A Framework for the Specification of Test Cases for Real-Time Distributed Systems

Thomas Walter[a] and Jens Grabowski[b]

[a]Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zürich, Gloriastrasse 35, CH-8092 Zürich, Switzerland, phone (+41 1) 632 7007, fax (+41 1) 632 1035, walter@tik.ee.ethz.ch, http://www.tik.ee.ethz.ch

[b]Institute for Telematics (ITM), Medical University of Lübeck, Ratzeburger Allee 160, D-23538 Lübeck, Germany, phone (+49 451) 500 3723, fax (+49 451) 500 3722, jens@itm.mu-luebeck.de, http://www.itm.mu-luebeck.de

Abstract

The *OSI conformance testing methodology and framework* (CTMF) is a well established standard which defines and regulates the conformance testing procedure for protocol implementations. Conformance testing is meant to be functional black-box testing. Besides concepts and terminology, CTMF standardizes testing architectures and the *Tree and Tabular Combined Notation* (TTCN) as the test specification language.

Since more and more distributed systems such as multimedia, safety-critical and real-time systems rely on the timely availability of information, testing of real-time requirements becomes a serious issue, too. Unfortunately, testing real-time and other non-functional requirements (performance and reliability) are outside the scope of CTMF.

In this paper we present an extension of CTMF which allows to specify test cases for testing real-time requirements. The extension includes a generic testing architecture and a notation for test specification which is called real-time TTCN.

Keywords: Software testing, conformance testing, real-time, real-time distributed systems, testing architectures, test notations, real-time distributed systems testing.

1. Introduction

In the past, distributed systems provided best-effort communication services. Best-effort means that an application sends data whenever it wants to, in any quantity and without asking or telling the network first, and the network delivers data if it can with no guarantee of reliability, delay bound, throughput, jitter control etc. This kind of service provision is, however, not sufficient for new classes of real-time distributed systems like multimedia and safety-critical systems, e.g., videoconferencing systems or process control systems. For such systems to work correctly, quality-of-service guarantees are essential, and communication tasks have to be performed in real-time.

The growing complexity of these systems makes the formal verification of real-time properties impossible (or at least intractable). In such situations, testing is a feasible approach to get

confidence that a distributed system behaves correctly in all respects, including the real-time behaviour even under various situations.

Testing means that expected outcomes as prescribed by the specification of the software program are compared with observed outcomes as produced by runs of the software program [7, 20,37,45]. If expected and observed outcomes differ, a fault has been discovered. A pair of inputs and expected outputs is termed a test case. Test specification is the process of identifying pairs of input and expected output. A testing methodology and framework for OSI-compliant distributed systems have been developed in the past by ISO (International Organization for Standardization) and ITU–T (International Telecommunication Union — Telecommunication Standards Sector) which is known as the conformance testing methodology and framework (CTMF) [21].

In this paper we develop a downwards compatible extension of CTMF for testing real-time distributed systems. The contributions of this paper to real-time distributed systems testing are threefold: First, classifications of distributed systems and types of testing are presented. The classification of distributed systems considers the following parameters: distribution (allows to distinguish between local and distributed systems), communication (distinguishes between synchronous and asynchronous communication as well as between unicast, multicast and broadcast communication), and real-time. In our classification of types of testing, we consider parameters such as testing objectives (testing functional and non-functional requirements), testing approaches (black-box, grey-box and white-box testing), and testing architectures (the components that make up a test system). Second, based on these classifications, we define testing architectures suitable for testing a diverse set of distributed systems with respect to different testing objectives. Our idea is to have a tool box of components that can be combined generically into a testing architecture suitable for the application or system to be tested. Third, a test notation is proposed for testing real-time aspects of distributed systems. Essentially, our approach allows the timing of actions relative to the occurrence of previous actions.

The paper is structured as follows: Section 2 gives a classification of distributed systems and distributed testing. We use this classification for the evaluation of existing testing frameworks and for identifying where our framework fits in. Section 3 presents the details of our proposed testing architecture and test specification language. Section 4 presents an application of our testing methodology. The application is in quality-of-service testing of a videoconferencing system. Before summarizing the main issues of the paper, we relate our work to research work documented in the literature (Section 5).

2. Real-time distributed systems and real-time distributed testing

Real-time distributed systems cover a wide range of systems focusing on solving different computational problems. In the literature, definitions of *embedded systems*, *reactive systems*, *real-time systems*, or *hybrid systems* can be found. Summaries of these definitions are:

- embedded system (process control system + real-time system): An embedded system is an electronic system embedded within an external process. It influences the external process to ensure that functional and performance requirements are met [28].

- reactive system: A reactive system is a system whose role is to maintain an ongoing interaction with its environment (e.g., concurrent and real-time programs, embedded systems, process control systems and operating systems) [35].

- real-time system: A real-time system is a reactive system whose interaction with its environment is constrained by timing requirements [25,27].

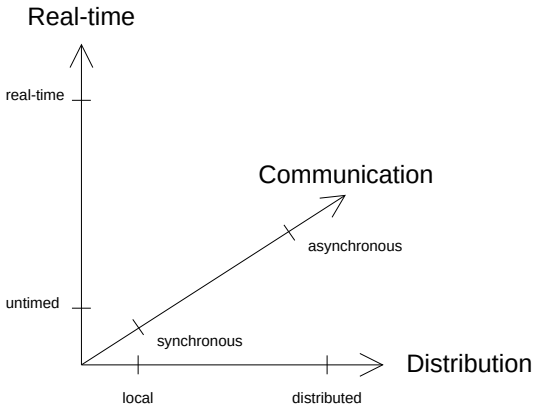


Figure 1. Classification of Distributed Systems

- hybrid system: A hybrid system is a reactive system that has discrete and continuous components (e.g., a digital controller that interacts with a continuously changing physical environment) [38].

The discussions on real-time distributed systems and their testing often try to cope with all kinds of systems mentioned above, although their requirements concerning reliability, security, performance, delay, etc. are different and, thus, their testing requires different methods. As a result there are often misunderstandings and ambiguities when talking about real-time distributed systems and real-time distributed testing. To avoid these problems, the following sections provide a classification of these terms.

2.1. Real-time distributed systems

A *distributed system* is a system of cooperating processes running on computers which are communicating. A *computer* consists of input and output units, memory, and a processor [39]. A *process* is a computer program in execution, i.e., it is executed by the processor following the instructions of the program. The *communication* among processes may be realized by means like shared memory, or a physical communication network between computers.

Based on these considerations, distributed systems can be classified according to the geographic distribution of computers and the methods used for interprocess communication. For coping with systems whose correct behaviour is constrained by real-time requirements, a third dimension is necessary. This classification scheme is shown in Figure 1. Any combination of properties given by the three dimensions determine a feasible instantiation of a distributed system.

2.1.1. Distribution

The aspect of distribution in Figure 1 allows to distinguish between local systems and really distributed systems. On the one extreme, a distributed system might be realized locally on a single workstation handling several communicating processes on a single processor. On the other extreme, a distributed system might be highly distributed like the global telephone network or the Internet.

2.1.2. Communication

We distinguish between *synchronous* and *asynchronous* communication (Figure 1). During communication the involved processes may send, receive, or send and receive information. A

further characteristic is the number of processes involved in the communication.

Synchronous communication means that the communicating processes have *to meet* for the exchange of information, i.e., processes are blocked waiting for their communicating peers. During synchronous communication, the processes may send and receive information.

Communicating asynchronously means that a sender process sends its information without waiting for the receiving process(es) to become ready. This means that only a receiving process may be blocked, if it waits for specific information not yet available.

In most cases we only think of two cooperating processes performing communication tasks. In reality we sometimes find situations, where there exist one sender process and several receiver processes. For these cases we may distinguish between unicast, multicast and broadcast communication.

Three examples may show the usage of the different communication paradigms: (1) the remote procedure call in CORBA environments [40] is a synchronous communication between two partners, (2) sending and receiving e-mails is asynchronous communication, and (3) television is an example for asynchronous communication with one sender process, e.g., a television station, and several thousands of receiver processes, e.g., TV decoders.

2.1.3. Real-time

The real-time dimension in Figure 1 allows to classify systems according to their real-time requirements. On one hand, we have systems with no real-time requirements, i.e., untimed systems. On the other hand, we have systems were the the fulfilment of hard real-time bounds is essential for the correct behaviour of the entire system. Within these extremes there exist systems which have to fulfil soft real-time requirements, i.e., the fulfilment of real-time requirements has to be evaluated by using statistical metrics.

Three examples may clarify the character of the different system types:

1. A typical timeless system is the Internet e-mail system. Sending and receiving an e-mail are not constrained by timing requirements. Even the reception of an e-mail by an e-mail server may happen at any time after sending the e-mail.

2. Multimedia systems are examples of systems which have to fulfill soft real-time requirements. The timely reproduction of audio and video streams implies that communication between source and sink obeys a maximum end-to-end delay. However, up to a certain degree, the loss of video or audio information, in case of an unacceptably long delay of audio and video data, may be tolerated.

3. An example of a crucial real-time distributed system is a flight control system. All flight data of all aircrafts have to be communicated and updated in real-time. If data is lost or delayed this may have catastrophic consequences.

2.2. Real-time distributed testing

Testing means that an Implementation Under Test (IUT) is stimulated and the observed responses are compared with the expected responses as prescribed by a specification. If expected and observed responses differ then a problem has been discovered. The IUT may either be faulty, or behave in a nondeterministic manner.

IUTs can be seen as (possibly infinite) state machines which have an actual state, take an input, perform some computations, produce outputs and go into the next state.

Validating the response of an IUT to all inputs in all states is often impossible, since for realistic problems, the number of state/input pairs is generally infinite. Therefore, an IUT can only be tested against a selected and finite subset of state/input pairs. The general assumption

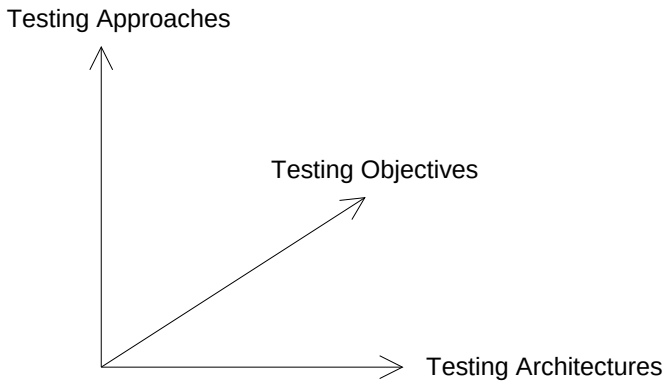


Figure 2. Classification of different types of testing

is, that if the IUT behaves correctly for a subset of all state/input pairs, then the IUT behaves correctly for all state/input pairs.

A sequence of stimuli and expected responses as prescribed by the specification of the IUT is termed *test case*. Thus, test case specification is the process of defining test cases, i.e., sequences of stimuli and expected responses. Please note that a test case may check more than one of the state/input pairs described above. Normally, for each test case a *test purpose* description is given explaining the goal of the test case in an informal manner.

Based on this terminology, a three dimensional classification of different types of testing is given. The dimensions describe testing objectives, testing approaches and testing architectures (Figure 2).

2.2.1. Testing objectives

The testing objectives axis in Figure 2 describes a very general dimension. We distinguish function and non-functional objectives.

Functional testing objectives refer to the correct functional behaviour of an IUT, i.e., the correct input/output behaviour in certain states. Non-functional objectives refer to timing constraints in general, reliability, robustness, and organizational aspects (like usability, configuration management etc.).

The scope of this paper is restricted to the correct input/output behaviour of functional testing objectives and to timing constraints of non-functional testing objectives. For timing constraints we restrict ourselves to real-time testing.

2.2.2. Testing approaches

Within the testing approaches dimension different possibilities to specify test cases are considered. The two main approaches for test case specification are *black-box testing*[1] and *glass-box testing*[2].

In the black-box testing approach, test cases are derived based on the knowledge about the specification. The internal structure of the IUT remains hidden, i.e., it is a black box for the test specifier.

The glass-box testing approach is concerned primarily with the structure of the IUT, i.e., the program code. The execution of test cases should cover the program code of the IUT. Test cases are derived with the program code at hand and test sequences of inputs and outputs are

[1] Black-box testing is also known as *specification-based testing*.
[2] Glass-box testing is also known as *white-box testing* or *structural testing*.

determined by analysing the code.

In between black-box and white-box testing there exists a third approach called *grey-box testing*. For the grey-box testing approach, test cases are determined by analysing the specification, but, some internals of the IUT are observable and the observations can be used in test case specifications. The observable internals may, for example, be state variables or message queues.

2.2.3. Testing architectures

The term *testing architecture* refers to the test devices, their interconnections and the connections between test devices and IUT. For testing a distributed system, the testing architecture forms a distributed system itself. Thus, in general, testing architectures can be classified according to the scheme presented in Section 2.1.

However, in Figure 2 this is represented by only one dimension. It refers to the distribution of the test equipment. It should be noted that a distributed IUT may be tested by a local test architecture, or a local IUT may be tested by a distributed test architecture.

2.3. Assessment of our work

We understand our work as an extension of the well established *OSI Conformance Testing Methodology and Framework* (CTMF), which is developed and standardized by ISO and ITU-T [21]. In the following, we relate CTMF to the classifications presented in the previous sections and explain our extensions of CTMF.

2.3.1. Conformance Testing Methodology and Framework

CTMF defines a comprehensive procedure for the conformance testing of OSI protocol entity implementations. The entire standard consists of seven parts and covers the following aspects: concepts (part 1), test suite specification and test system architectures (part 2), test notation (part 3), test realization (part 4), means of testing and organizational aspects (part 5,6,7).

CTMF target systems: By definition the target systems to be tested according to the CTMF principles are implementations of OSI protocol entities. OSI protocol entities are only part of distributed systems, but they are no real distributed systems themselves. However, the principles of CTMF, especially the parts 1, 2 and 3 of CTMF, have also been applied successfully to real distributed systems (see e.g., [4,10,44]). Therefore, in the distribution dimension of our classification (Figure 1) the target systems of CTMF belong to the local systems, but in practice, the range of CTMF applications covers also distributed systems.

The communication mechanism used for the exchange of information between peer protocol entities is asynchronous message exchange. Therefore, in the communication dimension of Figure 1 the target systems of CTMF belong to the asynchronously communicating systems. There have been attempts to apply CTMF to systems supporting other communication mechanisms [46], but these attempts were proprietary and cannot be generalized.

The third dimension of our classification is the real-time dimension. It denotes the importance of real-time requirements to be fulfilled for the correct behaviour of the system. OSI protocol entities can be seen as timeless systems. There exist some upper bounds for response and waiting times, but they do not describe hard real-time bounds. Extending CTMF to be used for systems with performance and real-time requirements is a research area to which this paper defines a contribution.

Classification of Conformance Testing: CTMF has also to be classified according to the testing classification described in Section 2.2 and visualized in Figure 2.

By definition conformance testing is functional testing. In the introduction of part 1 of CTMF [22] the assessment of performance, robustness or reliability of an implementation is explicitly

excluded from the scope of CTMF. In parts 5–7 CTMF also covers some organizational aspects, but for our work the focus on functional testing of CTMF in the testing objectives dimension (Figure 2) is important.

In the testing approaches dimension, CTMF follows the black-box testing approach. It is stated in [22] that *'although aspects of both internal and external behaviour are described in OSI International Standards and CCITT Recommendations, it is only the requirements on external behaviour that have to be met by real open systems'*.[3]

Describing the testing architectures dimension of CTMF is the most difficult part of the classification. CTMF recommends to test OSI protocol entities on the lower layer side via an underlying service provider. For doing this, CTMF defines four basic testing architectures called *local*, *distributed*, *coordinated* and *remote test method*.[4] These architectures mainly differ in the distribution of the access points to the IUT, called *Points of Control and Observation* (PCOs) in CTMF, and the possibilities to control and observe the IUT. For cases where protocol entities are able to communicate with more than one peer entity, CTMF provides the possibility to use the *multi-party context*, i.e., the basic test architectures are combined to complex architectures, forming themselves complex distributed systems. In summary, in the testing architecture dimension of our classification of types of testing (Figure 2), CTMF provides a wide range of testing architectures. Putting them into our classification of distributed systems, CTMF testing architectures follow the asynchronous communication approach, are timeless systems, and cover all of the distribution dimension.

2.3.2. CTMF extensions

Our work concentrates on extensions of parts 1, 2 and 3 of CTMF. In particular, we define extended test architectures (Section 3.1) and an extended test notation for real-time test specification (Section 3.3). Based on the classification of the CTMF target systems and CTMF itself it is possible to classify our work in relation to CTMF.

Target systems: The real-time distributed systems for which we develop a test methodology extend the CTMF target systems in all three dimensions (Figure 1). In the distribution dimension, it is allowed to apply our methodology to really distributed systems, i.e., we allow to have several IUTs to be tested. In the communication dimension, we intend to broaden the scope of CTMF towards systems using additional communication mechanisms. On the testing architecture side, this is already considered (Section 3.1), but for the test notation the extensions for handling synchronous communication are under study. In the real time dimension our methodology broadens the scope of CTMF to systems which have to fulfill hard real-time requirements. For the handling of soft real-time requirements some proposals exist [47]. They cover some aspects of performance test specification, but to our knowledge a formal basis is missing and aspects of performance testing are not covered.

Test methodology: Our methodology can also be classified according to the classification shown in Figure 2. For the testing approaches dimension we concentrate on black-box testing, although our generic test architecture presented in Section 3.1 allows to identify monitoring points which might be usable for grey-box testing. However, for test specification we do not support monitoring, i.e., grey-box testing. In the testing objectives dimension besides pure functional testing also real-time testing is supported. In the testing architectures dimension our

[3]The CCITT was renamed into ITU-T after the publication of CTMF.

[4]CTMF uses the term *abstract test methods* instead of test architectures. In the following, we will only use the term *test architecture* to avoid confusions with the terms *methods* and *methodology*.

methodology broadens the scope of CTMF from the range of timeless distributed systems towards real-time distributed systems, because a test architecture for testing real-time distributed systems is itself a real-time distributed system.

3. A framework for real-time test case specification

Following the CTMF approach, the methodology to be developed consists of a generic test architecture and a test language. The test architecture describes static aspects of a test case in terms of test components, interfaces and communication channels which are combined to a test system. The test language adds dynamic aspects in terms of behaviour descriptions which are assigned to test components. In the proposed methodology, behaviour descriptions cover functional and non-functional behaviour of a system. Some initial steps defining a generic test architecture [55] and studying real-time and performance extensions of TTCN [47,51] have been proposed.

3.1. Generic test architecture

We present a generic test architecture [55] that extends CTMF along the testing architectures dimension and, but only limited, along the testing approaches dimension. For the latter, we add grey-box testing capabilities. Our idea is to have a tool box of components that can be combined generically into a test architecture which suits a specific application or system to be tested best. A test architecture comprises possibly several instances of different types of components. These types are:

- An *Implementation Under Test* (IUT) represents the implementation or parts of the distributed system to be tested. In principle, an IUT may be distributed over physically separate real systems.
- An *Interface Component* (IC) is a component which is needed for interfacing the IUTs, e.g., an underlying service or a system in which an IUT is embedded.
- A *Test Component* (TC) is a component which contributes to the test verdict by coordinating other TCs or controlling and observing IUTs. A test architecture identifies all TCs necessary for the execution of a specific test case. A TC exists from the start of a test case or is created dynamically by other TCs. In each test architecture, there should be one special *Main Test Component* (MTC) which starts, ends and coordinates the test run.
- A *Controlled Component* (CC) is a component which does not contribute to the test verdict but provides SUT specific data to TCs or the SUT, e.g., a load generator, an emulator or a simulator.
- A *Communication Point* (CoP) represents a point at which communication takes place and can be observed, controlled and monitored. CoPs denote communication points between all types of components, including communication between IUT components. For the latter case CoPs may be placed somewhere in the IUT, thus they may be used for controlling and observing state information internal to the IUT or to monitor communication between IUT components.
- A *Communication Link* (CL) is a means for describing possible communication flows between TCs, IUTs, ICs and CCs and the kind of communication which may take place. For CLs we distinguish between active and passive CLs. An active CL can be characterized by its kind of communication (synchronous or asynchronous) and its direction of data flow

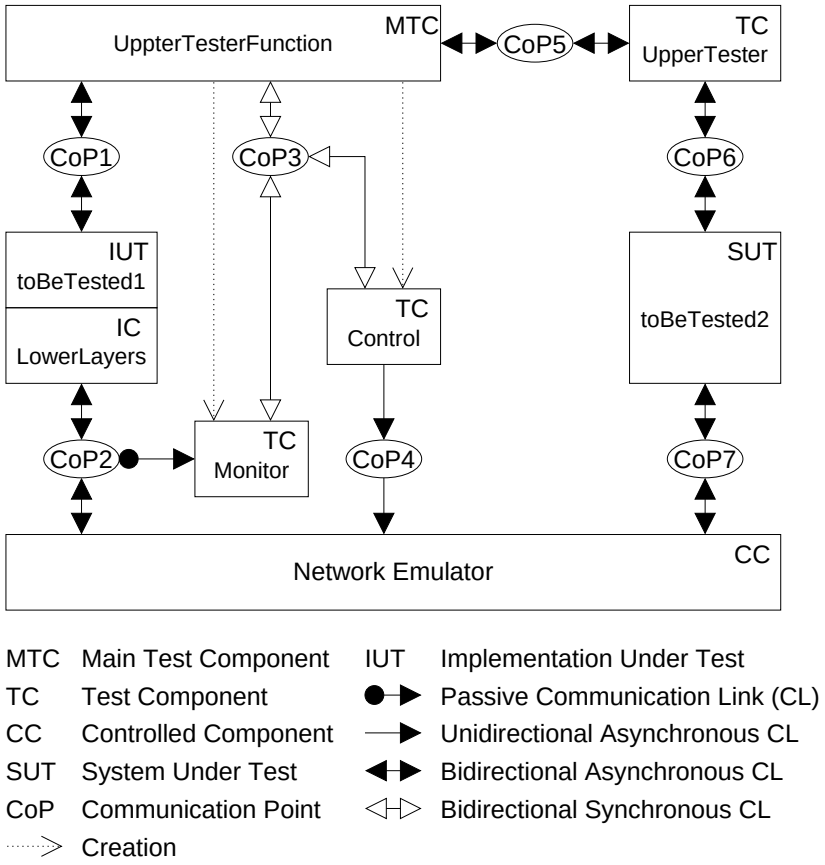


Figure 3. Test architecture for interoperability testing

(unidirectional or bidirectional). A passive CL allows to monitor communication, i.e., to listen at a CoP. If test components are dynamically created, then CLs are also dynamically created, too. CoPs linked to CLs are all known before test execution and are defined in the test configuration. This implies that CoPs cannot be created dynamically.

- The term *System Under Test* (SUT) denotes a combination of ICs and IUTs.

An example test configuration is shown in Figure 3. The different hard- and software components of the architecture are shown as boxes and ellipses. The communication flow, the kind of communication and the creation of TCs is indicated by different types of arrows.

Figure 3 describes an architecture proposed for interoperability testing in [55]. There are two IUTs to be tested. One is called *toBeTested1*; the other is embedded in the SUT *toBeTested2*. The IUTs communicate by using an underlying network which in our case is emulated by the CC *Network Emulator*. The IUT *toBeTested1* needs the IC *Lower Layers* for having the interface *CoP2* with CC *Network Emulator*. The communication at *CoP2* is monitored by TC *Monitor*. This is described by the passive CL between *CoP2* and *Monitor*. If necessary, the CC *Network Emulator* can be controlled by the TC *Control*.

The MTC is called *UpperTesterFunction*. It communicates asynchronously via *CoP5* with the peer TC *UpperTester*. As indicated by the dotted arrow, the TCs *Monitor* and *Control* are created by the MTC. It is assumed that they are running on the same computer and perform a synchronous communication with the MTC.

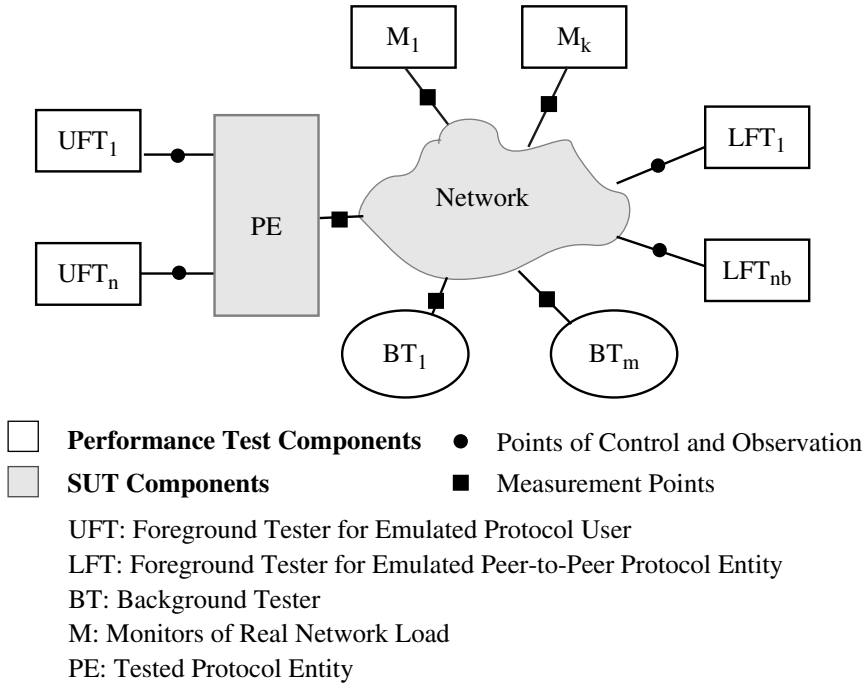


Figure 4. Performance test configuration for a communication protocol

As can be seen from the example (Figure 3) and the previous discussion, with our generic test architecture we provide a means to test really a distributed system. *ToBeTested1* and *toBeTested2* are running on separate systems. Even the test system is physically distributed. Some elements of the specific test architecture in Figure 3 have direct access to parts of the IUT through the interface component *LowerLayers*. In summary, the proposed generic test architecture complements CTMF along the testing architectures dimension and adds elements to enhance CTMF partly with regard to grey-box testing.

A further example shows the application of our generic test architecture to performance testing. The main objective of performance testing is to test the performance of a network component under normal and overload situations [47]. Performance testing identifies performance levels of the network component for different ranges of parameter settings and assesses the measured performance of the component. A performance test suite describes precisely the performance characteristics that have to be measured and procedures how to execute the measurements. In addition, the performance test configuration including the configuration of the network component, the configuration of the underlying network, and the network load characteristics are described.

Depending on the characteristics of the network component under test, different types of performance test configurations are defined: end-user telecommunication application, end-to-end telecommunication service and communication protocol (Figure 4). Foreground test components (FT) implement control and observation of the network under test. Background test components (BT) generate continuous streams of data to load the network component under test. Monitor components are used to monitor the real network load during the performance test. BTs do not control or observe directly the network under test but implicitly influences the network under test by putting the network into normal and overload situations.

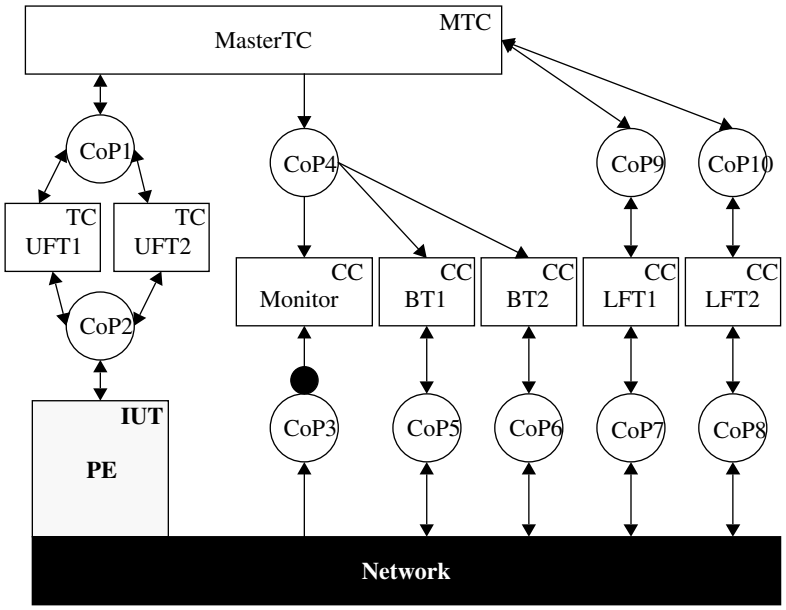


Figure 5. Performance test architecture

Figure 5 shows an instance of the performance testing configuration sketched in Figure 4 by using the generic test architecture model. The BTs (*BT1*, *BT2*) are meant to be load generators and therefore are mapped to CCs. The CC *Monitor* measures the real network load during a test run. The FTs (*UFT1*, *UFT2*, *LFT1*, *LFT2*) are mapped onto TCs. PCOs and measurement points are mapped onto CLs, their physical interface is described by using CoPs. Additionally, an MTC communicates with the other components in order to start and stop a test run.

3.2. TTCN — Tree and Tabular Combined Notation

In order to have the paper self-contained we give a brief introduction to TTCN [23]. TTCN is a notation for the description of test cases to be used in conformance testing. For the purpose of this paper we restrict our attention to TTCN concepts related to the description of the dynamic test case behaviour. Further details on TTCN can be found in [5,6,23,29,33,41,43].

3.2.1. Test case dynamic behaviour descriptions

The behaviour description of a TC consists of *statements* and *verdict assignments*. A verdict assignment is a statement of either PASS, FAIL or INCONCLUSIVE, concerning the conformance of an IUT with respect to the sequence of events which has been performed. TTCN statements are *test events* (SEND, IMPLICIT SEND, RECEIVE, OTHERWISE, TIMEOUT and DONE), *constructs* (CREATE, ATTACH, ACTIVATE, RETURN, GOTO and REPEAT) and *pseudo events* (qualifiers, timer operations and assignments).

Statements can be grouped into *statement sequences* and *sets of alternatives*. In the graphical form of TTCN, sequences of statements are represented one after the other on separate lines and being *indented* from left to right. The statements A1, A11, A111 in Figure 6 are a statement sequence. Statements on the same level of indentation and with the same predecessor are alternatives. In Figure 6 the statements A1, A2 and A3 form a set of alternatives.

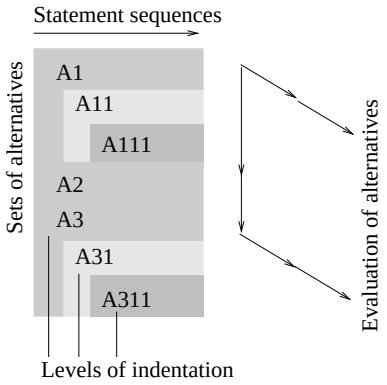


Figure 6. TTCN behaviour description

3.2.2. Test component execution

A TC starts execution of a behaviour description with the first *level of indentation* (the outermost left level of indentation in Figure 6) which becomes the current level of indentation, and proceeds towards the last level of indentation (either A111 or A311 in Figure 6). Only one alternative out of a set of alternatives at the current level of indentation is executed, and test case execution proceeds with the next level of indentation relative to the executed alternative. For example, in Figure 6 the statements A1, A2 and A3 are alternatives. A set of alternatives may consist of a single alternative, as is the case for the statements A11, A111 and A31, A311. If the statement A1 is executed, processing continues with the statement on the next level of indentation, i.e., the statement A11. Execution of a behaviour description stops if the last level of indentation has been visited (statements A111, A2 and A311 in Figure 6), a test verdict has been assigned, or a test case error has occurred. The latter two cases are not shown in Figure 6.

Before a set of alternatives is evaluated, a *snapshot* is taken [23], i.e., the state of the TC and the state of all PCOs, CPs and expired timer lists related to the TC are updated and frozen until the set of alternatives has been evaluated. This guarantees that evaluation of a set of alternatives is an *atomic* and *deterministic action*.

Alternatives are evaluated in sequence, and the first alternative which is *evaluated successfully* (i.e., all conditions of that alternative are fulfilled [23]) is executed. Execution then proceeds with the set of alternatives on the next level of indentation. If no alternative can be evaluated successfully, a new snapshot is taken and evaluation of the set of alternatives is started again.

Figure 7 gives an example of a behaviour description in the graphical form of TTCN. The example specifies a test case where data is sent in test step SendData MAX times. The loop is given by the GOTO on line 5 which branches to line 2. On line 2, the counter NumOfSends is incremented. Execution continues with sending data after which NumOfSends is tested against MAX.

3.3. Real-time test specification language

An essential part of the proposed integrated test methodology is concerned with extending TTCN into a real-time test specification language.

In our proposal for an extension of TTCN [51], language constructs are added that support the annotation of TTCN statements with time labels. The language extension is called *real-time TTCN*. An example of an real-time TTCN behaviour description is shown in Figure 8. The annotations of time labels is done in the *Time* column. Relative to the execution of the previous statement, the time labels define a time interval within which the TTCN statements must be

Test Case Dynamic Behaviour					
Nr	**Label**	**Behaviour Description**	**CRef**	**V**	**Comments**
1		[TRUE]			Qualifier
2	L1	(NumOfSends := NumOfSends + 1)			
3		+SendData			ATTACH
4		[NOT NumOfSends > MAX]			Alternative 1
5		-> L1			GOTO
6		[NumOfSends > MAX]			Alternative 2

Figure 7. Example of a TTCN behaviour description

Test Case Dynamic Behaviour							
Nr	**Label**	**Time**	**Time Options**	**Behaviour Description**	**C**	**V**	**Comments**
1	L1	2, 4	M	A ? DATA ind			Time label Mandatory *EET*
2				(NoDur := 3)			Time assignment
3		2, NoDur		A ! DATA ack			
4				(LET := 50)			LET update (ms)
5				A ? Data ind			
6		L1 + WFN, L1 + LET	M, N	B ? Alarm			Mandatory *EET* not pre-emptive

Figure 8. Annotation of TTCN behaviour lines with time labels

executed. The formal semantics of real-time TTCN is defined using *timed transition systems* [19]. In [52], it has been shown that real-time TTCN can be applied to multimedia systems for testing quality-of-service guarantees, i.e., soft real-time requirements.
In real-time TTCN, statements are annotated with time labels for earliest and latest execution times. Execution of a real-time TTCN statement is instantaneous. The syntactical extensions of TTCN (Section 3.3.2) are the definition of a table for the specification of time names and time units and the addition of two columns for the annotation of TTCN statements in the behaviour description tables. We define an operational semantics for real-time TTCN (Section 3.3.3). For this we define a mapping of real-time TTCN to timed transition systems [19] which are introduced in Section 3.3.1. Applying timed transition systems has been motivated by our experiences with the definition of an operational semantics for TTCN [53,54]. To emphasize the similarities of TTCN and real-time TTCN we also propose a refined snapshot semantics which takes time annotations into account and which is compliant with the timed transition system based semantics. In the following section we quote the main definitions of [19].

3.3.1. Timed transition systems

A *transition system* [26] consists of a set V of variables, a set Σ of states, a subset $\Theta \subseteq \Sigma$ of initial states and a finite set $\mathcal{T}$ of transitions which also includes the idle transition t_I. Every transition $t \in \mathcal{T}$ is a binary relation over states; i.e., it defines for every state $s \in \Sigma$ a possibly empty set $t(s) \subseteq \Sigma$ of so-called t-successors. A transition t is said to be *enabled* on state s if and only if $t(s) \neq \emptyset$. For the idle transition t_I we have that $t_I = \{(s, s) \mid s \in \Sigma\}$.

An infinite sequence $\sigma = s_0 s_1 \ldots$ is a *computation* of the underlying transition system if $s_0 \in \Theta$ is an initial state, and for all $i \geq 0$ there exists a $t \in \mathcal{T}$ such that $s_{i+1} \in t(s_i)$, denoted $s_i \xrightarrow{t} s_{i+1}$, i.e., transition t is *taken* at position i of computation σ.

The extension of transition systems to timed transition systems is that we assume the existence of a real-valued global clock and that a system performs actions which either advance time or change a state [19]. Actions are executed instantaneously, i.e., they have no duration.

A *timed transition system* consists of an underlying transition system and, for each transition $t \in \mathcal{T}$, an earliest execution time $EET_t \in \mathbb{N}$ and a latest execution time $LET_t \in \mathbb{N} \cup \{\infty\}$ is defined.[5] We assume that $EET_t \leq LET_t$ and, wherever they are not explicitly defined, we presume the default values are zero for EET_t and infinity (∞) for LET_t. EET_t and LET_t define timing constraints which ensure that transitions cannot be performed neither to early (EET_t) nor too late (LET_t).

A *timed state sequence* $\rho = (\sigma, T)$ consists of an infinite sequence σ of states and an infinite sequence T of times $T_i \in \mathbb{R}$ and T satisfies the following two conditions:

- *Monotonicity:* $\forall i \geq 0$ either $T_{i+1} = T_i$ or $T_{i+1} > T_i \wedge s_{i+1} = s_i$.
- *Progress:* $\forall t \in \mathbb{R}\, \exists\, i \geq 0$ such that $T_i \geq t$.

Monotonicity implies that time never decreases but possibly increases by any amount between two neighbouring states which are identical. If time increases this is called a *time step*. The transition being performed in a time step is the idle transition which is always enabled (see above). The progress condition states that time never converges, i.e., since $\mathbb{R}$ has no maximal element every timed state sequence has infinitely many time steps. Summarizing, in timed state sequences state activities are interleaved with time activities. Throughout state activities time does not change, and throughout time steps the state does not change.

A timed state sequence $\rho = (\sigma, T)$ is a *computation* of a timed transition system if and only if state sequence σ is a computation of the underlying transition system and for every transition $t \in \mathcal{T}$ the following requirements are satisfied:

- for every transition $t \in \mathcal{T}$ and position $j \geq 0$ if t is taken at j then there exists a position i, $i \leq j$ such that $T_i + EET_t \leq T_j$ and t is enabled on $s_i, s_{i+1}, \ldots, s_{j-1}$ and is not taken at any of the positions $i, i+1, \ldots, j-1$, i.e., a transition must be continuously enabled for at least EET_t time units before the transition can be taken.
- for every transition $t \in \mathcal{T}$ and position $i \geq 0$, if t is enabled at position i, there exists a position j, $i \leq j$, such that $T_i + LET_t \geq T_j$ and either t is not enabled at j or t is taken at j, i.e., a transition must be taken if the transition has been continuously enabled for LET_t time units.

A finite timed state sequence is made infinite by adding an infinite sequence of idle transitions or time activities.

3.3.2. Syntax of real-time TTCN

In real-time TTCN, timing information is added in the declarations and the dynamic part of a test suite.

As shown in Figure 9 the specification of time names, time values and units is done in an Execution Time Declarations table. Apart from the headings the table looks much like the

[5] In principle, time labels may not only be natural numbers. For an in-depth discussion of alternative domains for time labels, the reader is referred to [2].

Execution Time Declarations			
Time Name	**Value**	**Unit**	**Comments**
EET	1	s	EET value
LET	1	min	LET value
WFN	5	ms	Wait For Nothing
NoDur		min	No specified value

Figure 9. Execution Time Declarations Table.

TTCN Timer Declarations table. Time names are declared in the Time Name column. Their values and the corresponding time units are specified on the same line in the Value and Unit columns. The declaration of time values and time units is optional.

EET and LET[6] are predefined time names with default values zero and infinity. Default time values can be overwritten (Figure 9).

Besides the static declarations of time values in an Execution Time Declarations table, changing these values within a behaviour description table can be done by means of assignments (Figure 8). However, evaluation of time labels should alway result in EET and LET values for which $0 \leq EET \leq LET$ holds. As indicated in Figure 8 we add a Time and a Time Options column to Test Case Dynamic Behaviour tables (and similar for Default Dynamic Behaviour and Test Step Dynamic Behaviour tables). An entry in the Time column specifies EET and LET for the corresponding TTCN statement. Entries may be constants (e.g., line 1 in Figure 8), time names (e.g., the use of NoDur on line 3), and expressions (e.g., line 6).

In general, EET and LET values are interpreted relative to the enabling time of alternatives at a level of indentation, i.e., the time when the level of indentation is visited the first time. We call this time *enabling time*. However, some applications may require to define EET and LET values relative to the execution of an earlier test event, i.e, not restricted just to the previous one. In support of this requirement, a label in the Label column may not only be used in a GOTO but can also be used in the Time column, so that EET and LET values are computed relative to the execution time of the alternative identified by the label: In Figure 10 the situation is depicted where the sequence of statements numbered 1 to 4 are executed as indicated on the time axis. EET and LET of alternative line 5 are computed relative to the execution time of statement line 1, and not relative to the execution time of statement line 4 which where the case without reference to L1 on line 5.

Entries in the Time Options column are combinations of symbols M and N. Similar to using labels in expressions, time option N allows to express time values relative to the alternative's own enabling time even though some TTCN statements being executed in between two successive visits of the same level of indentation. Thus, the amount of time needed to execute the sequence of TTCN statements in between two successive visits is compensated: If time option N is defined, then execution of this alternative is not pre-emptive with respect to the timing of all alternatives at the same level of indentation.

In some executions of a test case, a RECEIVE or OTHERWISE event may be evaluated successfully before it has been enabled for EET units. If it is intended to define EET as a mandatory lower bound when an alternative may be evaluated successfully, then time option M has to be specified. Informally, if time option M is specified and the corresponding alternative can be successfully evaluated before it has been enabled for EET units, then this results in a

[6]We use different font types for distinguishing between syntax, EET and LET, and semantics, EET and LET.

Test Case Dynamic Behaviour					
Nr	**L**	**Time**	**TOptions**	**Behaviour Description**	**...**
1	L1	2, 4	M	A ? DATA ind	...
2				...	...
3				...	...
4				...	...
5	L2	L1 + WFN, L1 + LET		A ? Data ind	...
6				...	...
7				...	...

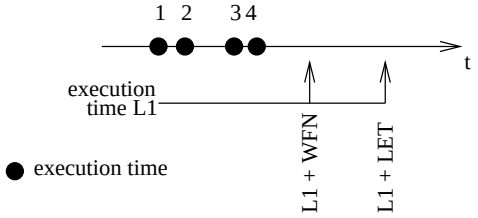


Figure 10. Real-time TTCN informal semantics: use of labels

Test Case Dynamic Behaviour					
Nr	**L**	**Time**	**TOptions**	**Behaviour Description**	**...**
1	L1	2, 4	M	A ? DATA ind	...
2				...	...
3				...	...
4				...	...
5				...	...
6				...	...
7				...	...

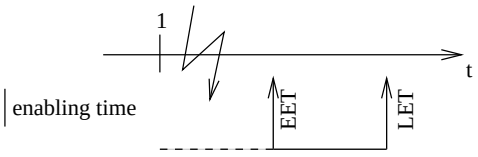


Figure 11. Real-time TTCN informal semantics: use of time option M

FAIL verdict. This situation is shown in Figure 11. Alternative line 1 should not be successfully evaluated before *EET* units after it has been enabled (see the line on the time axis).

3.3.3. Operational semantics of real-time TTCN

The operational semantics of real-time TTCN is defined in two steps:

1. We define the semantics of a TC using timed transition systems. An execution of a TC is given by a computation of the timed transition system associated with that TC. As time domain we use the real numbers $\mathbb{R}$ which are an *abstract* time domain in contrast to the *concrete* time domain of TTCN which counts time in discrete time units. Progress of time is, however, a continuous process adequately modelled by $\mathbb{R}$.

2. The semantics of a test system is determined by composing the semantics of individual TCs (for details see [51]).

Given a TC, we associate with it the following timed transition system: A state $s \in \Sigma$ of a TC is given by a mapping of *variables* to *values*. The set of variables V includes constants, parameters and variables defined for the TC in the test suite and, additionally, a variable for each timer. Furthermore, we introduce a *control variable* π which indicates the location of control in the behaviour description of the TC. π is updated when a new level of indentation is visited. We let PCOs and CPs be pairs of variables so that each holds a queue of ASPs, PDUs or CMs sent and received, respectively.

In the initial state of a TC all variables have assigned their initial values (if specified) or are undefined. All PCO and CP variables have assigned an empty queue and all timer variables have assigned the value *stop*. The control variable π has been initialized to the first level of indentation. If the TC is not running, i.e., the TC has not been created yet, then all variables are undefined.

The set $\mathcal{T}$ of transitions contains a transition for every TTCN statement in the TC behaviour description and the idle transition t_I. Furthermore, we have a transition t_E which models all activities performed by the environment, e.g., the updating of a PCO, CP or timer variables. Execution of t_E changes the state of the TC because shared PCO, CP or timer variables are updated.

In the following we assume that the current level of indentation has been expanded as defined in Annex B of [23]. After expansion its general form is $A_1[eexp_1, lexp_1], \ldots, A_n[eexp_n, lexp_n]$, where A_i denotes an alternative and $eexp_i$, $lexp_i$ are expressions for determining EET and LET values of alternative A_i. The evaluation of expressions $eexp_i$ and $lexp_i$ depends on whether $eexp_i$ and $lexp_i$ make use of a label Ln. If so, absolute time references are converted into time references relative to the enabling time of the current set of alternatives.

Let eval be a function from time expressions to time values for EET or LET. Let enablingTime (A_i) be a function that returns the time when alternative A_i has been enabled. Let executionTime (Ln) be a function that returns the execution time of an alternative at the level of indentation identified by label Ln. Function NOW returns the current global time. Notice that for all alternatives A_i in a set of alternatives, enablingTime(A_i) is the same. Since only one alternative of a set of alternatives is executed, executionTime(Ln) returns the execution time of the executed alternative. This discussion is summarized in Figure 12.

For the evaluation of time expressions in real-time TTCN behaviour description tables the following rules apply:

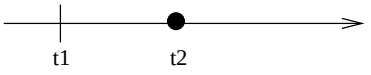
1. If $eexp_i$ and $lexp_i$ do not use any operator Ln then EET = eval($eexp_i$) and LET = eval($lexp_i$). It is required that $0 \leq EET \leq LET$ holds; otherwise test case execution should terminate with a test case error indication.

2. If $eexp_i$ and $lexp_i$ use any operator Ln then, firstly, executionTime(Ln) is substituted for Ln in $eexp_i$ and $lexp_i$ resulting in expressions $eexp'_i$ or $lexp'_i$, and secondly, EET = eval($eexp'_i$) – NOW and LET = eval($lexp'_i$) – NOW. It is required that $0 \leq EET \leq LET$ holds; otherwise test case execution should terminate with a test case error indication.

We say that alternative A_i is *potentially enabled* if A_i is in the current set of alternatives. A_i is *enabled* if A_i is evaluated successfully (Section 3.2.2), A_i is *executable* if A_i is enabled and A_i has been potentially enabled for at least EET_i and at most LET_i time units.

Refined snapshot semantics: We make the evaluation of a TC explicit by defining the following refined snapshot semantics (see Section 3.2.2).

Test Case Dynamic Behaviour					
Nr	**L**	**Time**	**TOptions**	**Behaviour Description**	**...**
1	L1			A1	...
2				A11	...
3				GOTO Ln	...
4				A2	...
5				A3	...
6				A31	...
7				GOTO L1	...



enablingTime(A1) = enablingTime(A2) = enablingTime(A3) = t1
executionTime(A1) = executionTime(L1) = t2
enablingTime(A11) = t2

Figure 12. Real-time TTCN informal semantics: enabling and execution times

1. The TC is put into its initial state.

2. A snapshot is taken, i.e., PCO, CP and timer variables are updated and frozen. The same applies for function NOW which for an iteration of the following steps returns the same value of the global clock.

   (a) If the level of indentation is reached from a preceding alternative (i.e., not by a GOTO or RETURN), then all alternatives are marked *potentially enabled* and the global time is taken and stored. The stored time is accessible by function enablingTime(A_i).

   (b) If the level of indentation is reached by executing a GOTO or RETURN and enabling Time(A_i) has been frozen (see Step 5 below), then all alternatives are marked *potentially enabled* but enabling Time(A_i) is not updated.

   (c) If the level of indentation is reached by executing a GOTO or RETURN but enabling Time(A_i) has not been frozen previously, then all alternatives are marked *potentially enabled* and the global time is taken and stored. The stored time is accessible by function enablingTime(A_i).

   (d) Otherwise, it is a new iteration of Steps 2-5.

   EET and LET are computed as described above.

   If for an alternative A_i enablingTime(A_i) + LET_i < NOW, then test case execution stops (FAIL verdict).

3. All alternatives which can be evaluated successfully are marked *enabled*. If no alternative in the set of alternatives can be evaluated successfully, then processing continues with Step 2.

   If for an enabled alternative, say A_i, time option M is set and if enablingTime(A_i)+EET_i > NOW, then test case execution stops with a FAIL verdict.

4. An enabled alternative A_i is marked *executable* provided that enablingTime(A_i)+$EET_i \leq$ NOW $\leq$ enablingTime(A_i) + LET_i and if there is another enabled alternative A_j with

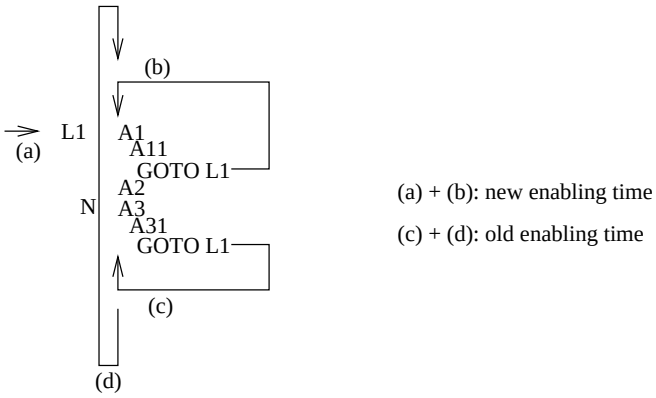


Figure 13. Real-time TTCN informal semantics: update scenarios for enabling time

enablingTime$(A_j) + EET_j \leq$ NOW $\leq$ enablingTime$(A_j) + LET_j$, then $i < j$, i.e., the i-th alternative precedes the j-th alternative in the set of alternatives.

If no alternative can be marked executable, then processing continues with Step 2.

5. The alternative A_i marked executable in Step 4 is executed. If a label Ln is specified then the alternative's execution time is stored. If time option N is specified for the executed alternative, enablingTime (A_i) is frozen for later use. Control variable π is assigned the next level of indentation.

   Test case execution terminates if the last level of indentation has been reached or a final test verdict has been assigned; otherwise, evaluation continues with Step 2.

While iterating through Steps 2 through 5 time progresses and the value of the global clock changes. However, function NOW returns the same value upon all invocations. Only between end and start of an iteration of the evaluation procedure, progress of time becomes visible (i.e., function NOW returns a different value). The evaluation of Steps 2 through 5 may consume time in reality, but we do not pose any requirements on the performance of a system that executes a test case, except that the explicitly specified timing requirements are to be obeyed. This implies that the test system is fast enough so that timely responses of a system under test are processed properly. In the model we can assume that Steps 2 through 5 are performed instantaneously.

The subitems of Step 2 (a) to (d) are represented in Figure 13. The arrows indicate from which statement the current level of indentation is reached: from the previous level (a), from a subsequent level by a GOTO (b), from a subsequent level by a GOTO (c) or by iterating through the evaluation of alternatives. The enabling time is update in cases (a) and (b) only. In case (c) the enabling time is kept because time option N is given for alternative A3 (which belongs to the set of alternatives (A1, A2 and A3) identified by label L1).

If no potentially enabled alternative can be evaluated successfully before latest execution time then a specified real-time constraint has not been met and test case execution stops. Conversely, if an alternative can be evaluated successfully before it has been potentially enabled for *EET* time units (Step 3), then a defined real-time constraint is violated, too, and test case execution terminates with an error indication. In Step 4, the selection of alternatives for execution from the set of enabled alternatives follows the same rules as in TTCN [23]. If a TC stops (Step 5), then the finite timed state sequence is extended to an infinite sequence by adding an infinite sequence of idle transitions. Every iteration of Steps 2 - 5 is assumed to be *atomic*.

Test Case Dynamic Behaviour					
Nr	**L**	**Time**	**TOptions**	**Behaviour Description**	**...**
1	L1	EET, LET		A1	...
2				A11	...
3				GOTO Ln	...
4		EET, LET		A2	...
5		EET, LET		A3	...
6				A31	...
7				GOTO L1	...

Example - Timed state sequences and snapshot semantics: According to Section 3.3.1, a computation of a TC is a timed state sequence $\rho = (\sigma, T)$. Since we have used a slightly different notation in this section, we have to adjust this in the following. Therefore, we substitute *potentially enabled* for *enabled* and *executed* for *taken*, the real-time TTCN snapshot semantics becomes:

1. for all alternatives A and positions $j \geq 0$, if alternative A is executed at position j of ρ, then there exist positions i and l with $i \leq l \leq j$ such that enablingTime$(A) = T_i$ and $T_i + EET \leq T_j$ and alternative A is evaluated successfully on states $s_l, s_{l+1}, \ldots, s_{j-1}$ and is not executed at any position $l, l+1, \ldots, j-1$ (see also item 1. on page 14).

2. for all alternatives A and positions $i \geq 0$, if enablingTime$(A) = T_i$, then there exists a position j, $i \leq j$, such that $T_i + LET \geq T_j$ and alternative A is not evaluated successfully in state s_j or A is executed at j provided no other alternative A' exists for which these conditions hold and which precedes A in the set of alternatives (see also item 1. on page 14).

For the case that alternative A has time option M specified, then 1 above becomes:

1. for all alternatives A and positions $j \geq 0$, if alternative A is executed at position j of ρ, then there exist positions i and l with $i \leq l \leq j$ such that enablingTime$(A) = T_i$ and $T_i + EET \leq T_j$ and alternative A is evaluated successfully on states $s_l, s_{l+1}, \ldots, s_{j-1}$ with $T_i + EET \leq T_l$ and is not executed at any position $l, l+1, \ldots, j-1$.

An informal interpretation of the above rules 1 and 2 is as follows: (1) alternative A is potentially enabled for at least EET time units before it is executed provided it can be evaluated successfully after having been potentially enabled, and (2) the first alternative in a set of alternatives that evaluated successfully, is executed at latest LET units after being potentially enabled.

For the illustration of concepts we use the following (partial) real-time TTCN behaviour description: The corresponding (partial) computation associated with the TC is commented subsequently:

$$(s_0, T_0) \longrightarrow \ldots$$

The TC is put into its initial state and T_0 becomes the enablingTime of alternatives A1, A2 and A3. Eventually, the alternatives have been potentially enabled for EET time units, i.e., for some position i enablingTime(A1) + $EET \leq T_i$. In a computation, progression of time takes place performing idle transition t_I.

$$(s_0, T_0) \longrightarrow \ldots$$
$$\ldots \xrightarrow{t_I} (s_0, T_i) \longrightarrow \ldots$$

In order for an alternative to evaluate successfully some external conditions, e.g., reception of a message, must be fulfilled. External events are mapped to transition t_E triggered by the environment of the TC.

$$(s_0, T_0) \longrightarrow \ldots$$
$$\ldots \xrightarrow{t_I} (s_0, T_i) \longrightarrow \ldots$$
$$\ldots \xrightarrow{t_I} \ldots \xrightarrow{t_E} (s_l, T_l) \longrightarrow \ldots$$

In the above computation progression of time and reception of a message is shown in the last line. Although being executable, alternative A1 is executed not later than enablingTime(A1) + LET time units. For a position j with

$$\text{enablingTime(A1)} + LET \leq T_l \leq T_j \leq \text{enablingTime(A1)} + LET$$

alternative A1 is executed thus giving the following computation:

$$(s_0, T_0) \longrightarrow \ldots$$
$$\ldots \xrightarrow{t_I} (s_0, T_i) \longrightarrow \ldots$$
$$\ldots \xrightarrow{t_I} \ldots \xrightarrow{t_E} (s_l, T_l) \longrightarrow \ldots$$
$$\ldots (s_j, T_j) \xrightarrow{t_{\text{A1}}} \ldots$$

Referring back to the description of the refined snapshot semantics, we have seen there, that while performing Steps 2 through 5, progress of time and changes of the environment of the TC, i.e., an update to PCO, CP and timer variables, may happen in parallel (although they become effective only when the next snapshot is taken). In a computation this concurrent behaviour is modelled as an interleaving of idle transitions t_I (which model progress of time) and environment transitions t_E (which model an update of PCO, CP and timer variables). The evaluation of alternatives and the execution of a successfully evaluated TTCN statement coincide are executed instantaneously. Only execution of an alternative is accounted for in a computation. From this we conclude that a computation represents a specific execution of a TC, which is consistent with the refined snapshot semantics.

4. Example - Quality-of-Service testing

4.1. Teleteaching - A multimedia application scenario

Teleteaching is a multimedia application that uses multimedia workstations distributed over a wide area network [57]. Each workstation acts as a communication unit that transmits, receives, and processes video, audio and data streams. Every teleteaching participant should have a feeling like being in a face-to-face meeting. Figure 14 describes the scenario schematically.

In most cases, for each data stream there exists some metric for describing the quality of service (QoS) expected by users. For instance, for a video stream we may distinguish high-definition-television (HDTV) quality, (PAL) colour quality, and black-and-white quality. Besides QoS values for individual data streams there also exist QoS values describing the quality of synchronization between different streams. According to [49], in most cases the synchronization of data streams in multimedia scenarios is *soft synchronization*. This means that the synchronization can be done within some time interval. The extent to which the synchronization should be

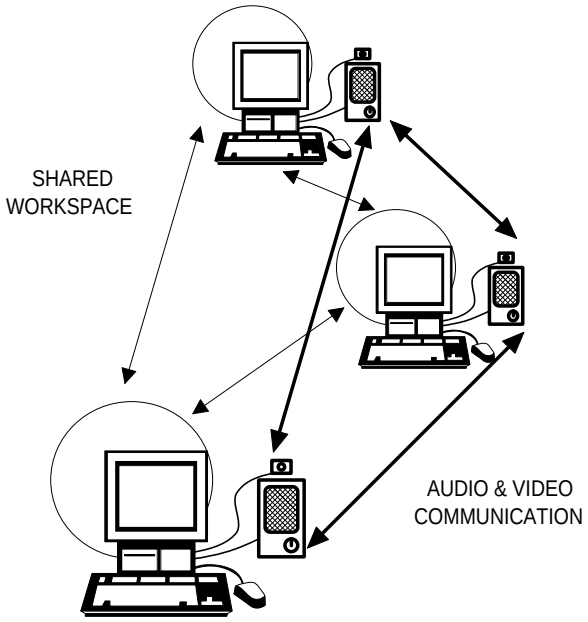


Figure 14. Teleseminar application scenario

QoS value	video before audio	audio before video
optimal	0 – 20 ms	–
good	20 – 40 ms	0 – 20 ms
acceptable	40 – 80 ms	20 – 80 ms
not acceptable	$>$ 80 ms	$>$ 80 ms

Figure 15. Possible assignment of QoS values for synchronization

achieved depends on the combination of data streams: A good approximation for the synchronization of video and audio is about 80 ms which means the audio should be at most 80 ms ahead of the video or at most 80 ms behind the video. In order to enable the synchronization of audio and video data so-called *event stamps* [48] are introduced in the data streams, i.e., are attached to audio and video data packets. Synchronization is performed relatively to these event stamps.

For lip synchronization of video and audio, we may distinguish several degrees, i.e., QoS values, of synchronization. Possible QoS values are *optimal*, *good*, *acceptable*, and *not acceptable*. Relating time intervals to these QoS values may be influenced by the preferences of the users. But field trials have shown that video before audio is more accepted than the other way round [49]. Figure 15 shows an example of how QoS values for lip synchronization may be related to time intervals.

4.2. Quality-of-Service testing issues

As explained informally in the previous section, quality-of-service (QoS) refers to a set of parameters that characterize a connection between communicating entities across a network [50,56].

The negotiation of QoS values takes place between calling and called service users (e.g., multimedia applications) and service provider. We distinguish between *best effort*, *guaranteed*, *compulsory*, *threshold*, and *mixed compulsory and threshold* QoS values [15]. A QoS semantics [3,15] defines the way how QoS values are negotiated during connection establishment and how QoS values are managed for the lifetime of a connection. Management of QoS values implies monitoring of data streams in order to determine the actual QoS and maintenance of QoS values.

Guaranteed QoS requires the highest degree of commitment from a service provider in maintaining the QoS of a connection. Particularly, the service provider has to take any necessary precautions so that under any conceivable circumstances the negotiated QoS values are supported. For threshold and compulsory QoS the obligations of a service provider are monitoring the QoS values and informing service users as soon as a violation of negotiated QoS values is detected (threshold QoS values) or aborting the connection in the case that the QoS has become worse than the negotiated ones (compulsory QoS values).

Threshold, compulsory and guaranteed QoS semantics all require that a service provider besides implementing the usual protocol functions, is also requested to implement additional functions for QoS management such as a monitoring function in order to predict actual QoS values of data streams.

QoS testing, to our understanding, refers to assessing the behavior of a service provider performing QoS management.

For our teleteaching scenario, the question is what should happen if audio data is, for instance, delayed for more than 80 ms (not acceptable QoS according to Figure 15)? If we assume that the QoS semantics agreed upon between the multimedia applications involved is supporting *compulsory* QoS values then the users of the applications should receive an indication that the negotiated QoS values are violated and, according to the QoS semantics as defomed om [15], the audio and video connection should be aborted. For this scenario we develop a test case in the following section. Note that any other reaction of the service provider to QoS violations is possible but the concrete behaviour, if such a situation occurs, has to be given in the corresponding protocol specification to be testable.

4.3. Application of generic testing architecture and real-time TTCN

For simplicity, we consider a teleteaching session with two participants only. The two sites A and B in Figure 16 exchange audio and video data over an underlying network, i.e., interface component Communication Subsystem in Figure 16. On each site we install a test component which acts as a teleteaching user. In addition to test components user A and B, we have another test component Monitor and a controlled component Load Generator. The latter component has the purpose to feed the network with data so that it becomes saturated and, as a consequence, audio or video data are delayed and run out of synchronization. It is the responsibility of component Site B to advice User B that due to a late arrival of packets of either data stream, the negotiated QoS value for synchronization is violated. The test component Monitor looks into the traffic from the network to site B. As soon as the Monitor detects an out of synchronization of audio and video, this should be detected by site B as well and should be indicated to User B and, subsequently, the connection should be aborted (as prescribed by the compulsory QoS semantics).

The communication links between IUTs and test system components are as indicated in Figure 16 (for an explanation of symbols please refer to Figure 3). Over the synchronous communication link between User A and Load Generator the transfer of normal data and background traffic is coordinated.

We only discuss a small fraction of the overall dynamic test case behaviour description for

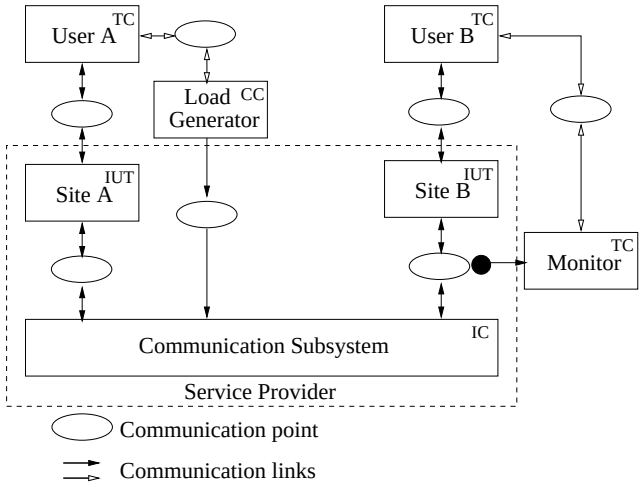


Figure 16. Teleseminar testing architecture

Test Case Dynamic Behaviour							
Nr	**Label**	**Time**	**Time Options**	**Behaviour Description**	**C**	**V**	**Comments**
1	L1			?audio			
2		0, 80ms		?video			sync video
3				GOTO L1			loop
4		80ms,		!alarm			out of sync
5				?video			
6		0, 80ms		?audio			sync audio
7				GOTO L1			loop
8		80ms,		!alarm			out of sync

Figure 17. Real-time TTCN behaviour description of component Monitor

our test scenario. In Figure 17 the behaviour of component Monitor is given. Essentially, Monitor loops through the sequence of events on lines 1–3 and lines 5–7 if audio and video are received within 80 ms delay. Otherwise, an alarm should be sent to User B over the synchronous communication link between Monitor and User B. Upon reception of the alarm signal, User B should be prepared to receive an out-of-synchronization indication from Site B and an abort connection indication.

5. Related work

The contributions of the paper are threefold: Firstly, classifications of distributed systems and types of testing have been presented (Section 2). Based on these classifications we, secondly, have defined testing architectures suitable for testing a diverse set of distributed systems with respect to different testing objectives. And, thirdly, a test notation is proposed for testing real-time aspects of distributed systems. In this section, we relate our research results to those reported in the literature.

To our best knowledge the classification schemes discussed are new. Although definitions of

basic concepts such as real-time distributed system, testing objectives, testing approaches and testing architectures can be found in the literature, we put these concepts into close relation. The advantage of our approach is that within these classification schemes we can precisely identify the holes that have to be filled so to get a testing methodology for real-time distributed systems. Or, to state it positively, we can locate those approaches known from the literature.

Real-time distributed systems: Our definition of real-time distributed system obeys much of the characteristics of a *real-time system*, i.e., a system that reacts to external inputs and in a timely manner affects the environment in which it operates [25,27]. However, in this paper we explicitly require real-time systems to be composed of communicating processes. This leads to the definition of parallel and distributed systems, i.e., systems running processes that have several threads of control. In a further step, we have refined these definitions by stating that computers executing processes may be geographically distributed. In other terms, in distributed systems communication between processes lasts longer than the individual computer's instruction times [14].

Testing approaches: As already mentioned in previous parts of the paper, testing software systems has a long tradition and is well understood [7,8,37,45]. With the conformance testing methodology and framework (CTMF) [21] the basic techniques of software testing have been put into a framework that makes these techniques applicable to distributed systems. In CTMF distribution means that a system under test (SUT) is tested by a tester from remote over a communication network. Only the tester may be distributed over several real systems whereas the SUT is a single real system. CTMF follows the black-box testing approach which is known from the literature above cited.

Testing architectures: In this paper, we have defined a testing environment where testers and systems under test can be distributed. Communication between components, i.e., test components and system under test, may be asynchronous or synchronous, and unicast (i.e., point-to-point) or multicast (i.e., point-to-multipoint). Even monitoring of communication is supported in the architecture (we have used this type of communication in the discussion of the example in the previous section). Our proposed testing architectures covers CTMF [21] and other testing architectures such as [16,34] as a subset. We do not discuss this here but refer the reader to Section 3.1.

In [16,34], the authors extend CTMF such that implementations under test may span over several distributed systems. Communication between testers and a distributed implementation is along points of control and observations, is asynchronous and is unicast. The difference between the two cited approaches is that in [16] communication between testers is supported by points of coordination (which are called coordination points in CTMF [21]), whereas in [34] communication between testers is possible only through the implementation itself. Additionally, in [16] a timer process and a specific point of timing are used for the coordination of testers with respect to time information. In order to retrieve time information, a process has to communicate asynchronously with a timer process through a point of timing.

Starting with [16], we extend this architecture with respect to the possible types of communication between testers and testers and IUT (see above). And with respect to time information we use the concept of a single time source which, in a real system, may be implemented by a *global positioning system* which also is a time-transfer system [32]. Each test component keeps track of its own time and timers. The testing architecture presented in [34] restricts itself to testers, PCOs and IUT which quite obviously is a simple instance of our testing architecture.

Some testing architectures have been proposed for performance testing [24,47]. As it has been shown previously, the performance testing architectures are covered by our generic test architecture, too.

Test notation: A testing architecture is only one component within a general framework for distributed real-time system testing. A notation for specifying behaviour of test components is another one. Real-time TTCN obeys much of the ideas presented in [19,23].

The basic approach to introduce real-time into TTCN is by annotating TTCN statements. This approach is straightforward and has been used in several other approaches such as time Petri Nets [9,36], timed automata [1], temporal communicating systems [13], LOTOS [11,18,30,42], SDL [18,31] and ESTELLE [17], too. As in the cited literature, our approach allows the timing of actions relative to the occurrence of previous actions. Timed transition systems are the foundation for an operational semantics for real-time TTCN. We found timed transition systems a suitable model for real-time TTCN since we already had experience with a definition of an operational semantics for TTCN by means of tranisition systems [29]. A difference of the cited approaches and ours is that the former are used for the specification of functional and real-time requirements of systems, whereas our emphasis is put on testing real-time requirements. Furthermore, timed transition systems as used in our approach assume existence of a global clock which provides all processes with the current global time.

If we assume that no time values are defined (in this case EET and LET are zero and infinity, respectively), execution of a test case results in the same sequence of state-transitions as in TTCN. Therefore, our definition of real-time TTCN is compatible to TTCN [5,23].

In real-time TTCN all requirements on the execution of a test case are given in the test case specification. For the execution of a test case we require that any test system obeys all timing requirements that are part of the behaviour descriptions. This extends also to the time spent for the evaluation of snapshots which has to be fast enough so that any changes of a TC's environment can be handled properly. The latter, however, is an implementation issue which beyond the specification of test cases.

Testing objectives: In [12], ten testing objectives are identified which can be roughly put into two categories: functional and non-functional testing objectives. From the non-functional testing objectives we have dealt with timing requirements. Reliability aspects are not covered by our approach, and performance aspects to a limited extent only.

6. Conclusions

In this paper, we have discussed extensions of CTMF which make it applicable to testing real-time requirements in a distributed environment. With respect to our classification of different types of testing (Figure 2) we presented extensions along the testing architecture and testing objectives axes. With the generic testing architecture, a tool-box of components is provided that can be combined to testing architectures so that test case specifications for various distributed systems will be possible. Since the definition of a testing architecture is only one part of the overall story, we have defined an extended TTCN, called real-time TTCN, for the specification of real-time behaviour of test components. The generic testing architecture in combination with real-time TTCN opens a wide spectrum of new applications for CTMF.

Although the extensions are serious changes to CTMF and, especially, to TTCN, the new versions are downwards compatible with the older ones. For the generic test architecture, we have added some components to the ones already defined in CTMF. Points of control and observation can be recovered from communications points with bidirectional asynchronous communication

links attached connecting two test components only. The compatibility of TTCN and real-time TTCN is given if zero and infinity are set for *EET* and *LET*, and time options and time labels are not used.

Parts of the discussed extensions to CTMF and TTCN are considered by ETSI for integration into a future version of TTCN. This effort is done by an expert team which has been established within ETSI. One author is member of this expert team. Furthermore, the test architecture will be implemented in the context of a research project funded by the Swiss National Science Foundation.

Acknowledgements. The authors are indebted to Stefan Heymer for proofreading and for his detailed comments on earlier drafts of this paper. The authors would like to gratefully acknowledge the anonymous referees for fruitful comments to the previous version of this paper.

REFERENCES

1. R. Alur, D. Dill. A theory of timed automata. In *Theoretical Computer Science*, Vol. 126, Elsevier, 1994.
2. R. Alur, T. Feder, T. Henzinger. The Benefits of Relaxing Punctuality. In *Journal of the ACM*, Vol. 43, 1996.
3. C. Aurrecoechea, A. Campbell, L. Hauw. A Review of Quality of Service Architectures. In *ACM Multimedia Systems Journal*, November 1995.
4. M. Anlauf. Programming service tests with TTCN. In A. Petrenko, N. Yevtushenko, editors, *Testing of Communicating Systems*, volume 11, Kluwer Academic Publishers, September 1998.
5. B. Baumgarten, G. Gattung. A Proposal for the Operational Semantics of Concurrent TTCN. *Technical Report, GMD 975*, 1996.
6. B. Baumgarten, A. Giessler. OSI conformance testing methodology and TTCN. Elsevier, 1994.
7. Boris Beizer. Software Testing Techniques Second Edition. Van Nostrand Reinhold, New York, 1990.
8. B. Beizer. Black-box testing: Techniques for functional testing of software and systems. Wiley, 1995.
9. B. Berthomieu, M. Diaz. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. In *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991.
10. J. Bi, J. Wu. Application of a TTCN based conformance test environment on the Internet email protocol. In M. Kim, S. Kang, K. Hong, editors, *Testing of Communicating Systems*, volume 10, Chapman & Hall, September 1997.
11. H. Bowman, L. Blair, G. Blair, A. Chetwynd. A Formal Description Technique Supporting Expression of Quality of Service and Media Synchronization. In *Multimedia Transport and Teleservices*, Lecture Notes in Computer Science 882, Springer-Verlag, 1994.
12. R. W. Buchanan. The Art of Testing Network Systems. Wiley Computer Publishing, 1996.
13. L. Cacciari, O. Rafiq. Validation of protocols with temporal constraints. In *Computer Communications*, Vol. 19, 1996.
14. G. Coulouris, J. Dollimore, T. Kindberg. Distributed Systems Concepts and Design, Second Edition. Addison Wesley, 1994.
15. A. Danthine, Y. Baguette, G. Leduc, L. Léonard. The OSI 95 Connection-Mode Transport Service - The Enhanced QoS. In A. Danthine, O. Spaniol, editors, *High Performance Networking*, North-Holland, 1993.
16. J. de Meer, V. Heymer, J. Burmeister, R. Hirr, A. Rennoch. Distributed Testing.

In *Workshop on Protocol Test Systems*, Participants proceedings, also available from http://www.fokus.gmd.de, 1991.
17. S. Fischer. Implementation of multimedia systems based on a real-time extension of Estelle. In R. Gotzhein, J. Bredereke, editors, *Formal Description Techniques IX Theory, application and tools*, Chapman & Hall, 1996.
18. D. Hogrefe, S. Leue. Specifying Real-Time Requirements for Communication Protocols. *Technical Report IAM 92-015*, University of Berne, 1992.
19. T. Henzinger, Z. Manna, A. Pnueli. Timed Transition Systems. In *Real-Time: Theory in Practice*, Lecture Notes in Computer Science 600, Spinger-Verlag, 1991.
20. ANSI/IEEE. Glossary of Software Engineering Terminology. *ANSI/IEEE Std 729-1983*, ANSI/IEEE Std 729-1983, 1983.
21. ISO. Information Technology - OSI - Conformance Testing Methodology and Framework - Parts 1 - 7. ISO, IS 9646, 1994 - 1997.
22. ISO. Information Technology - OSI - Conformance Testing Methodology and Framework - Part 1: General Concepts. ISO IS 9646-1, 1994.
23. ISO. Information Technology - OSI - Conformance Testing Methodology and Framework - Part 3: The Tree and Tabular Combined Notation (TTCN). ISO IS 9646-3, 1997.
24. R. Jain, G. Babic, A. Durresi. ATM Forum Performance Testing Specification - Baseline Text. *ATM Forum Document Number: BTD-TEST-TM-PERF.00.05 (96-0810R8)*, February 1998.
25. K. M. Kavi (editor). Real-Time Systems - Abstactions, Languages, and Design Methodologies. IEEE Computer Society Press, 1992.
26. R. Keller. Formal Verification of Parallel Programs. In *Communications of the ACM*, Vol. 19, No. 7, 1976.
27. H. Kopetz, P. Verissimo. Real Time and Dependability Concepts. In S. Mullender, editor, *Distributed Systems*, Addison Wesley, 1993.
28. A. Kündig. A Note on the Meaning of Embedded Systems. In A. Kündig, R. Bührer, editors, *Embedded Systems*, Lecture Note in Computer Science 284, Springer-Verlag, 1986.
29. F. Kristoffersen, T. Walter. TTCN: Towards a formal semantics and validation of test suites. In *Computer Network and ISDN Systems*, Vol. 29, 1996.
30. L. Léonard, G. Leduc. An Enhanced Version of Timed LOTOS and its Application to a Case Study. In R. Tenney, P. Amer, M. Uyar, editors, *Formal Description Techniques VI*, North-Holland, 1994.
31. S. Leue. Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic Based Approach. In P. Dembinski, M. Sredniawa, editors, *Protocol Specification, Testing and Verification XV*, Chapman & Hall, 1996.
32. W. Lewandowski, J. Azoubib, W. Klepczynski. GPS: Primary Tool for Time Transfer. In *Proceedings of the IEEE*, Vol. 87, No. 1, 1999.
33. R. Linn. Conformance Evaluation Methodology and Protocol Testing. In *IEEE Journal on Selected Areas in Communications*, Vol. 7, No. 7, 1989.
34. G. Luo, R. Dssouli, G. v. Bochmann, R. Venkatarm, A. Ghedamsi. Test generation with respect to distributed interfaces. In *Computer Standards & Interfaces*, Vol. 16, 1994.
35. Z. Manna, A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1991.
36. P. Merlin, D. Faber. Recoverability of Communication Protocols. In *IEEE Transactions on Communications*, Vol. 24, No. 9, September 1976.
37. G. J. Myers. The Art of Software Testing. John Wiley, 1979.
38. O. Maler, Z. Manna, A. Pnueli. From Timed to Hybrid Systems. In J. de Bakker, C. Huizing,

W. de Roever, G. Rozenberg, editors, *Real-Time: Theory and Practise*, Springer-Verlag, 1991.
39. D. Patterson, J. Hennessy. Computer Organization & Design The Hardware/Software Interface. Morgan Kaufmann Publishers, 1994.
40. A. Pope. The CORBA Reference Guide - Understanding the Common Object Request Broker Architecture. Addison Wesley, 1998.
41. R. Probert, O. Monkewich. TTCN: The International Notation for Specifying Tests of Communications Systems. In *Computer Networks and ISDN Systems*, Vol. 23, 1992.
42. J. Quemada, A. Fernandez. Introduction of Quantitative Relative Time into LOTOS. In H. Rudin, C. West, editors, *Protocol Specification, Testing and Verification VII*, North-Holland, 1987.
43. B. Sarikaya. Conformance Testing: Architectures and Test Sequences. In *Computer Networks and ISDN Systems*, Vol. 17, 1989.
44. I. Schieferdecker, A. Rennoch. Formal Based Testing of ATM Signalling. In U. Herzog, H. Hermanns, editors, *Formale Beschreibungstechniken für Verteilte Systeme*, Band 29, Nummer 9, Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung, Erlangen, Mai 1996.
45. I. Sommerville. Software engineering. Addison Wesley, 1989.
46. I. Schieferdecker, M. Li, A. Hoffmann: Conformance Testing of TINA Service Components - the TTCN/CORBA Gateway. In *Proceedings of the 5th International Conference on Intelligence in Services and Networks*, Antwerp, Belgium, May 1998.
47. I. Schieferdecker, S. Stepien, A. Rennoch. PerfTTCN, a TTCN Language Extension for Performance Testing. In M. Kim, S. Kang, K. Hong, editors, *Testing of Communicating Systems*, volume 10, Chapman & Hall, September 1997.
48. R. Steinmetz. Synchronization Properties in Multimedia Systems. In *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 3, April 1990.
49. R. Steinmetz, K. Nahrstedt. Multimedia: Computing, Communications & Applications. Prentice Hall, 1995.
50. A. Vogel, B. Kerhervé, G. v. Bochmann, J. Gecsei. Distributed Multimedia and QOS: A Survey. In *IEEE MultiMedia*, 1995.
51. T. Walter, J. Grabowski. A Proposal for a Real-Time Extension of TTCN. In M. Zitterbart, editor, *Kommunikation in Verteilten Systemen*, Informatik aktuell, Springer-Verlag, 1997.
52. T. Walter, J. Grabowski. Real-time TTCN for Testing Real-time and Multimedia Systems. In M. Kim, S. Kang, K. Hong, editors, *Testing of Communicating Systems*, volume 10, Chapman & Hall, September 1997.
53. T. Walter, J. Ellsberger, F. Kristoffersen, P.v.d. Merkhof. A Common Semantics Representation for SDL and TTCN. In R. J. Linn, M. Ü. Uyar, *Protocol Specification, Testing and Verification XII*, North-Holland, 1992.
54. T. Walter, B. Plattner. An Operational Semantics for Concurrent TTCN. In G. v. Bochmann, R. Dssouli, A. Das, editors, *Protocol Test Systems V*, North-Holland, 1992.
55. T. Walter, I. Schieferdecker, J. Grabowski. Test Architectures for Distributed Systems - State of the Art and Beyond (Invited Paper). In A. Petrenko, N. Yevtushenko, editors, *Testing of Communicating Systems*, volume 11, Kluwer Academic Publishers, September 1998.
56. ITU. Data Communication Networks Open Systems Interconnection (OSI) Model and Notation, Service Definition. X.200 - X.219, 1988.
57. S. Znaty, T. Walter, M. Brunner, J.-P. Hubaux, B. Plattner. Multimedia Multipoint Teleteaching over the ATM Pilot. In B. Plattner, editor, *Broadband Communications*, Lecture Notes in Computer Science 1044, Springer-Verlag, 1996.