written as a single plug-in, complex tools typically split their functionality across several plug-ins.

The TRex TTCN-3 Refactoring and Metrics tool has been developed as a collaborative effort of Motorola and University of Göttingen. TRex extends Eclipse with TTCN-3 editing and refactoring[5] facilities [4], and is available along with the source code from [80].

The OpenTTCN Tester tool integrates the OpenTTCN TTCN-3 compiler and virtual machine with the Eclipse Platform and TRex. Therefore the Reprise tool being developed is essentially a collection of Eclipse plug-ins that supplement the existing system by providing various views for reviewing previously recorded execution logs. Figure 30 illustrates the principle. The plug-ins are implemented in Java language, but the code may employ components written in other languages. Many of the OpenTTCN Tester plug-ins merely provide a graphical user interface for an underlying command-line tool.

[5]Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure [28].

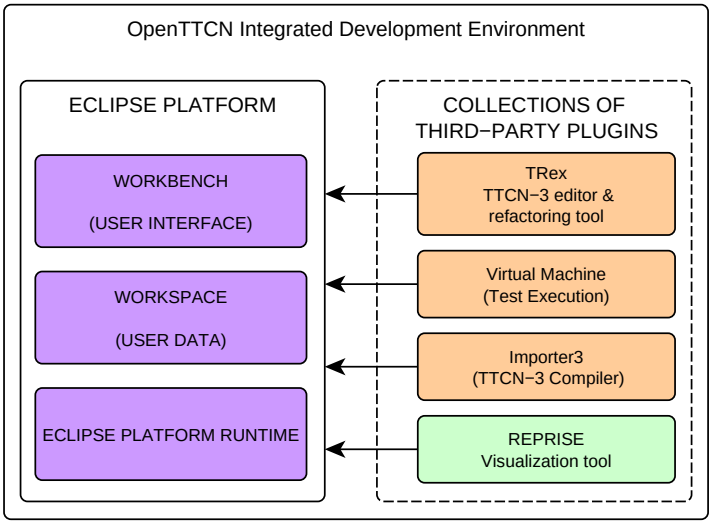


Figure 30: OpenTTCN Integrated Development Environment

The plug-ins are typically bundled inside Java archives (JAR), and placed into the appropriate place in the Eclipse directory hierarchy. Each plug-in has a manifest file, which declares its interconnections to other plug-ins. During startup, the Eclipse Platform Runtime discovers the set of available plug-ins and builds a plug-in registry based on the respective manifest files. Although the manifest information of all plug-ins is loaded immediately during startup, the plugins are fully activated only on demand. This lazy-loading approach allows having a large base of installed but rarely used plugins without increasing the memory footprint or startup time [16]. Eclipse also makes an excellent job at hiding this complexity from the end-user: the on-demand activation of a plug-in happens automatically and entirely in the background without requiring any effort from the user.

7.2 Eclipse User Interface

The Eclipse Platform UI paradigm is based on three fundamental concepts: *editors, views and perspectives* [16]. Figure 31 illustrates this concept in practice by using the OpenTTCN Tester user interface as an example. It is worth noticing that Eclipse always leaves the ultimate control to the end-user. The arrangement of the user interface elements is highly customizable, and the initial layout presented here is merely the default value.

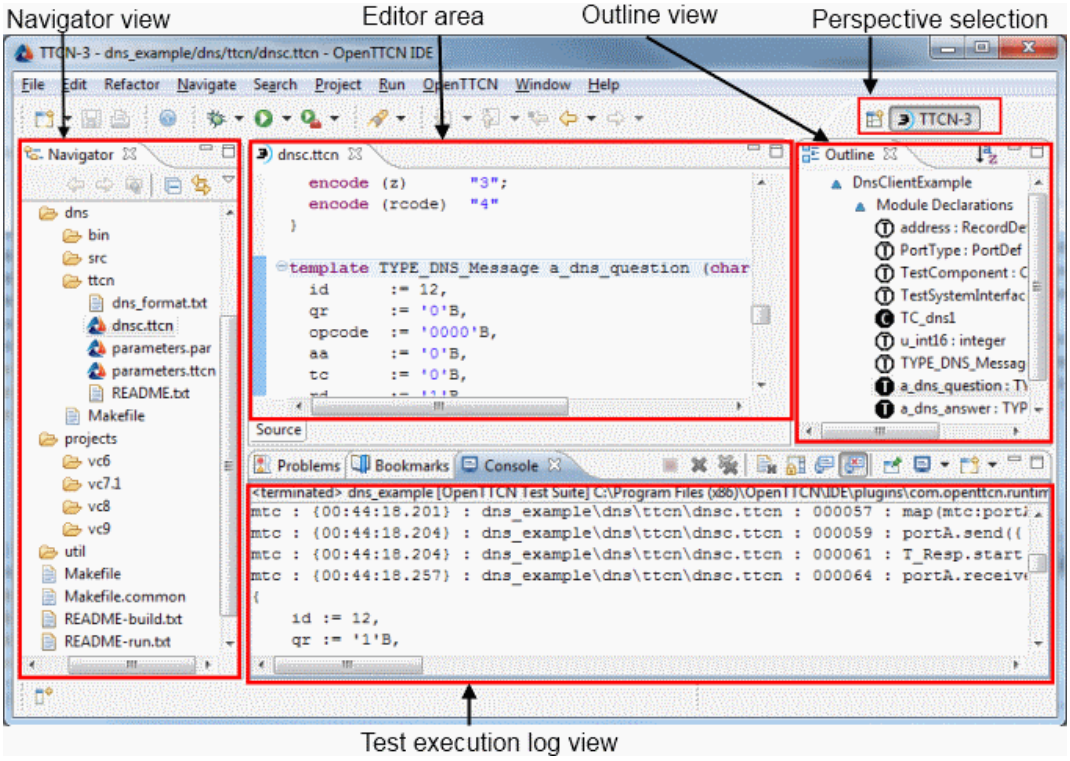


Figure 31: OpenTTCN IDE Overview

The *navigator view* is used for navigating the user resources, which are effectively files and folders in the underlying file system. The *editor area* allows the user to edit resources selected from the workspace. In this case, the editor view shows the editor of the TRex tool. TRex editor extends the default Eclipse text editor with various features relevant to TTCN-3, such as syntax highlighting. The *outline view* may be used for quick navigation in navigate the editor. The *test execution log view* shows the current test execution log display, which is effectively a capture of the output of the `tester` command-line tool. The tester command provides means for initiating and controlling the test execution – in TTCN-3 terms it implements the TM and TL entities.

The *perspective selection* button allows the user to switch between perspectives. Each perspective has its own views and editors that are arranged for presentation on the screen. The perspective controls the initial layout and visibility of the views inside it. Perspectives are designed as a way for the user to quickly switch between tasks, and to retain the possibly customized layout for each task. The TTCN-3 perspective shown in this example is primarily an editing perspective provided by TRex. OpenTTCN Tester further extends it by adding test execution and viewing of the execution log. The Reprise visualization tool will add its own perspective, which supports the task of log analysis.

7.3 Design considerations

7.3.1 Online versus offline visualization

We are visualizing test execution logs, which could be also interpreted to imply that the visualization is done offline. In other words the visualization is done entirely after and separately from the actual execution. Despite lacking live display of the execution, the offline approach has also its benefits: The data may be preprocessed as a whole prior to rendering the visualization and the visualization can be made navigable in a way that would not be possible online. For instance, the trace might be reviewed, even backwards, starting from any arbitrary point at a rate independent of the original execution speed [81].

Obviously it would be possible to visualize the data online as it becomes available during the execution, but a design decision has been made to limit the initial functionality so that traces may only be viewed after the execution has finished. In addition to benefits discussed above, this approach relieves Reprise from performance considerations. Especially if multiple components of the test system are running on the same physical computer, one has to be cautious with introducing additional overhead that may slow down test execution and introduce various unpredictable delays.

7.3.2 Related work

Arguably our task of drawing message sequence charts from seemingly limited set of data is trivial compared to many challenges in visualization applications that deal with large and multidimensional datasets. Regardless, any visualization tool is responsible for mapping a set of data to a graphical presentation. Software design patterns, as made famous by [30], are a method for describing reusable solutions to common software design tasks. While most of the visualization publications seem to concentrate on developing novel visualization ideas and barely discuss the software design aspect at all, a literature review revealed a couple of papers which are worth briefly crediting here. Tang [75] presents experiences and some key decisions involved with the design of the Rivet [10] visualization environment. Heer [38] even goes as far as reviewing various existing visualization frameworks and capturing the recurring design decisions into more formal design patterns. Furthermore, the paper serves as a good overview of what kind of frameworks are available for building information visualization applications.

7.4 Implementation

As stated above, the Reprise log analysis and visualization tool is built as a set of plug-ins integrated to the existing system. This section describes the implementation of the tool and is structured as follows. Section 7.4.1 gives an overview of the entire logging subsystem in OpenTTCN Tester and the Reprise tool. The overview is followed by a more detailed description of reprise, split to several sections. Section 7.4.2 introduces the log repository and data model. Overview of the user interface is given in Section 7.4.3 and finally the graphical log view (visualizer) is described in Section 7.4.4.

7.4.1 Architecture overview

OpenTTCN Tester is a distributed system on its own. The test system entities are interconnected using the Common Object Request Broker Architecture (CORBA) [57] middleware technology. The overall architecture of the logging subsystem is illustrated in Figure 32. The campaign recorder acts as a bridge between the existing system and the newly developed components.

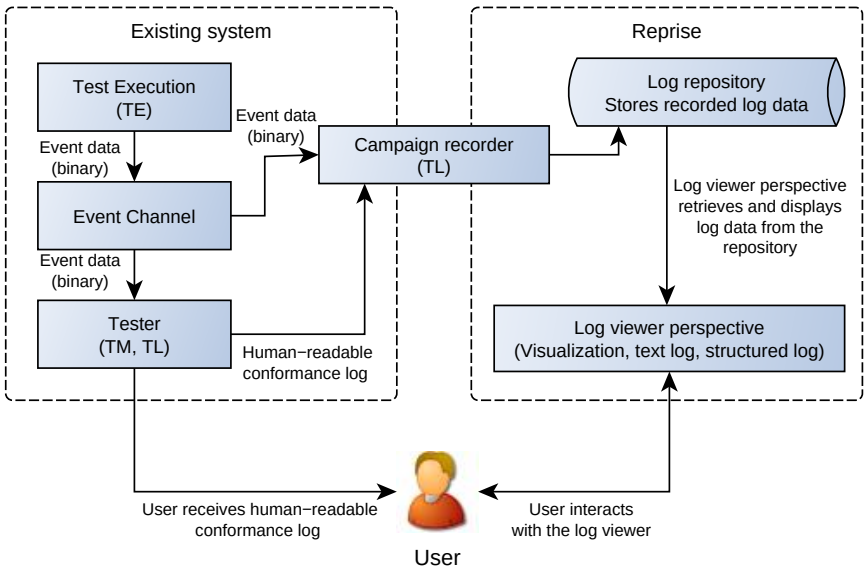


Figure 32: Overall architecture

The events are received via the event channel. The concept of event channel is defined in the CORBA Event Service specification [56] as *"intervening object that allows multiple suppliers to communicate with multiple consumers asynchronously"*. The event service

decouples the communication partners by acting as a man in the middle. Two roles are specified for the communication partners: *suppliers* produce event data to *consumers*. In our case the event channel is implemented in the `openttcnd` background service. The test components act as event suppliers and the TL entities as consumers.

Majority of the existing system such as the virtual machine (test components), background services, and the tester tool are implemented in C++ language, with omniORB [35] as the CORBA implementation. On the other hand, Java is the obvious choice for implementing Eclipse plug-ins. Interoperability between components written in different languages would be trivial over CORBA, and therefore the most obvious and appealing approach would be to write the Reprise tool entirely in Java. However, in order to be able to reuse the existing codebase, some parts of the tool were written in C++ , as described in the subsequent sections.

The tool is split to three separate Eclipse plug-ins as follows:

1. `org.etsi.ttcn.tc.tl.model`: Defines the application data model.
2. `com.openttcn.reprise.repository`: Constructs an instance of the model from the recorded log data.
3. `com.openttcn.reprise.ui`: Constructs the presentation (user interface) based on the model.

7.4.2 Repository and data model

Figure 33 shows an architectural illustration of the data flow in the Reprise tool. The campaign recorder saves the log data in the filesystem as two files per test campaign: `log.txt` contains the plaintext log and `log.bin` contains the received events as serialized CORBA data structures. This approach allows a very simple and efficient implementation of the campaign recorder, as the received data is dumped to the storage as-is, without performing any conversions. The lightweight implementation is beneficial, as we wish to minimize the overhead that might degrade the overall system performance during test execution. The campaign recorder is implemented in C++ language and it is started as a part of the test campaign startup by the OpenTTCN test launcher. However, the startup procedure goes outside the scope of this document. We can just conclude

that for each recorded test campaign the two files described above will appear in the filesystem.

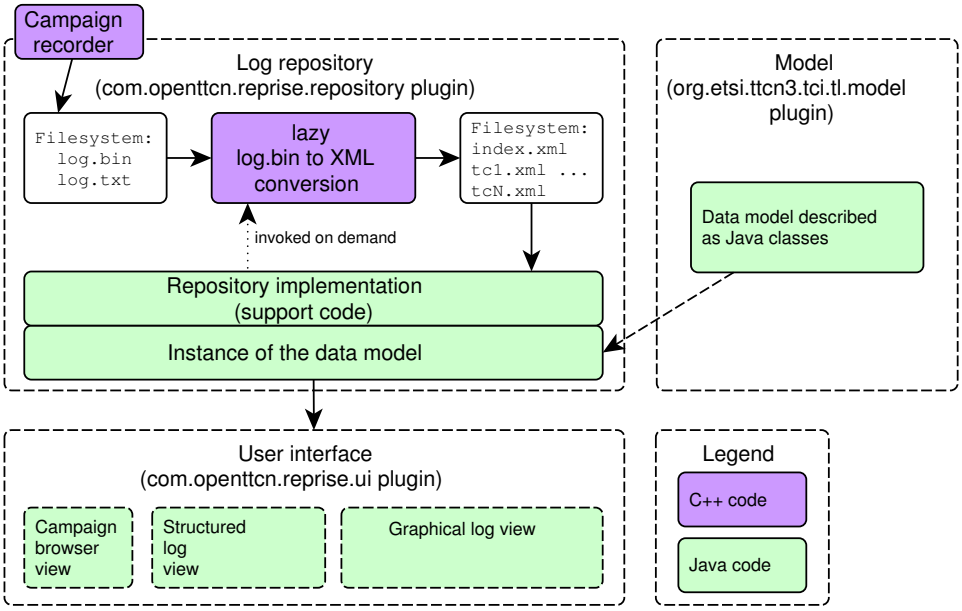


Figure 33: Reprise architecture

Whenever a previously recorded log is accessed for the first time, the data is converted from CORBA data structures to XML [84] format. The CORBA serialization mechanism is standardized, but the structure of the data is proprietary to OpenTTCN Tester. On the other hand, the structure of the XML documents is defined using XML schema [86] in the TCI standard [24]. Due to the existing OpenTTCN codebase having large amount of C++ code for handling the log events, the conversion tool was most conveniently written in C++ as a shared library. The conversion routine is accessed from Java code through the Java Native Interface [73].

The main responsibility of the repository is to provide the log data to the log viewer in a convenient format. In other words, the repository separates the user interface from the raw data files by offering an abstract model of the data. The model is a specification of the application data, and in this case it is structured in a hierarchical fashion as illustrated in Figure 34. Essentially the root of the hierarchy is the log repository, which may contain zero or more recorded test campaigns. Each test campaign contains zero or more test cases. It is possible that TTCN-3 execution never leaves the control part, and therefore a test campaign with no test cases is perfectly valid. A test case, however, always has at least two events: `tliTcStarted` and `tliTcTerminated`, which indicate the startup and termination of the test case, respectively.

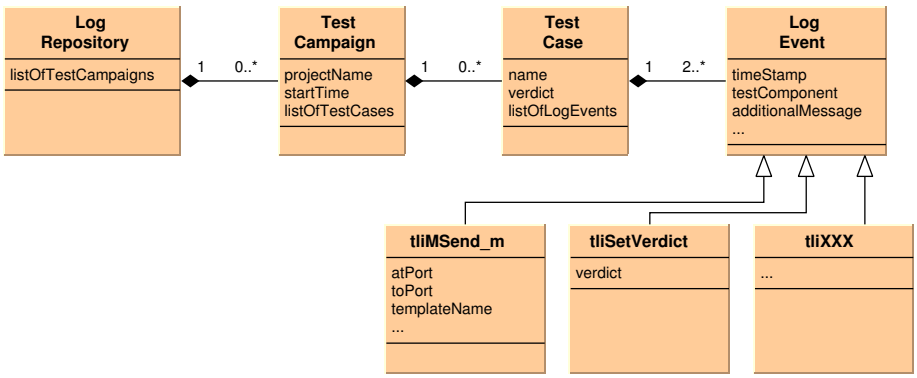


Figure 34: Conceptual UML class diagram of the data model

The Eclipse Modeling Framework [78] (EMF) is a modeling framework and code generation facility for building tools based on a structured datamodel. Exploring the capabilities of the framework could be topic of a thesis alone, and therefore only a very brief description is given here. In this particular case, an EMF model was first generated using the XML schema model importer with the schema definition from the TCI standard as input. Java code for representing the model during execution was then generated from the EMF model. Now, the runtime model (Java objects) may be populated automatically from the XML documents using the persistence facilities provided by EMF.

EMF turned out to be a major productivity boost by eliminating the tedious, mechanical work: the code generator generated the necessary Java classes for each of the 105 log events in a matter of seconds, and populating the model from the XML documents requires only a few lines of code. Obviously, the generated code requires some adjustments and supporting code to be written, but overall the code appears to be of high quality. The only drawback discovered so far has been the steep learning curve.

7.4.3 Log viewer perspective

As described in Section 7.2, a *perspective* is a collection of views. The log viewer perspective contains three major views: campaign browser, structured log and graphical log. Each view populates its contents from the data model described in the previous section. The text-based views (campaign browser and structural log) are connected to the model using the EMF Observables databinding technique. In practice, the application code essentially sets up a link that binds a specific data element from the model to a widget in the graphical user interface, and the framework handles the rest. Figure 35

illustrates the architecture of the log viewer perspective.

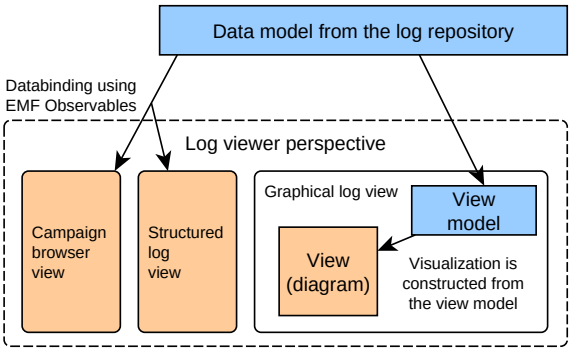


Figure 35: Log viewer perspective architecture

The views are further discussed along with a concrete example in Section 7.5. The figure also shows that the graphical presentation is constructed from a separate *view model*. The second model was introduced to avoid polluting the data model with presentation-specific details. Heer [38] captured this principle as the *Reference Model* design pattern, and mentions that based on his research it has been widely used and advocated. Aniszczyk refers to these two models as the Business model and the View model, and states that the view model may also be referred to as a *notational model* [3]. In our particular case the view model is essentially a thin layer on top of the data model. In addition to the log data, the view model stores information that is relevant only for construction of the graphical presentation, such as coordinates and relations between the graphical elements.

7.4.4 Graphical log view

The visualizer is built using the Eclipse Graphical Editing Framework (GEF) [77], which is an application-neutral toolkit for creating graphical views and editors. Currently we have no use for the editing capabilities, but GEF is also well suited for constructing read-only views. GEF is based on the Model-View-Controller (MVC) design pattern [64]. The MVC paradigm separates handling of the user input, modeling the external world and the visual feedback given to the user. Handling of these tasks is divided to three explicitly separated subsystems. The *view* manages the graphical (or textual) output and the *model* manages the behaviour and the data of the application domain. The *controller* acts as a mediator between everything else: it interprets the user input and modifies the model or the view as necessary [12].

GEF is also model-agnostic, which means that the model may have any structure and GEF does not require the use of any particular modeling framework. The controller implementation is provided to GEF as a set of EditPart objects, and the view is composed of various Figure objects. Figure 36 illustrates these concepts.

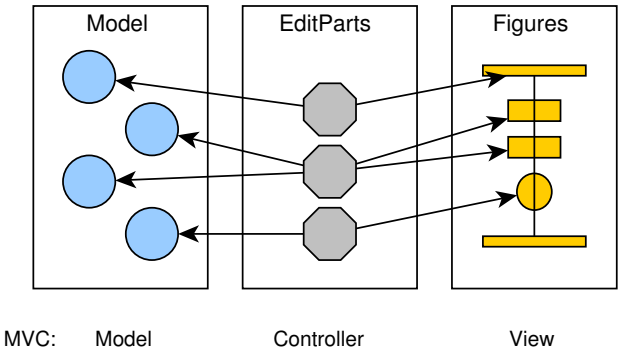


Figure 36: Eclipse Graphical Editing Framework architecture

The GEF architecture is designed so that each type of object in the model maps to a specific type of EditPart, which then constructs and controls the visual representation. In our case the GEF architecture provides a natural way to implement the rules for mapping each event to a certain symbol. Particularly, one-to-one mapping exists between a graphical symbol and an EditPart. This allows reusing the same EditPart class for log event types that are visualized using the same graphical symbol, such as an arrow or the action box. Figure 37 shows overview of the GEF architecture in the Reprise implementation.

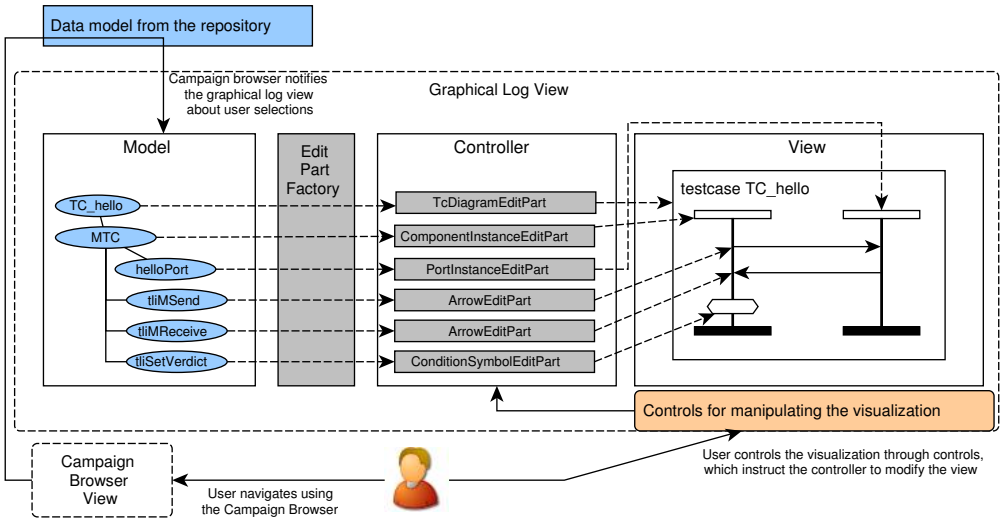


Figure 37: Visualizer architecture

The figure shows also the role of the user. Once a test case is selected in the campaign browser, the graphical log view gets notified via the Eclipse SelectionProvider framework. This notification triggers construction of the view model, which is then connected to the graphical view. The user may also interact with the graphical view by using the provided controls, which then instruct the controller to manipulate the display.

7.5 Practical example

At the time of publishing this thesis, the core functionality of the software is implemented but it is still classified as a prototype. This section describes the present implementation.

As we recall from Section 4.4, the seven fundamental tasks that the user often needs to perform are [71]: overview (1), zoom (2), filter (3), details-on-demand (4), relate (5), history (6) and extract (7). The tasks have been numbered to allow conveniently referring to them in the subsequent discussion. Tasks 1, 2, 4 and 5 have been implemented in the present version as presented below.

The default layout of the Reprise log analysis perspective is shown in Figure 38. The areas of interest have been marked with squares and labels have been assigned to them in order to allow more conveniently relating between the figure and the discussion. The screenshot shows execution log of a test case that executes five send/receive pairs against an echo server SUT. The areas of interest are:

Campaign browser: When the user starts the tool, only this view is initialized. It provides an interface to the log repository in a tree format. This is the first *overview* (task 1) of the entire data set. The folders shown at the highest level are test campaigns, and the list of test cases for the relevant campaign is shown when the folder is expanded. In this example, the testcase `TC_VisDemo_5` has been selected for analysis. The details-on-demand principle (task 4) has been applied here: upon selection of a test case from the list, the other views will populate their contents with events from the selected test case.

Structured log: This view displays the log events of the selected test case in textual format. The term *structured* is used to indicate that the data elements are separated from each other in the cells of the table. This kind of presentation may be further sorted or

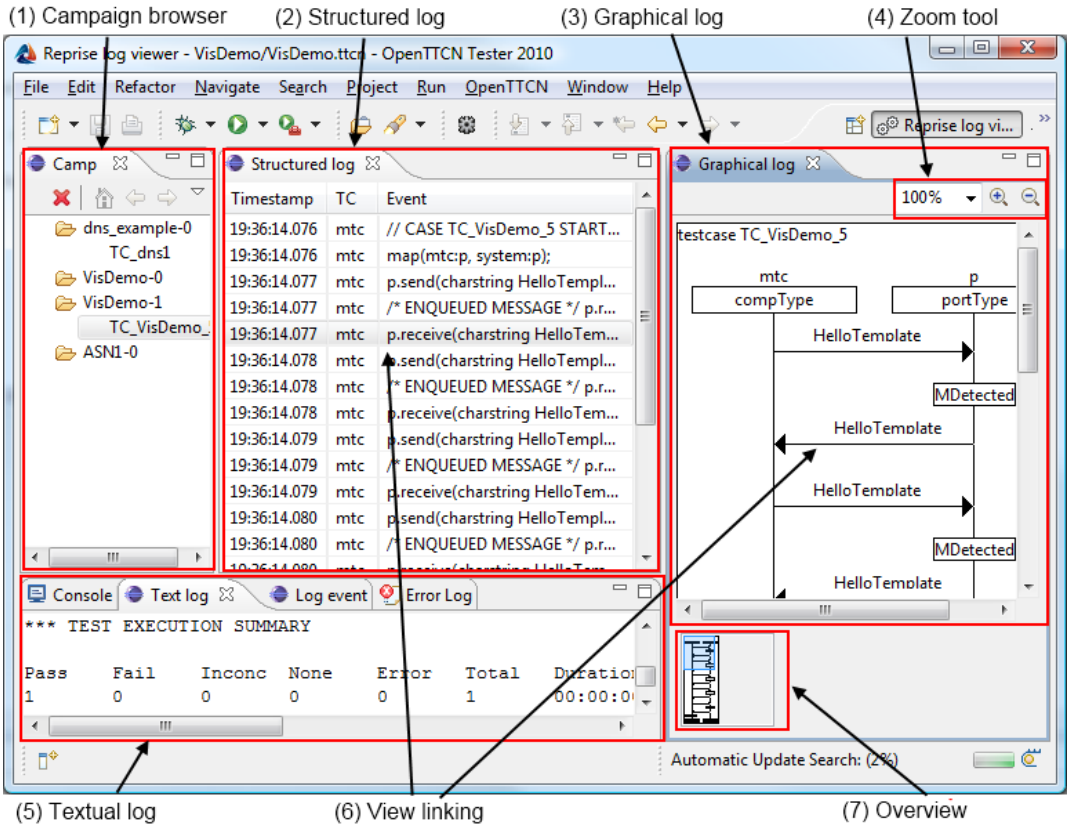


Figure 38: Screenshot of Reprise in action

filtered more conveniently than traditional plain text.

Graphical log: Shows graphical presentation of the log for the selected test case. This view implements the mapping from the log events to the GFT representation, as described in Chapter 6.

Zoom tool: Allows *zooming* (task 2) the graphical presentation. The user may either zoom in or out using the two buttons shown on the right side, select one of the predefined percentages from the dropdown menu or enter a custom zoom percentage. Additionally, the menu contains items for scaling the diagram so that the width, the height, or both of them match the available display area.

Textual log: Log in plaintext format. Various arguments for keeping the traditional log can be made: it preserves backwards-compatibility, it is easy to manipulate and easy to copy around.

View linking: Each symbol in the diagram represents a single event. For each event

drawn in the diagram, a matching textual presentation exists in the structured log. Whenever the user clicks on an event in the graphical view, the same event is highlighted in the structured log view. This allows the user to easily *relate* (task 5) between the different presentation formats.

Overview: This thumbnail view shows an overview of the entire diagram. User may rapidly navigate inside the diagram by moving the highlighted area with mouse. This is also an application of the overview task.

The filter, history and extract tasks have not been implemented in the present version. Regardless, at least filtering is a feature that will be certainly needed. The user must be able to filter out uninteresting test cases and events – for example based on the test case verdict, event type, or the test component that emitted the event. Implementation of the filtering support is expected to be straightforward using the EMF model query component, which provides a SQL-like interface to the data model.

The extract task refers to the ability to first locate the interesting subset of the data and then extract either the data or the query parameters in a convenient format. Arguments for implementing this feature could certainly be made, for example to support cooperation with other team members, but no decisions or detailed plans have been made yet.

Finally, the history task refers to keeping a history of user actions to support undo, replay and progressive refinement. For this particular tool, the history is arguably the least relevant of the seven tasks, and no implementation plans have been made yet.

8 CONCLUSIONS AND FUTURE WORK

The objective of this study was to develop a method for graphical representation of test execution logs generated by a TTCN-3 test system. The chosen approach was to utilize the relevant building blocks provided by the TTCN-3 standard suite: the Test Logging Interface (TL), the Core Language (CL) and the Graphical Presentation Format (GFT). TL defines the log events that may be emitted during test execution, CL is the source code representation and GFT specifies graphical symbols for describing test behaviour. An explicit, bidirectional mapping between CL and GFT is given, but no relationship between TL and GFT is specified.

Therefore, a mapping from TL to GFT had to be established. Based on a brief review of the TL interface it became obvious that majority of the events simply indicate execution of a specific TTCN-3 statement. As a consequence, the graphical symbol defined for the respective statement was adopted for representing the log event. First step was thus mechanical and straightforward: evalute each of the 105 events and determine whether this kind of correlation between these three exist. In other words, the first step filtered out the trivial cases.

The remaining events were further analysed and three groups relevant to either log analysis or the visualization tool implementation were identified. The events that describe the control flow in the TTCN-3 program were considered to be important for implementation of the visualization tool, but to be slightly less important in the log analysis. Conversely, the events that indicate insertion of something to the input queue of a port were considered useful, specifically in scenarios where the test suite fails to handle all input. Similarly, the events that indicate a mismatch – a scenario where the received value does not match the expected value – were considered to be useful in analysing faulty behaviour of the system under test. Finally, some of the events, such as indications of accessing the value of a module parameter or the local verdict, were considered to be irrelevant for both purposes.

This thesis also described the Reprise log analysis and visualization tool, which implements the developed technique in practice. The tool receives the TTCN-3 log data in the standardized XML format and presents it to the user using the facilities of the Eclipse Platform. Particularly, the Eclipse Modeling Framework and the Eclipse Graphical Editing framework are utilized extensively. The tool is also integrated with the OpenTTCN Tester toolchain.

In terms of future work, various possible directions can be seen immediately. First of all, this work made a conscious effort to utilize the TTCN-3 standard suite as extensively as possible. In a sense, this principle was followed blindly without questioning its merits. At first sight, it appears to have the obvious benefit of giving the user a common environment and presentation format for both specifying the tests and analysing the results. An argument could be made to question the benefits of this approach altogether and look into other types of log presentation techniques.

Also, the approach taken in this work was to focus on graphical presentation of test *behaviour*, with the data presentation limited to displaying the abstract template names. Data analysis is certainly an important task during the log analysis, but effective data presentation is also highly dependant on the application domain. For some protocols a 8-bit unsigned entity is naturally a sequence of bits while sometimes the integer representation is more meaningful. Consequently, adding data presentation along with support for domain-specific encodings would certainly be useful.

Finally, there is always further work to be done in the area of tool support. Integrated Development Environments aim at enhancing productivity, and practically speaking there is no end to the potential improvements that facilitate this goal. For instance, support for handling large diagrams and diagram references conveniently is necessary. Simply drawing all the symbols in a large static diagram is most likely not sufficient, and intuitive MSC presentation techniques have been already suggested in various papers. HyperMSC [41, 6] is a method for visualizing a referenced MSC diagram inline inside the reference symbol. Eick [19] discusses the use of colors and various linked views for MSC presentation. Sedlmair [69] describes an interactive MSC presentation that supports various tasks for manipulating the display inside a single view. Also, adopting abstraction techniques from related fields [36, 15, 49] could be considered.

The environment also has a major role in selection of the optimal tool implementation strategy. An interactive technique that is effective with plenty of screen real estate available will most certainly not be the most optimal in a more limited environment. Mobile devices are getting increasingly more popular – who knows when the users start to demand test results on their iPhone™?

References

[1] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, Verification, and Testing of Computer Software. *ACM Comput. Surv.*, 14(2):159–192, 1982.

[2] S. Ahokas. Developing a .NET Framework Software Development Kit for a TTCN-3 tool. Bachelor's thesis, Lappeenranta University of Technology, 2007.

[3] C. Aniszczyk and R. Hudson. Create an Eclipse-based application using the Graphical Editing Framework. Available: `http://www.ibm.com/developerworks/library/os-eclipse-gef11`. Visited 13-May-2010.

[4] P. Baker, D. Evans, J. Grabowski, H. Neukirchen, and B. Zeiss. TRex - the Refactoring and Metrics Tool for TTCN-3 test specifications. In *TAIC-PART '06: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 90–94, Washington, DC, USA, 2006. IEEE Computer Society.

[5] P. Baker, J. Grabowski, E. Rudolph, and I. Schieferdecker. Schieferdecker: A Message Sequence Chart-profile for graphical test specification, development and tracing. In *18th International Conference and Exposition on Testing Computer Software, Theme: Meeting the New Challenges of Testing*, pages 18–22, 2001.

[6] P. Baker, E. Rudolph, and I. Schieferdecker. Graphical Test Specification - The Graphical Format of TTCN-3. *Lecture notes in computer science*, pages 148–167, 2001.

[7] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[8] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[9] G. V. Bochmann and A. Petrenko. Protocol testing: review of methods and relevance for software testing. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 109–124, New York, NY, USA, 1994. ACM.

[10] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: a flexible environment for computer systems visualization. *ACM SIGGRAPH Computer Graphics*, 34(1):73, 2000.

[11] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), Oct. 1989. Updated by RFCs 1349, 4379.

[12] S. Burbeck. Applications Programming in Smalltalk-80: How to use Model-View-Controller (MVC), 1992.

[13] S. K. Card, J. D. Mackinlay, and B. Shneiderman. Using vision to think. In *Readings in information visualization: using vision to think*, pages 579–581. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[14] G. F. Coulouris and J. Dollimore. *Distributed systems: concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[15] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *COOTS'98: Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems*, pages 16–16, Berkeley, CA, USA, 1998. USENIX Association.

[16] J. des Rivières and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[17] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[18] G. Din, J. Zander, and S. Pietsch. Test execution logging and visualization techniques. In *17th International Conference Software and Systems Engineering and their Applications*, Paris, France, 2004.

[19] S. G. Eick and A. Wards. An interactive visualization for Message Sequence Charts. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 2, Washington, DC, USA, 1996. IEEE Computer Society.

[20] ETSI ES 201 873-1 V4.1.1 (2009-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute, 2009.

[21] ETSI ES 201 873-3 V3.2.1 (2007-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular Presentation Format (TFT). European Telecommunications Standards Institute, 2007.

[22] ETSI ES 201 873-3 V3.2.1 (2007-02). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT). European Telecommunications Standards Institute, 2007.

[23] ETSI ES 201 873-5 V4.1.1 (2009-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI). European Telecommunications Standards Institute, 2009.

[24] ETSI ES 201 873-6 V4.1.1 (2009-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI). European Telecommunications Standards Institute, 2009.

[25] ETSI TS 102 027. Methods for Testing and Specification (MTS); Conformance Test Specification for SIP (IETF RFC 3261). 3-part deliverable. European Telecommunications Standards Institute, 2008.

[26] European Telecommunications Standards Institute. TTCN-3 Standards Suite. Available: `http://www.ttcn-3.org/StandardSuite.htm`. Visited 7-Feb-2010.

[27] J.-D. Fekete, J. J. Wijk, J. T. Stasko, and C. North. The value of information visualization. In *Information Visualization: Human-Centered Issues and Perspectives*, Lecture Notes In Computer Science, pages 1–18. Springer-Verlag, Berlin, Heidelberg, 2008.

[28] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 2004.

[29] G. W. Furnas and B. B. Bederson. Space-scale diagrams: understanding multiscale interfaces. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 234–241. ACM Press/Addison-Wesley Publishing Co., 1995.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[31] R. Gecse and S. Dibuz. An intuitive TTCN-3 Data Presentation Format. In *Testing of Communication Systems*, volume 2644 of *Lecture Notes in Computer Science*, pages 63–78, 2003.

[32] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 31(6):687–695, 1988.

[33] J. Grabowski. TTCN-3 - a new test specification language for black-box testing of distributed systems. In *Proceedings of the 17th International Conference and Exposition on Testing Computer Software (TCS'2000), Theme : Testing Technology vs.Testers' Requirements, Washington D.C*, 2000.

[34] J. Grabowski and D. Hogrefe. Towards the third edition of TTCN. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems*, pages 19–30, Deventer, The Netherlands, 1999. Kluwer, B.V.

[35] D. Grisby. omniORB – Free High Performance ORB. Available: `http://omniorb.sourceforge.net`. Visited 10-May-2010.

[36] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.

[37] R. V. L. Hartley. Transmission of Information. *Bell System Technical Journal*, 7:535–563, 1928.

[38] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):853–860, 2006.

[39] D. P. Helmbold, C. E. McDowell, and J. Wang. TraceViewer: A graphical browser for trace analysis. Technical report, University of California, Santa Cruz, CA, USA, 1990.

[40] IEEE Std 1012-2004. *IEEE Standard for Software Verification and Validation*. IEEE Computer Society, 2004.

[41] E. R. Ina, I. Schieferdecker, and J. Grabowski. HyperMSC – a graphical representation of TTCN. In *Proceedings of the 2nd Workshop of the SDL Forum, Society on SDL and MSC (SAM2000)*, pages 2–8, 2000.

[42] ISO/IEC 7498-1:1994. *Information technology. Open Systems Interconnection. Basic Reference Model: The basic model.* International Organization for Standardization, Geneva, Switzerland, 1994.

[43] ISO/IEC 9646:1994. *Information technology. Open Systems Interconnection. Conformance Testing Methodology and Framework.* International Organization for Standardization, Geneva, Switzerland, 1994.

[44] ISO/IEC 9646:1994. *Information technology. Open Systems Interconnection. Conformance Testing Methodology and Framework. Part 2: Abstract Test Suite Specification.* International Organization for Standardization, Geneva, Switzerland, 1994.

[45] ISO/IEC JTC1/SC7 Working Group 26. ISO/ICE 29119 Software Testing Standard. Available: `http://softwaretestingstandard.org`. Visited 26-Jan-2010.

[46] ISO/IEC JTC1/SC7 Working Group 26. ISO/IEC 29119 Software Testing - Part 2 - Test Process. Draft, September 2009.

[47] ITU-T. Evolution of TTCN. Available: `http://www.itu.int/ITU-T/studygroups/com17/ttcn.html`. Visited 7-Feb-2010.

[48] ITU-T. ITU-T Recommendation Z.120 (04/2004): Message Sequence Chart (MSC). International Telecommunication Union, 2004.

[49] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 360–370, New York, NY, USA, 1997. ACM.

[50] Kamailio. The Open Source SIP Server. Available: `http://www.kamailio.org`. Visited 04-Feb-2010.

[51] K. G. Knightson, editor. *OSI protocol conformance testing: IS 9646 explained*. McGraw-Hill, Inc., New York, NY, USA, 1993.

[52] J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–100, 1987.

[53] S. Mauw and V. Bos. Drawing Message Sequence Charts with LaTeX. *TUGBoat*, 22(1-2):87–92, March/June 2001.

[54] G. J. Myers. *The Art of Software Testing, Second Edition*. Wiley, June 2004.

[55] H. Nyquist. Certain Factors Affecting Telegraph Speed. *Bell System Technical Journal*, page 324, 1924.

[56] Object Management Group. Event Service Specification. Revision 1.2, October 2004. Available: `http://www.omg.org/technology/documents/formal/event_service.htm`. Visited 2-May-2010.

[57] Object Management Group. The Common Object Request Broker Architecture: Architecture and Specification. Revision 2.6, December 2001. Available: `http://www.omg.org/spec/CORBA/2.6`. Visited 2-May-2010.

[58] OpenTTCN Ltd. OpenTTCN Tester. Available: `http://www.openttcn.com/products`. Visited 2-May-2010.

[59] Oxford University Press. Compact Oxford English Dictionary. Available: `http://www.askoxford.com`. Visited 24-Apr-2010.

[60] K. Perlin and D. Fox. Pad: an alternative approach to the computer interface. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 57–64. ACM, 1993.

[61] S. Pook, E. Lecolinet, G. Vaysseix, E. L. G. Vaysseix, and E. Barillot. Context and interaction in zoomable user interfaces. In *In Proocedings of the 5th International Working Conference on Advanced Visual Interfaces (AVI 2000)*, pages 227–231. ACM Press, 2000.

[62] A. Rao and H. Schulzrinne. Real-world SIP Interoperability: Still an Elusive Quest. In *1st SIP Forum SIP Interoperability Workshop at IETF 70*, December 2007.

[63] D. Rayner. Future directions for protocol testing, learning the lessons from the past. In *International Workshop on Testing of Communicating Systems 10*, pages 3–10. Chapman & Hall, 1997.

[64] T. Reenskaug. Models - Views - Controllers. Technical report, Xerox PARC, 1979. Available: `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`. Visited 8-May-2010.

[65] S. Reiss. Software tools and environments. *ACM Computing Surveys (CSUR)*, 28(1):281–284, 1996.

[66] J. Rosenberg. A Hitchhiker's Guide to the Session Initiation Protocol (SIP). RFC 5411 (Informational), Feb. 2009.

[67] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630.

[68] I. Schieferdecker and J. Grabowski. Conformance Testing with TTCN. *Telektronikk (ISSN 0085-7130), 96 (4), 2000*, pages 85–95, Dec. 2000.

[69] M. Sedlmair. MSCar: Enhancing Message Sequence Charts with interactivity for analysing (automotive) communication sequences. In *Proceedings of the 2nd International Workshop on the Layout of (Software) Engineering Diagrams*, September 2008.

[70] C. Shannon, N. Petigara, and S. Seshasai. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 1948.

[71] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Visual Languages*, number UMCP-CSD CS-TR-3665, pages 336–343, College Park, Maryland 20742, U.S.A., 1996.

[72] W. Stallings. *Data and computer communications (5th ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[73] Sun Microsystems. Java Native Interface. Available: `http://java.sun.com/j2se/1.5.0/docs/guide/jni`. Visited 10-May-2010.

[74] A. Tanenbaum. *Computer Networks 4th edition*. Prentice Hall Professional Technical Reference, 2003.

[75] D. Tang, C. Stolte, and R. Bosch. Design choices when architecting visualizations. In *Proceedings of InfoVis 2003, IEEE Symposium on Information Visualization (INFOVIS'03)*, pages 41–48. IEEE Computer Society, 2003.

[76] The Eclipse Foundation. Eclipse. Available: `http://www.eclipse.org`. Visited 17-Jan-2010.

[77] The Eclipse Foundation. Eclipse Graphical Editing Framework. Available: `http://www.eclipse.org/gef/`. Visited 13-May-2010.

[78] The Eclipse Foundation. Eclipse Modeling Framework. Available: `http://www.eclipse.org/modeling/emf`. Visited 13-May-2010.

[79] M. Tory and T. Möller. Human factors in visualization research. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):72–84, 2004.

[80] TRex - the TTCN-3 Refactoring and Metrics Tool. Available: `http://www.trex.informatik.uni-goettingen.de`. Visited 17-Jan-2010.

[81] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. *SIGPLAN Not.*, 33(10):271–283, 1998.

[82] J. J. v. Wijk. Bridging the gaps. *IEEE Computer Graphics and Applications*, 26(6):6–9, 2006.

[83] C. Willcock, T. Deib, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3*. Wiley, 2005.

[84] World Wide Web Consortium. Extensible Markup Language (XML) Specification. W3C Recommendation, February 1998. Available: `http://www.w3.org/XML`. Visited 13-May-2010.

[85] World Wide Web Consortium. Scalable Vector Graphics (SVG) 1.1 Specification. W3C Recommendation, January 2003. Available: `http://www.w3.org/XML/`. Visited 13-May-2010.

[86] World Wide Web Consortium. XML Schema 1.0 Specification. W3C Recommendation, May 2001. Available: `http://www.w3.org/XML/Schema`. Visited 13-May-2010.

[87] J. S. Yi, Y. a. Kang, J. Stasko, and J. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 2007.

[88] H. Zimmermann. OSI reference model – The ISO model of architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.