Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests

Dissertation

zur Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von
Helmut Wolfram Neukirchen
aus Krefeld

Göttingen 2004

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbiografie; detaillierte bibliografische Daten sind im Internet unter `http://dnb.ddb.de` abrufbar.

Fehler-bereinigte Auflage 19. April 2006

Revised edition 19th of April 2006

D 7
Referent: Prof. Dr. Jens Grabowski
Korreferent: Prof. Dr. Dieter Hogrefe
Tag der mündlichen Prüfung: 25. August 2004

Zusammenfassung

Für moderne Verteilte Systeme ist es wichtig, dass sie Echtzeit-Anforderungen einhalten, um z.B. innerhalb einer gegebenen Zeitspanne eine Antwort auf eine Anfrage zu liefern. Testen ist die wichtigste Maßnahme zur Qualitätssicherung von Software-Systemen. Das Echtzeit-Testen von Verteilten Systemen wird jedoch bisher nur unzureichend unterstützt.

Diese Dissertation behandelt Sprachen, Werkzeuge und Muster für die Beschreibung von verteilten Echtzeit-Tests. Zur Testbeschreibung wird *Timed*-TTCN-3 eingeführt. Es handelt sich hierbei um eine Echtzeit-Erweiterung der standardisierten Testbeschreibungssprache *Testing and Test Control Notation version 3* (TTCN-3). Um die Entwicklung von Echtzeit-Tests zu vereinfachen, werden ein Werkzeug und die zugrundeliegenden Übersetzungsregeln vorgestellt, die es ermöglichen, *Timed*TTCN-3 Echtzeit-Testfälle aus Echtzeit-Testzwecken zu generieren. Echtzeit-Testzwecke werden hierzu anhand von *Message Sequence Chart*s (MSCs), die Echtzeit-Eigenschaften enthalten, formalisiert. Um die Spezifikation von Echtzeit-Anforderungen und die Auswertung von Echtzeit-Tests zu vereinheitlichen, werden *Real-time Communication pattern*s (RTC-patterns) eingeführt. Diese Muster bieten wiederverwendbare Lösungen zur Spezifikation von Echtzeit-Tests mit MSC und *Timed*TTCN-3. In diesem Zusammenhang werden außerdem eine Übersicht und eine allgemeine Klassifikation von existierenden Test-Mustern gegeben.

Abstract

For modern distributed systems, it is important that they adhere to real-time requirements, e.g., to deliver a response to a request within a given deadline. For assuring the quality of software systems, testing is the most important means. However, a mature support for real-time testing of distributed systems is missing.

This thesis treats languages, tools, and patterns for the specification of distributed real-time tests. For test specification, *Timed*TTCN-3 is proposed. It is a real-time extension of the standardised *Testing and Test Control Notation version 3* (TTCN-3). To ease real-time test development, a tool and underlying transformation rules which allow to generate *Timed*TTCN-3 test cases from real-time test purposes are presented. *Message Sequence Chart*s (MSCs) are used to express real-time properties as formalised real-time test purposes. In order to harmonise real-time requirement specification and real-time test evaluation, *Real-time Communication pattern*s (RTC-patterns) are introduced. They provide reusable solutions for real-time test specification based on MSC and *Timed*TTCN-3. The work on this kind of patterns includes also a survey and a classification of existing test patterns in general.

Acknowledgements

I like to thank my supervisor Prof. Dr. Jens Grabowski for his kind support during my research. His knowledge and valuable suggestions contributed much to the results which are presented in this thesis.

I would also like to thank my former supervisor Prof. Dr. Dieter Hogrefe for providing excellent working conditions during my time as research assistant at the Institute for Telematics in Lübeck.

Furthermore, I am grateful to all my former colleagues in Lübeck and to my current colleagues in Göttingen. It was always a pleasure to work with them and the discussions I shared with them were very fruitful.

This thesis would not be in its current shape without the comments of numerous people. Thus, I truly appreciate the efforts of Zhen Ru Dai, Michael Ebner, Christoph Neukirchen, Markus Ottensmann, and Edith Werner for proof reading.

All this would not be possible without my parents who stimulated my interest in computer science and encouraged me during my studies.

Finally, I like to thank Ruth for being there and being patient when I was not there.

Contents

Chapter 1

Introduction

Modern distributed systems are becoming more and more complex and have to adhere to real-time requirements. Such systems pervade daily life: In business and administration, e.g., online brokers provide share prices in real-time, and subsequent orders need to be processed within seconds. In the telecommunication and multimedia domain, e.g., *Voice over IP* (VoIP) requires to transfer audio signals within a few 100 milliseconds via the Internet and to replay it with a jitter of less than a half millisecond to deliver an acceptable quality. Process control in industrial plants or air traffic control requires that hard time deadlines are met, otherwise, a disaster might result. Testing is the most important means to give confidence that a distributed real-time system implementation meets its requirements with respect to functional and real-time behaviour.

Languages, methods, and tools for specifying functional tests of distributed systems have become mature: For test specification, the *Testing and Test Control Notation version 3* (TTCN-3) is standardised and well supported by tool providers. Powerful test generation tools allow to derive test cases from formal specifications based on standardised industrial strength specification languages. By this means, international telecommunication standards provide not only protocol specifications, but also corresponding test specifications which allow to assess the conformance of an implementation to its specification. A comparable and adequate support for real-time test specification is lacking.

1.1 Scope of this Thesis

In this thesis, an approach for the specification of black-box tests for testing hard real-time requirements of distributed systems is presented. This comprises the following contributions:

1. A TTCN-3-based language for specifying distributed real-time black-box tests.
2. Automatic generation of real-time test cases from graphical test purpose definitions.
3. Patterns for harmonising real-time requirement description and real-time test evaluation.

The work presented in this thesis originates from the author's participation in the *INTERVAL* project [Int02], a European research project which aimed at formal design, validation and testing of real-time telecommunications systems. The continuation of this work resulted in the participation in the *Patterns in Test Development* (PTD) [ETS04] work item at the *European Telecommunications Standards Institute* (ETSI).

1.2 Structure of this Thesis

The structure of this thesis is as follows: After this introduction, some established foundations for this thesis are given in Chapter 2. This includes an overview on testing, the testing process in general, and testing of real-time requirements in particular. Furthermore, the chapter provides an introduction into the *Message Sequence Chart* (MSC) language, the *Inres* protocol case study which is used throughout this thesis and the *Testing and Test Control Notation version 3* (TTCN-3).

In Chapter 3, *Timed*TTCN-3 and its alternative presentation formats are presented. *Timed*TTCN-3 is a real-time extension of the TTCN-3 test specification language. *Timed*TTCN-3 allows to specify distributed real-time tests. Real-time testing with *Timed*TTCN-3 is based on the generation and evaluation of time stamps for time critical events.

Subsequently, in Chapter 4, the generation of real-time test cases from graphical test purpose descriptions is explained. This is achieved by using real-time test purposes which are formalised as MSCs containing real-time constraints. Hence, a tool is able to generate *Timed*TTCN-3 real-time test cases from them. The underlying transformation rules are outlined in this chapter. This includes generation of real-time test cases for both kinds of test architectures, local and distributed ones.

A supplemental method for obtaining test cases is a pattern-based approach, which is presented in Chapter 5. Besides a survey on existing patterns and a classification scheme for test patterns, *Real-time Communication patterns* (RTC-patterns) are introduced. RTC-patterns can be used for pattern-based specification of real-time requirements and real-time test purposes. Since each RTC-pattern is accompanied by *Timed*TTCN-3 code for generating and evaluating time stamps, corresponding real-time test cases can be easily obtained by using RTC-patterns.

Finally, a summary, a discussion of related work, and an outlook are given as a conclusion in Chapter 6. This thesis is completed by a list of acronyms and the referenced bibliography.

1.3 Dependencies of Chapters

Chapters 3 to 5 depend on the foundations provided by Chapter 2. Nevertheless, a reader who is familiar with the principles of testing, the Inres example protocol, TTCN-3 as well as MSC and its real-time constructs, can safely skip Chapter 2.

*Timed*TTCN-3, which is presented in Chapter 3, is heavily used in Chapter 4 and in Section 5.3. Thus, Chapter 4 and Section 5.3 should not be read without the foundation of Chapter 3. However, a reader who is only interested in using RTC-patterns for real-time requirements specification, but not in their application to testing, can read Chapter 5 on its own, but should skip Section 5.3.

A reader who just wants to get a quick overview on the topics of this thesis should read the summaries which are provided at the end of each chapter and the overall summary and conclusions given in Chapter 6.

Chapter 2

Foundations

This chapter provides the foundations which are used in the subsequent chapters. Section 2.1 gives an introduction into software testing and the testing process. Then, in Section 2.2, testing of real-time properties is discussed. This is followed by a section describing *Message Sequence Chart*s (MSCs), which can be used for specifying real-time properties. After that, the *Inres* protocol is presented in Section 2.4. This protocol is used as an example case study throughout this thesis. The *Testing and Test Control Notation version 3* (TTCN-3), a language for specifying functional black-box tests, is explained in Section 2.5. Finally, this chapter is summarised.

2.1 Testing

Testing is one of the most important constituents of software quality assurance. It is an analytic means for assessing the quality of software [Wal01]. In [Mye79], G. Myers defines testing as "[...] *the process of executing a program with the intent of finding errors.*" However, errors in a program can also be revealed without execution by just examining its source code [FLS04]. This is usually referred to as *static testing* (in contrast to *dynamic testing* based on execution). Thus, [Bal98] regards testing more general as a means which aims at revealing errors in a program.

While [Mye79] refers to a *program* which is tested, a more general term for the subject of test is *item under test*. This might be a simple program, a single function, a group of software components, or a whole system, e.g. even a large distributed system.

Besides the granularity, the characteristics of an item under test may differ in further aspects. According to [Gra02], a classification of systems is possible with respect to the dimensions depicted in Figure 2.1: A system may be characterised concerning its communication with the environment, i.e. it may communicate either via message exchange or via (remote) procedure

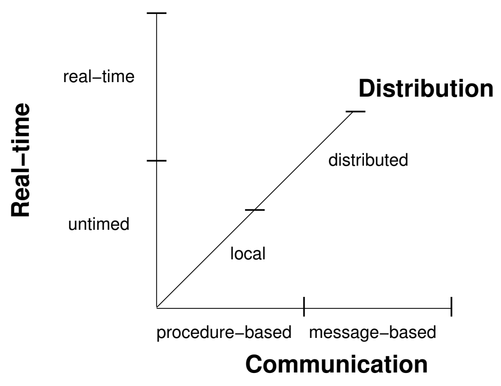


Figure 2.1: Dimensions of Systems

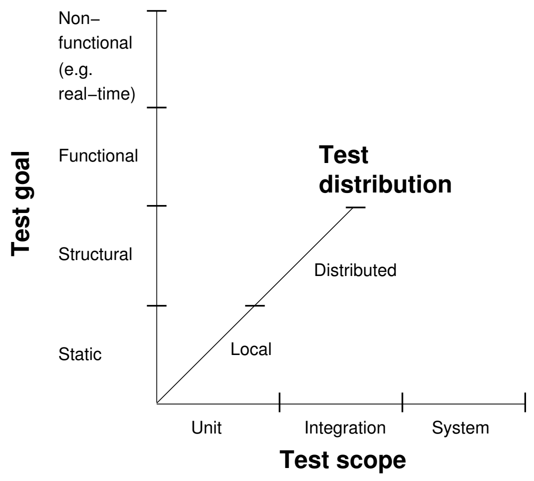


Figure 2.2: Dimensions of Testing

calls. Another distinction is possible with respect to real-time aspects: some systems are untimed, i.e. just their functional behaviour is important. For other systems' behaviour it is crucial that it adheres additionally to certain real-time requirements. A further dimension of systems is the distribution of their components: a monolithic system consists only of components which are local to a single node and share memory, while the components of a distributed system may be located on several remote nodes which communicate with each other without shared memory.

In a similar way, different kinds of tests may be distinguished. To some extent, the different kinds of tests result from the characteristics of the item under test. In Figure 2.2, different possible dimensions of testing are depicted.[1] The three different dimensions are as follows:

[1]Note that even more dimensions might be used for classifying software tests. For example, the communication aspect, which has been distinguished in Figure 2.1, has been

Test goal: The brief survey on definitions of testing already revealed that static and dynamic tests can be distinguished:

1. *Static testing:* Static tests assess an item under test without executing it. Static tests are able to locate defects in an earlier stage than dynamic tests and thus may reduce costs in contrast to defects which are revealed in later stages.

The remaining elements of this dimension are dynamic tests, for which a more close grained distinction is possible. It is dependent on the goal at which a test is aimed.

2. *Structural testing:* Structural test approaches have the goal to cover the structure (e.g. control or data flow) of an item under test during test execution. To achieve this, the internal structure of the item under test needs to be known. Therefore, another term for structural tests is *glass-box tests* [Mye79].
3. *Functional testing:* The goal of this type of test is to assess an item under test with respect to the functionality it should fulfil. Functional testing is based on the specification of the item under test. In contrast to structural tests, functional tests do not require any knowledge of internals of the item under test. Therefore, they are called *black-box tests* [Bei95].
4. *Non-functional testing:* Like functional tests, non-functional tests are usually performed against requirements contained in a specification. In contrast to pure functional testing, non-functional testing aims at the assessment of non-functional requirements. A variety of different non-functional properties exists, e.g. real-time. Examples of such properties are given in Section 2.2.

   Non-functional tests are usually black-box tests. Nevertheless, for retrieving certain information, like values of internal clocks, access to internals of the item under test may be convenient. In this case, such tests have to be regarded as *grey-box tests.*

Test scope: The *test scope* describes the granularity of the item under test which may vary as already described. Due to composition of the item under test, testing at different scopes may reveal different defects [Wey86, Wey88]. Since units are composed into larger groups of components which finally make the whole system, tests are usually performed in the following order of scopes:

subsumed, since for testing itself the actual communication mechanism does not matter. Nevertheless, a universal test language should support both types of communication to enable testing of both types of systems.

1. *Unit:* At the scope of unit testing, the smallest testable unit (e.g. a class in an object-oriented implementation or a module in a procedural language) is tested in isolation. By definition, a unit is not distributed.
2. *Integration:* A further kind of test is concerned with testing the integration of a strongly connected composition of units which does not form a whole system on its own.
3. *System:* The whole system is the scope of system tests. A complex system may be distributed and has usually different interfaces at which it can be accessed.

Test distribution: Not only the item under test may be distributed, but also the *tester* or *test system* itself can be characterised with respect to its distribution:

1. *Local:* A local test consists of just one test component located on a single node. The test is driven by the test component which accesses the item under test through one or more interfaces.
2. *Distributed:* A distributed test consists of several test components which may be distributed over several nodes. Thus, the whole test consists of concurrently running components which interface the item under test. To achieve a deterministic test, some coordination of the components is necessary.

It has to be noted that even a distributed system can be tested using a local tester and vice-versa. The different interfaces of a distributed item under test may be accessed from within one test component and a monolithic item under test may be accessed concurrently from several test components as well.

This thesis focuses on distributed black-box system and integration testing, i.e. distributed testing against requirements via public interfaces of the item under test. In particular, the emphasis is on non-functional testing of hard real-time properties. This is shown in Figure 2.3 (the projection to the coordinate planes is shown as shaded area). Since distributed testing may be restricted to local testing, also solutions for local testing are provided.

2.1.1 The Testing Process

Like software development in general, the quality of the testing procedure itself benefits from a process of systematic testing activities. Figure 2.4 shows the activities of a typical black-box testing process. Activities are depicted as rounded boxes, the artefacts which are input and output of the activities are depicted as rectangles.

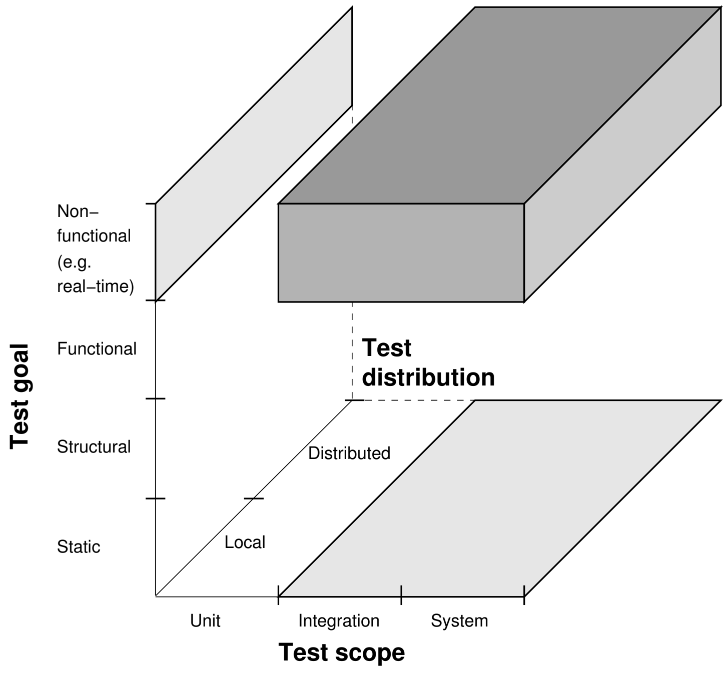


Figure 2.3: Area of Testing Considered in this Thesis

The overall goal of a black-box testing process is to obtain a test result, which indicates whether an *implementation*, which is the item under test, fulfils its *specification* or not.[2] This is achieved by defining *test purposes*. A test purpose describes an objective of testing which focuses on an individual requirement being part of the specification. Since a test purpose abstracts from the actions which are necessary for testing a requirement, corresponding *test cases* have to be developed. A test case is a detailed description of all test actions which need to be performed to achieve a test purpose. All test cases and their associated test data are grouped together into a *test suite*. A test suite can then be executed against the implementation. During *test execution*, a *test log* of the occurring test events is recorded. Finally, the test has to be *evaluated* and the *test result* is denoted by a *verdict* which indicates whether the implementation passed the test or failed.

2.1.2 Conformance Testing Methodology and Framework

The ISO/IEC standard 9646 *Conformance Testing Methodology and Framework* (CTMF) [ISO97b] is an example for an incarnation of a black-box testing process. Even though CTMF is intended for testing the functional behaviour of *Open Systems Interconnection* (OSI) [ISO97a] protocol entities

[2]However, since testing can only show the presence, never the absence of errors [Dij70], this cannot be guaranteed by testing.

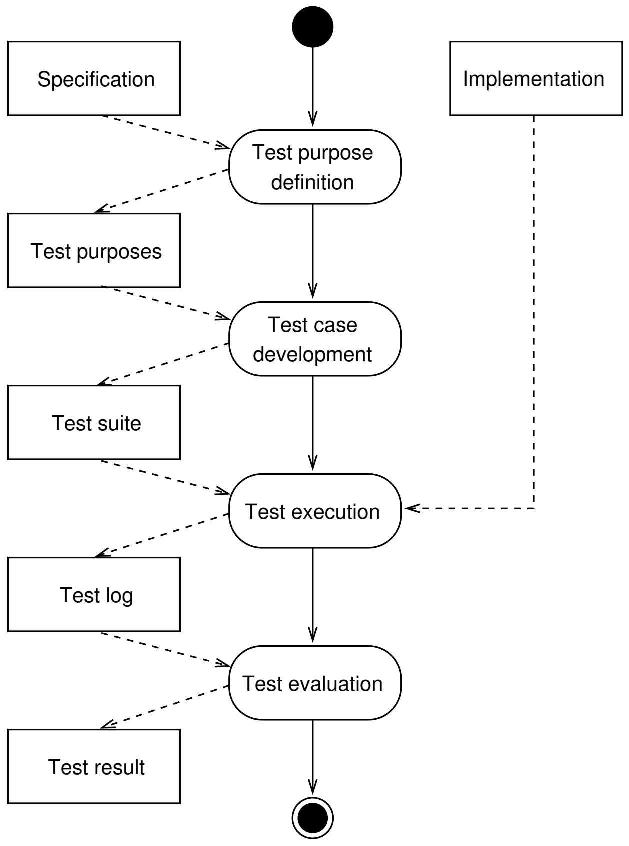
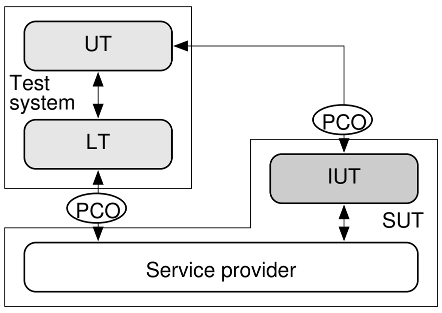


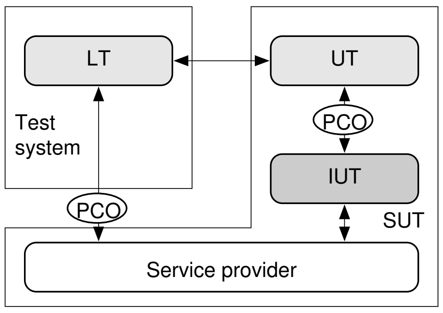
Figure 2.4: Activities of the Black-Box Testing Process

and thus uses a lot of OSI specific notions, it can be generalised and has thus been successfully applied to black-box testing of distributed systems in general [SR96, BW97, Anl98, Gec98]. Hence, those concepts of CTMF which are appropriate in the context of this thesis are presented in the following.

To allow the specification of portable tests which can be executed on different test systems, CTMF distinguishes between abstract and executable test cases or *Abstract Test Suite*s (ATS) and *Executable Test Suite*s (ETS), respectively. An ATS abstracts from test implementation details, e.g. special hardware equipment through which the item under test is interfaced. Thus, as part of test realisation, an ATS needs to be transformed into an ETS by adding the information which is specific to, e.g., the operating system and hardware of the test system, but also to the item under test (*Protocol Implementation eXtra Information for Testing* (PIXIT)). Then, the final ETS can be obtained by, e.g., compiling the ATS and the additional information into executable machine code.



(a) Local Test Method



(b) Distributed Test Method

Figure 2.5: Abstract Test Methods

Furthermore, CTMF introduces several *abstract test methods*, which define test architectures which are appropriate for testing entities of a protocol stack. A simplified view of the *local test method* and the *distributed test method* is given in Figure 2.5.

In CTMF, the protocol entity under test is denoted as *Implementation Under Test* (IUT), whereas the *System Under Test* (SUT) additionally comprises further elements, like the underlying service provider through which the lower layer boundary of the IUT is accessed. The *Lower Tester* (LT) is used to connect to the lower layer boundary through an underlying service. In contrast, the *Upper Tester* (UT) accesses the upper layer boundary of the IUT usually directly.

The interfaces through which the IUT is stimulated by sending messages and observed by receiving messages[3] are called *Points of Control and Observation* (PCOs), i.e. black-box testing can only be performed if the IUT is controllable and observable. A PCO is modelled as a FIFO queue for preventing that messages sent to and received from the IUT are lost.

The difference between the two depicted test methods is the distribution of the UT and LT *Test Component*s (TCs). As the names of the test methods suggest, the TCs of the *distributed test method* are distributed (the UT is considered to be part of the SUT) and thus need to be coordinated remotely.

2.2 Real-Time Properties and Testing

In Section 2.1, two different kinds of black-box testing have been classified: testing of functional and of non-functional requirements. In the following, the difference between both is addressed and real-time properties are discussed.

Black-box testing is performed against *requirements*. [Par91] gives two definitions of a requirement:

[3]In testing, sending and receiving is considered from the test system's point of view.

1. *A condition or capability needed by a user to solve a problem or achieve an objective.*

2. *A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. The set of all requirements forms the basis for subsequent development of the system or system component.*

Requirements or respectively the properties which are required can be divided into functional and non-functional ones. Functional requirements are associated with tasks or behaviours a system must support, while non-functional requirements are constraints on various attributes of the functional behaviour.[4] Thus, non-functional properties are always related to functional behaviour and do not exist on their own.

A coarse classification of requirements which does not claim to be complete is given in Figure 2.6. As depicted there, non-functional requirements can be subdivided with respect to the following properties: *real-time* properties; *reliability* properties like *Mean Time Between Failure* (MTBF) or robustness; *economic* properties, e.g. costs or maintainability; *usability* properties and *security* properties. In the remainder, this thesis focuses on real-time properties.

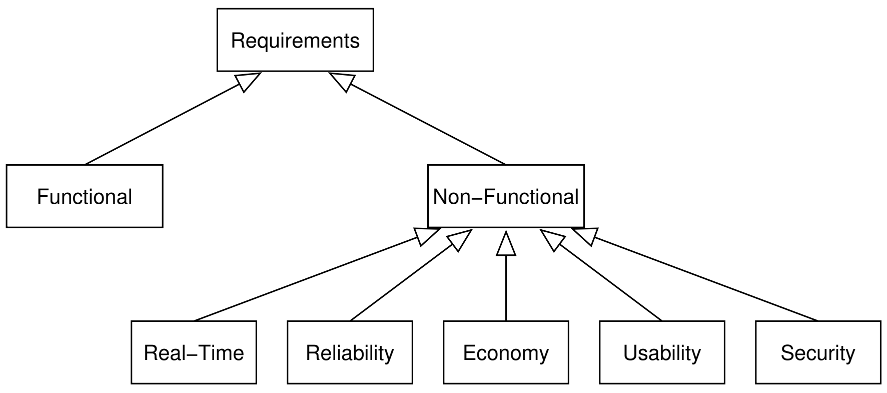


Figure 2.6: Hierarchy of Requirements

2.2.1 Real-Time Systems and Real-Time Properties

According to [lL90], a *real-time system* is "*a computing system where initiation and termination of activities must meet specified timing constraints.*"

[4]In the context of services, most of the non-functional properties are also denoted as *Quality of Service* (QoS).

Concerning the timing constraints a real-time system has to obey, *hard* and *soft* real-time properties can be distinguished [SR90]:

Hard real-time properties are timing constraints which have to be fulfilled in any case. If the hard real-time constraint is, e.g., on the duration of a response to a request, this means that a late answer is a wrong answer, because it is of no use anymore due to its lateness. An example is the autopilot of an aircraft, where violation of hard real-time constraints might lead to a crash. Mathematically, hard real-time properties can usually be described by simple equations or inequations.

Soft real-time properties are time constraints which need to be satisfied only in the average case or to a certain percentage. A late answer is still a valid answer. An example is video transmission where a delayed frame might either be displayed delayed or dropped which is not perceivable as long as no consecutive frames are affected. Even though soft real-time systems are easier to implement, the mathematical description of soft real-time properties is more complex, because it involves, e.g., statistical expressions.

Since statistics requires a series of samples, soft real-time properties are usually associated to streams of information, e.g. multimedia streams. Properties of streams are usually regarded as *performance* properties, whereas *hard real-time* properties relate to a small set of discrete events.[5] In this thesis, only hard real-time properties are treated.

Examples for hard real-time properties are the delays *latency* and *response time* but also *frequency* or its reciprocal cycle duration *periodicity* as well as *throughput* [ATM99a, ATM99b, ATM00, Buc96, IET90, IET91, IET98, IET99, IET02].[6] A detailed description of these hard real-time requirements can be found in Section 5.2.

Hard real-time requirements relate to discrete events like message reception or stimulating requests. Thus, hard real-time requirements can be expressed based on the relationship of time stamps of such events [Koy91]. A real-time requirement description consists of both, the actual real-time property and the functional behaviour on which the real-time property is imposed.

Besides informal prose specification of real-time requirements, formal techniques can be used to specify real-time requirements. [AH91] surveys some of them. However, those techniques did not prevail since most of them are based on temporal logic [Pnu77] which is hard to understand for practitioners. A more intuitive formal language for real-time requirements specification is described in Section 2.3.

[5]Though, the terms *real-time* and *performance* are also used synonymously [Jai91].

[6]Since throughput and periodic events involve repeated behaviour, both may also occur as soft real-time properties—though, with statistical constraints instead of hard bounds.

2.2.2 Functional vs. Real-Time Testing

Functional testing is concerned with assessing the functional behaviour of an item under test independently of any timing aspects. While the temporal (i.e. causal) order of events is regarded, the exact timing of the events is not considered. Functional testing can be used for assessing both real-time and non-real-time systems.[7]

In contrast, real-time testing assesses the real-time properties of an item under test by considering the points in time when test events are exchanged via the PCOs. It has to be noted that this does not only apply for observations. Also the stimuli which are sent by the test system may be subject of real-time requirements, since an item under test might also assume certain real-time properties from the environment, which for testing is provided by the test system.

Since non-functional properties like real-time properties are always related to functional behaviour, they cannot be tested on their own, but require a test case which involves also the associated functional events for stimulating and observing the item under test. Thus, a prerequisite of real-time testing is that functional testing has been performed and the item under test successfully passed its functional test.

An important requirement on testing is that tests shall be deterministic, i.e. *reproducible*. Otherwise, if the test verdict changes for each execution of the same test case, the outcome of a test run would be of no relevance. While reproducibility is usually quite easy to achieve for non-distributed pure functional testing, this may be a problem for distributed testing. In this case, race conditions of concurrent components may lead to different orders of events. This has to be considered during test case development, e.g. by appropriate synchronisation of the concurrent components.

For real-time testing, reproducibility is a severe problem [Sch93]. The problem occurs if the item under test delivers a real-time performance which is close to the limit of the actual real-time requirement, i.e. it is observed that the real-time property is sometimes just below and sometimes just beyond that limit. In this case, it is difficult to decide whether the item under test really failed or whether the test system is in fact at its limits and thus responsible for non-deterministic observations.[8] This can be avoided if the test system itself obeys two requirements: Firstly, the time resolution of the clock used to assess the test item's real-time property is at least in the same order of magnitude as the real-time requirements which are subject of

[7]Even though a non-real-time system or its corresponding functional test might involve timers, e.g. for timeout provocation or detection, this is not considered as a real-time property. Such timers are merely used to assure within a reasonable time frame the absence of an event which is hence in fact a functional property.

[8]In worst case (i.e. the test system is far too slow), a test may be deterministic in the sense that it yields always the wrong verdict.

test. Secondly, the test system in general has to be fast enough to send the stimuli and make the observations in time.

Hence, to be able to execute real-time test suites, an important prerequisite is that the test system is fast enough. This can, e.g., be evaluated by benchmarking the test system itself [HKN01, DST04].

Furthermore, if both distributed testing and real-time testing are combined, clock synchronisation is an important issue. In this case, the clocks of the distributed test components which are used for assessing the test item's real-time properties are running in parallel. Thus, if real-time requirements are imposed on events which are observed on distinct distributed test components, it is only valid to compare the obtained time stamps, if the clocks of the distributed test components are synchronised.

In total, for real-time testing one has to be aware that not only the SUT is a real-time system, but also the test system itself forms a (distributed) real-time system. Hence, for the implementation of a real-time test suite, real-time support from a real-time operating system is required, too. However, low-level test implementation is out of the scope of this thesis.

Literature on implementing real-time tests [Sch93] often discusses probe effects, i.e. the influence of instrumenting an item under test with monitors for observing certain events. In fact, this is only relevant for white- or grey-box tests, not for black-box tests as considered in this thesis. Nevertheless, observations are also necessary for black-box testing. This is performed by the test system via the PCOs, which have thus also to be taken into account for real-time black-box testing. This is related to the previous considerations on the speed of the test system. Hence, the test system should be fast enough to deal with the queues of the PCOs in a way that events for which real-time requirements apply are not queued but immediately processed. Using black-box testing, only the time points when events occur at a PCO are observable, not when a message leaves or enters the actual SUT. Though, this is not a restriction since in real world, the real-time properties of the SUT are perceived at the same location as the PCOs, i.e. not inside, but just outside the SUT.

Since the concepts of CTMF are proven for functional testing, it is desirable to reuse or extend and transfer them also to the area of real-time testing. For example, even though the abstract test methods itself can only be applied in simple cases for distributed real-time testing, the constituents of the CTMF test architectures can be reused. Nevertheless, e.g., for assessing the real-time behaviour of an SUT at different load situations, additional components for generating load might be required in a real-time test architecture. In principle, this can be achieved by an ordinary test component which generates the load. Detailed discussions of test architectures which are suitable for distributed real-time black-box tests can be found in [dMHB$^+$91, SSR97, WG97, WG99].

Moreover, CTMF defines a language for the specification of abstract conformance tests. However, this language is only intended for pure functional black-box tests. A more powerful successor of that test language, the *Testing and Test Control Notation version 3* (TTCN-3) is described in Section 2.5. Based on TTCN-3, a language for specifying distributed black-box real-time tests is introduced in Section 3.

2.3 Message Sequence Charts

This section gives an introduction into *Message Sequence Chart*s (MSCs), a trace language which is used in this thesis for formal real-time test purpose description (Section 4) and for capturing real-time requirements using *Real-time Communication pattern*s (RTC-patterns) (Section 5).

MSC is standardised by the *International Telecommunication Union* (ITU) Recommendation Z.120 [ITU99b].[9] It is a formal graphical language describing the flow of messages inside a distributed system. Besides the graphical language, a machine readable textual format, which allows exchange and processing by tools, is defined.

Z.120 defines three different types of diagrams: plain *MSCs*[10], *High-level Message Sequence Chart*s (HMSCs) and *MSC documents*. Plain MSCs describe the actual message exchange, whereas HMSCs can be used to compose more complex traces by combining MSCs. Plain MSCs, and HMSCs can be collected in MSC documents.

2.3.1 Plain Message Sequence Charts

Basically, a plain MSC describes the flow of *messages* between the *instances* or the environment of a system. For example, the MSC named Simple shown in Figure 2.7 includes three instances. Instances PCO1 and PCO2 are both of type PCO, instance System is of type SUT. The *instance name* is written inside the instance head, the *type* of an instance is written above. However, the type of an instance may be omitted. In this case, just the instance name is written above the instance head (cf. Figure 2.8).

Furthermore, the depicted MSC Simple specifies that message m1 containing the parameter value is sent from instance PCO1 to System and subsequently, message m2 with the same parameter is sent from System to PCO1 and PCO2. Both, sending and receiving a message are considered as individual events. For the order of events, MSC uses a partial order semantics: the events along an instance axis are totally ordered from top to bottom. (If

[9]Currently, a new version is being standardised. However, it only contains minor changes. Most of them are part of [ITU01] to which the author of this thesis has contributed.

[10]The term *MSC* is used both for a diagram (either plain MSC or HMSC) written in the MSC language and for the language itself.

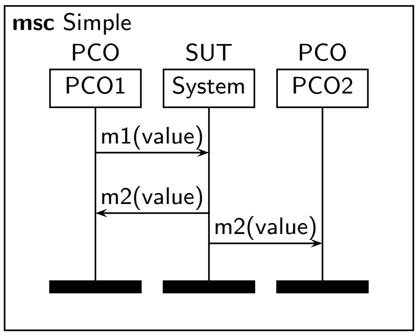


Figure 2.7: A Simple MSC

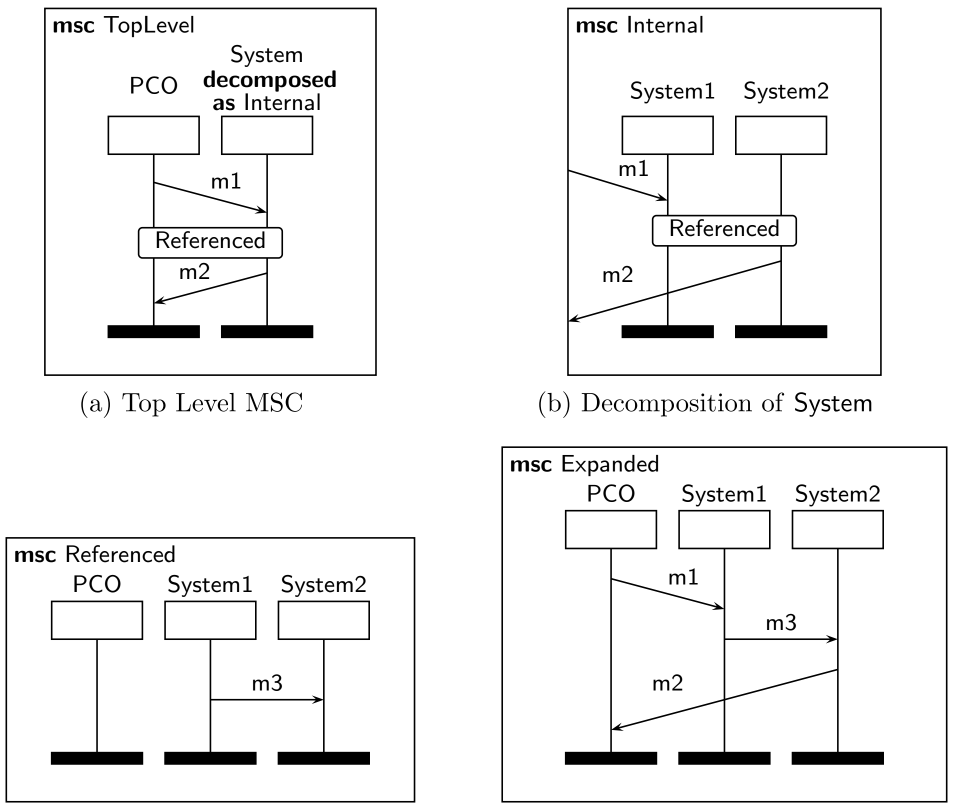
that is not desired, *co-regions* can be placed along an instance axis to allow a concurrent interleaving of events.) However, between different instances, there is no order except for the one imposed by the fact that a message cannot be received before being sent. In particular, comparing the vertical positions of events on different instances gives no indication on any ordering. Hence, the MSC in Figure 2.7 describes in fact three different traces:

1. send m1, receive at System, send m2 to PCO1, receive at PCO1, send m2 to PCO2, receive at PCO2.
2. send m1, receive at System, send m2 to PCO1, send m2 to PCO2, receive at PCO1, receive at PCO2.
3. send m1, receive at System, send m2 to PCO1, send m2 to PCO2, receive at PCO2, receive at PCO1.

In addition to asynchronous message sending and reception, further kinds of events are possible, e.g. *procedure-based communication* (i.e. remote procedure call), *actions*, *instance creation* and *termination*, but also, e.g., *lost* and *found messages*. Moreover, it is possible to attach *comments* to events.

Besides these basic concepts, MSC provides several means for structuring diagrams. Abstraction from and refinement of behaviour is supported in the MSC language by *decomposed instances* and *references*. The decomposition mechanism allows to refine the behaviour of an instance. This is shown in figures 2.8a and 2.8b. The keywords **decomposed as** followed by the name Internal in the header of instance System (Figure 2.8a) indicate that System is an abstraction of the behaviour specified by MSC Internal (Figure 2.8b). To obtain a well formed MSC, the communication events contained in the referenced MSC must be a superset of the events attached to the decomposed instance. Therefore, in the example, the reception of message m1 and the sending of message m2 can be found in MSC TopLevel and MSC Internal.

The MSCs in figures 2.8a and 2.8b also contain reference symbols, which both refer to the MSC Referenced. The semantics of a reference symbol is given by the referenced MSC, i.e. the behaviour of the referenced MSC



(a) Top Level MSC (b) Decomposition of System

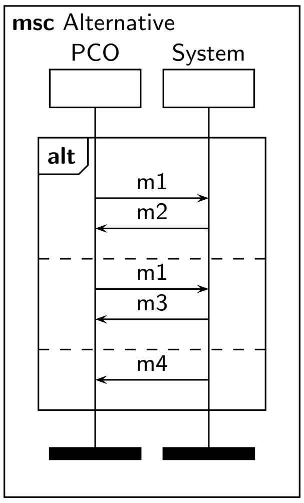
(c) MSC Referenced in (a) and (b) (d) Expansion of MSC (a) Using (b) and (c)

Figure 2.8: Abstraction and Refinement in MSC

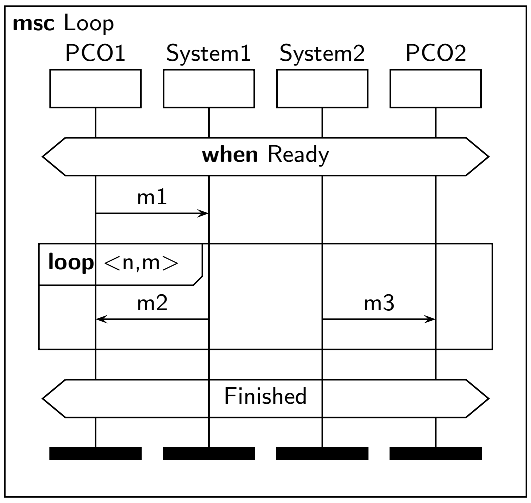
replaces the reference. For well-formedness, the instances of a referenced MSC must be a superset of the instances covered by the reference symbol in the referring MSCs. By applying the rules for decomposed instances and references, the MSC TopLevel can be expanded to the MSC Expanded shown in Figure 2.8d.

For specifying complex behaviour in a compact manner within one diagram, the MSC language provides *inline expressions*. They allow the description of *looped*, *alternative*, *optional*, *exceptional*, and *parallel* behaviour. Examples for alternatives and looped behaviour are given in the following paragraphs.

Figure 2.9a depicts an MSC in which an **alt** inline expression is used to specify three possible alternative behaviours: In the first case, message m1 is sent from PCO to System and subsequently, message m2 is sent from System to PCO. The next case is separated by a dashed line and contains a similar behaviour like before, except that message m3 is sent instead of message m2. The last case consists just of sending message m4 from System to PCO. This **alt** inline expression does not specify under which conditions which



(a) Alternative Behaviour



(b) Repeated Behaviour

Figure 2.9: MSC Inline Expressions and Conditions

alternative case is taken. Instead, it just describes that all alternatives are valid traces. MSC uses a late choice semantics to decide which alternative actually occurred.

In Figure 2.9b, an MSC is shown which contains a **loop** inline expression to specify n to m repetitions of the behaviour framed by the inline expression. The second operand in angle brackets may also be omitted which is then interpreted as exactly n iterations. The value **inf** can be used to denote infinity. If both operands are omitted, this is interpreted as <1,**inf**>.

The events contained in an inline expression or referenced by a reference symbol are inserted into the surrounding MSC using weak sequential composition. In Figure 2.9b, messages m3 may, e.g., be sent and received prior to message m1. Also the passes of a loop are connected by weak sequential composition, thus in Figure 2.9b arbitrary interleavings of the n to m occurrences of messages m2 and m3 are possible.

Furthermore, MSC supports *conditions* for describing states either locally to a single instance, spanning over several instances, or globally for all instances. Conditions can either be set or used as guard. The latter restricts the execution of events until a condition is set. Guards are indicated via the keyword **when**. Either symbolic condition names or boolean expressions are allowed as guarding condition.

A global guarding condition is depicted at the beginning of MSC Loop in Figure 2.9b, where all instances must be in state Ready before execution continues. A setting condition is shown at the bottom of that MSC, where a condition labelled Finished is set. Since the condition symbol covers only instances PCO1 and PCO2, that condition is a non-global one.

2.3.2 High-Level Message Sequence Charts

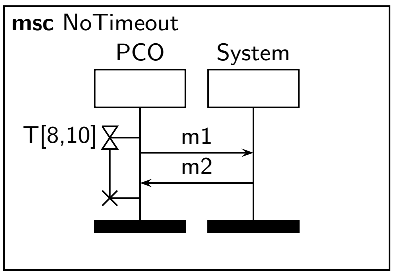
An HMSC is a directed graph which describes how MSCs can be combined into larger scenarios. A node of the graph is either a start node, an end node, a connection point, a global condition, a reference symbol, or a parallel frame. An HMSC abstracts from details like instances and events.

An example is shown on Page 26 in Figure 2.19: The HMSC InresScenario starts in the global state disconnected. Then, it proceeds with the behaviour of the referenced MSC ConnectionEstablishment. Since, in principle, connection establishment might either be successful and lead to a connected state or fail and remain in a disconnected state, the HMSC proceeds only if the guarding condition connected is enabled. Then, an arbitrary number of DataTransfer is performed. The iteration is either terminated with the behaviour of ConnectionRelease or with the behaviour of MSC DataTransferFailure. (Like for alternative behaviour in plain MSC, just possible behaviours are specified by an HMSC and late choice semantics is used to determine which of the possible branches is actually chosen.) Either branch terminates with setting the global state to disconnected.

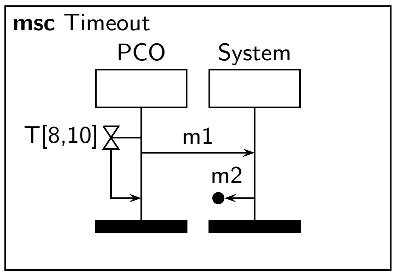
2.3.3 Time Constructs

MSC provides timers and time annotations for dealing with time. A clock which is global to all instances is assumed. An MSC can be parameterised with respect to the data language it uses in, e.g., message parameters, loop boundaries, but also in expressions which are used in time constructs. No assumption is made whether the time domain is dense or discrete, this is dependent on the data language. Furthermore, the default time unit of the data language is used as time unit in MSC. Since in this thesis MSC is used in combination with TTCN-3, it is assumed that the data language of TTCN-3 is used and thus, time is represented as floating point number with time unit *seconds*.

In MSC, *timers* can be placed along an instance axis to specify *start*, *expiration* and *reset* of a timer. A timer has a name and an interval which specifies the minimal and maximal duration after which a timer expires.

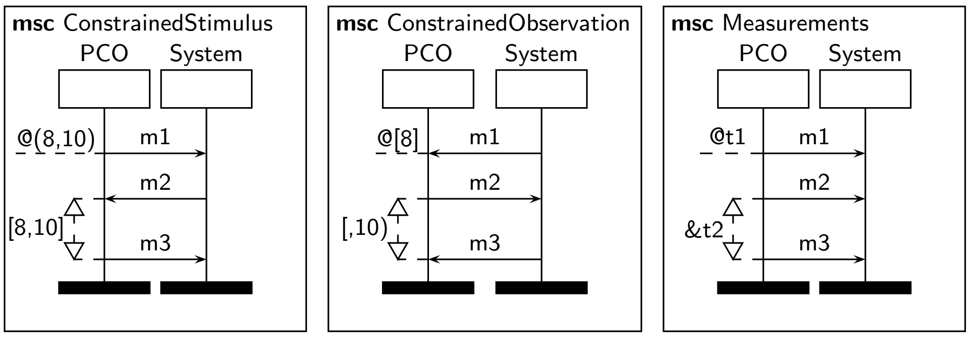


(a) No Timeout



(b) Timeout

Figure 2.10: MSC Timer Constructs



(a) Time Constraints (b) Time Constraints (c) Time Measurements

Figure 2.11: MSC Time Constraints and Measurements

Figure 2.10a depicts the start of a timer T before sending message m1 and its cancellation after receiving message m2. The example in Figure 2.10b describes that a timer expires after a duration between 8 and 10 seconds. Such timers are usually applied to detect the absence of other events, e.g. the arrival of message m2 which may have been lost as shown in Figure 2.10b.

MSC timers shall not no be used for specifying real-time requirements, because during further usage of an MSC for system implementation, it cannot be distinguished whether a timer translates into functional behaviour, i.e. an actual timer inside the implementation to detect timeouts or whether it was just used to describe real-time requirements.

Moreover, timers in MSC have further deficiencies: The events of timer start and expiration may not span over distinct instances. Furthermore, since timer start and expiration are events on their own, an arbitrary amount of time may pass between starting a timer and the occurrence of events surrounding a timer like sending or receiving a message.

A better means for specifying hard real-time properties are MSC time annotations.[11] They can be attached to events like sending or receiving a message. Examples of how the most common hard real-time requirements can be expressed using MSC can be found in Section 5.2.

MSC distinguishes between *time constraints* and *time measurements*. Time constraints are shown in figures 2.11a and 2.11b. They can be either absolute (indicated by @), i.e. refer to the absolute time of occurrence of a single event, or relative, i.e. constrain the duration between two events. In Figure 2.11a, sending of m1 has to occur inbetween absolute time of 8 and 10 seconds, and the time difference between receiving m2 and sending m3 at instance PCO is restricted to be between 8 and 10 seconds.

The possible values of a time constraint are specified using intervals. The interval boundaries may be open or closed. An open boundary is indicated by a parenthesis, i.e. '(' or ')', and a closed boundary is defined by a square

[11]However, MSC is not well suited for expressing soft real-time requirements.

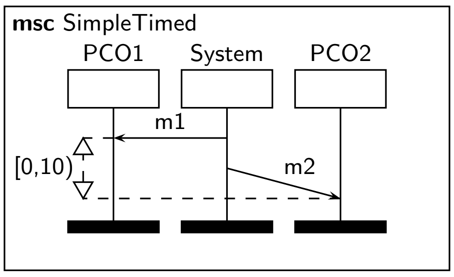


Figure 2.12: Partial Ordering and Time Constraints

bracket, i.e. '[' or ']'. An omitted lower bound is treated as zero (Figure 2.11b), an omitted upper bound as ∞. A single value in square brackets represents an interval which contains just that single element (Figure 2.11b).

Time measurements are used in Figure 2.11c. Either the absolute time when a single event occurs or the relative delay between two events may be measured. The value of a measurement is stored in a variable which is local to the instance that owns that variable. For example, in Figure 2.11c, the time point of sending message m1 is stored in variable t1 and the relative time between sending m2 and m3 is measured and stored in variable t2.

Time constraints can also be attached to the beginning and end of inline expressions and references. (This includes also references in HMSC.) In this case, the constraint refers to the first or last event respectively which occurs inside the inline expression or reference respectively. In cases of doubt, those events are determined using late choice semantics.

When using time annotations, the partial order semantics of MSC still applies. I.e., in Figure 2.12, the order in which m1 and m2 are received is not fixed. Thus, for the time points t_{m1} of receiving message m1 and t_{m2} of receiving message m2, either $0 \leq t_{m1} - t_{m2} < 10$ or $0 \leq t_{m2} - t_{m1} < 10$ holds depending on the actual occurred ordering.[12]

While relative time constraints for two explicitly given events can be easily expressed using MSC, it is not possible to express the frequency or respectively the cycle duration of periodic events. The reason is that standard MSC does not allow to attach relative time constraints to events which appear graphically only once in a diagram, but actually occur in the corresponding trace several times due to iterations of a loop.

Thus, MSC extensions for HMSC [ZK02] and plain MSC [Neu00] have been suggested. The notation for the extension of plain MSC is shown in Figure 2.13. It allows to attach relative time constraints also to a pair of events which spans over adjacent repetitions of a loop. The semantics of this extension can be obtained by unrolling that loop as shown in Figure 2.14.

[12]The forthcoming MSC standard will support *unidirectional* relative time annotations, which apply only when events occur in a desired order.

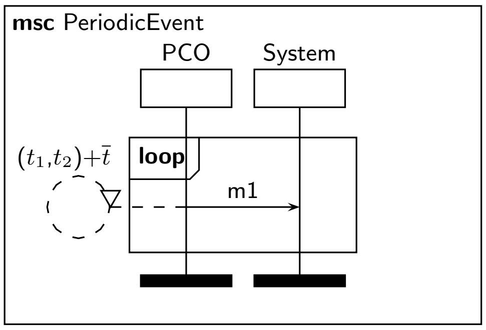


Figure 2.13: MSC Extension for Periodic Events

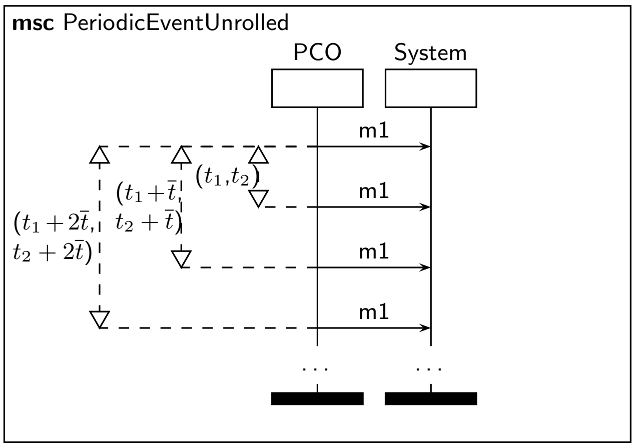


Figure 2.14: MSC from Figure 2.13 with Unrolled Loop

2.4 The Inres Protocol Case Study

Inres [Hog89, Hog91] is a sample protocol and service which has been created for educational and research purposes. Nevertheless, it possesses many concepts of a real-life protocol for communication systems which can be tested using distributed black-box testing. Even though Inres was designed with no real-time properties in mind, it is used as a case study throughout this thesis. For this purpose, real-time requirements are imposed on some message exchanges. This is done where appropriate in the individual chapters of this thesis. In the following, just the pure functional properties of Inres are presented.

Inres provides a reliable, connection-oriented and asymmetric service, i.e. only one side (the *Initiator user*) can initiate connection establishment and send data to the other side. The other side (the *Responder user*) can release a connection[13] and receive data.

[13]Note that in secondary literature, many Inres-based case studies allow also the Initiator user to release a connection.

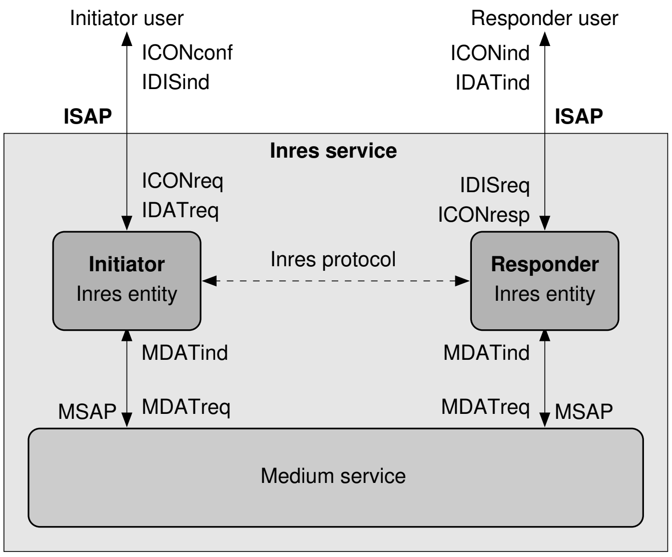


Figure 2.15: Structure of Inres

For delivering its service, Inres transfers data units over an underlying *Medium* service. The Medium service is assumed to be unreliable in the sense that data units may get lost, but are not corrupted or duplicated.

The structure of Inres is shown in Figure 2.15. The Inres protocol entities are the Initiator and the Responder Inres entities. They communicate with each other using the Medium service, which offers its service via the *Service Access Point* (SAP) MSAP. The service primitives at that SAP are MDATreq for requesting the transfer of data and MDATind for indicating the arrival of data to the Medium service user.

Inres offers its service via the SAP ISAP. The establishment of a connection can be requested by a user of the Initiator protocol entity using the service primitive ICONreq. This is shown Figure 2.16. Initially, it is assumed that the participants are in a disconnected state. The connection request results in a CR *Protocol Data Unit* (PDU) being sent to the peer Inres protocol entity. For delivering CR to the Responder, the Medium service is used. The Responder indicates the received connection request to the Responder user via the ICONind service primitive. Both, Initiator and Responder are now waiting for a confirmation. The response of the Responder user (ICONresp) is transferred to the Initiator as a CC PDU via the Medium. If this is detected at the Initiator, the connection confirmation ICONconf is sent to the Initiator user and the participants are in a connected state.

Once a connection has been established, data can be transferred. The corresponding sequence of messages is presented in Figure 2.17. The transport of some data can be requested using the IDATreq(data) service primitive. Internally, the Inres protocol uses sequence numbers to identify its PDUs.

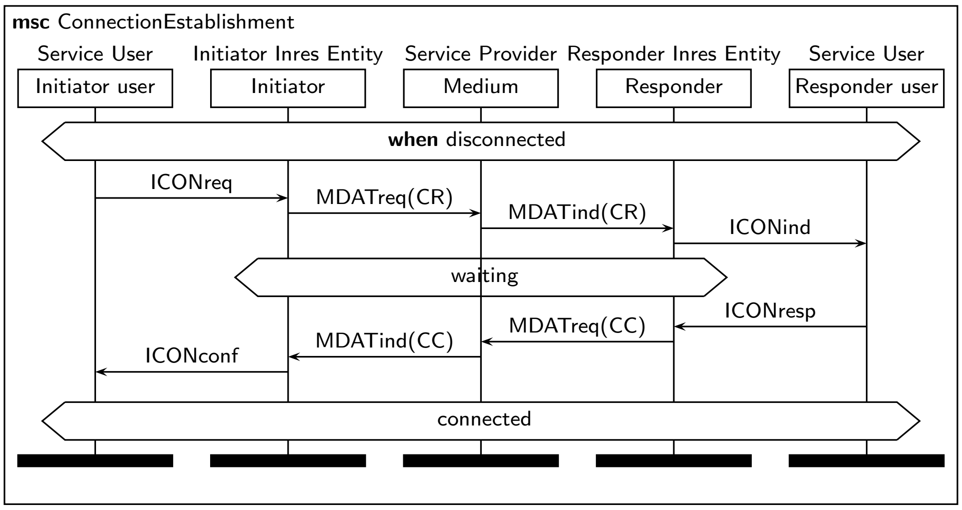


Figure 2.16: Inres Connection Establishment

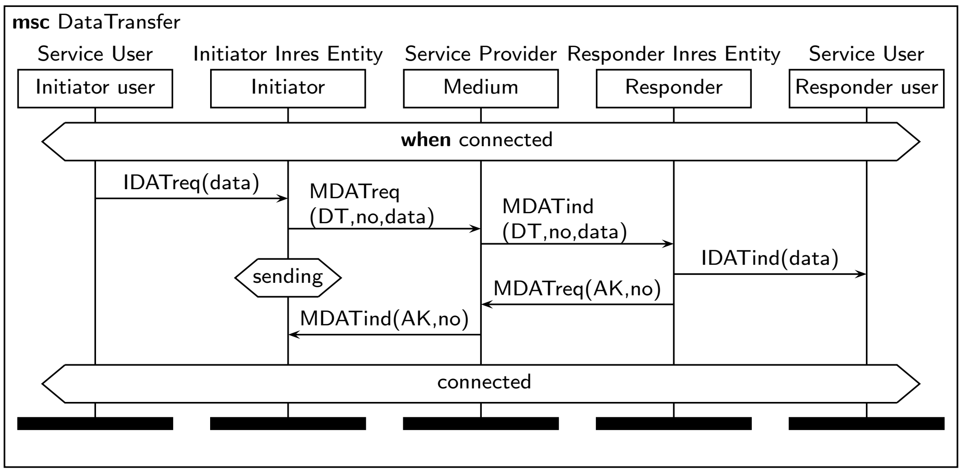


Figure 2.17: Inres Data Transfer

Thus, the DT PDU which is sent by the Initiator via the Medium to the Responder contains not only data as parameter but also the sequence number no. The Initiator is waiting in state sending for the acknowledgement. After the data PDU has been received by the Responder, an IDATind is indicated to the Responder user. Furthermore, the Responder acknowledges the reception of data to the Initiator with a PDU containing AK and the sequence number. If that acknowledgement was successfully received by the Initiator, all participants remain in the connected state.

The MSC DataTransfer shows only the case of successful data transmission. However, since the Medium service is unreliable, Inres PDUs may also get lost. To detect this, the Initiator entity uses a timer and resends PDUs

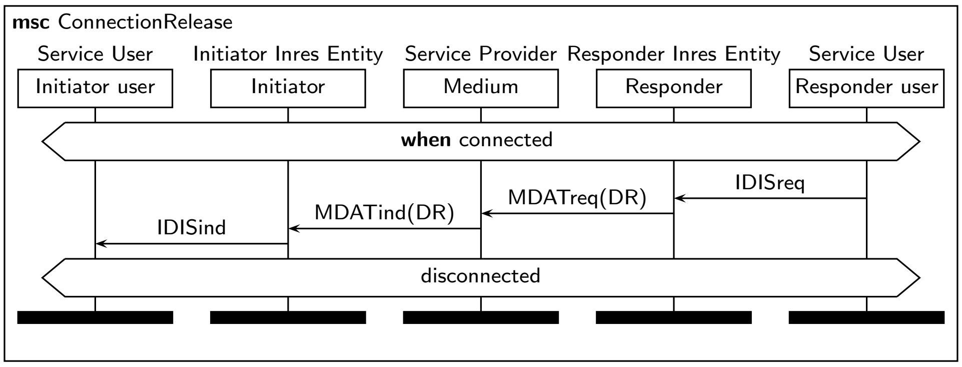


Figure 2.18: Inres Connection Release

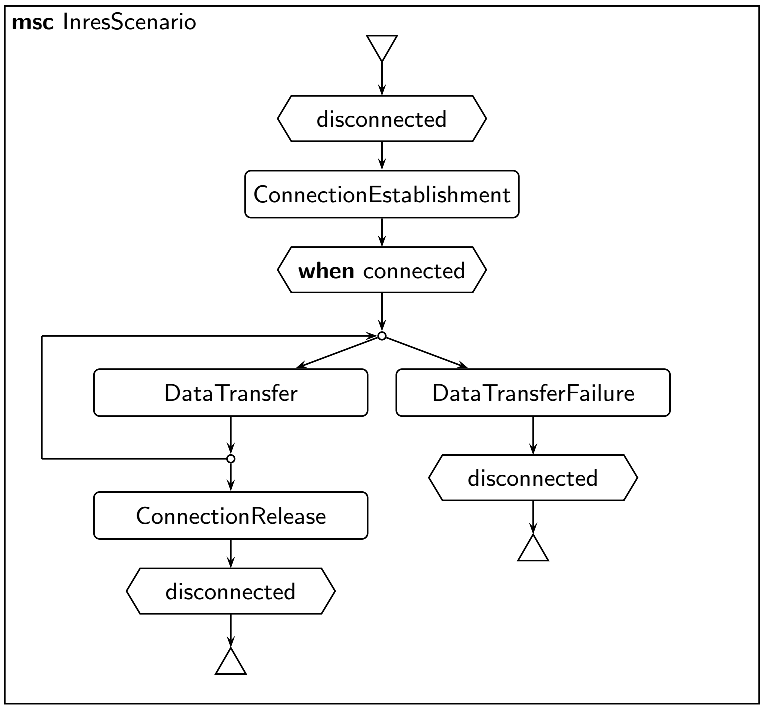


Figure 2.19: High-level MSC Describing Inres Scenarios

after a timeout up to 4 times. If retransmission fails after the 4^{th} try, the Initiator entity releases the connection by itself. The same holds for connection establishment, which may also fail due to transmission problems.

In the normal case, a connection is released by the Responder user. This is straightforward as depicted in Figure 2.18. The IDISreq service primitive is forwarded as a DR disconnection request PDU. After the reception of that PDU, the Initiator indicates this to its user with the IDISind service primitive. Finally, the participants are in the disconnected state.

The HMSC in Figure 2.19 gives an impression of possible scenarios for using the Inres service. In particular, once a connection has been successfully established, it is possible to transfer data several times. However, a failure during data transmission leads to a disconnected state. (The behaviour of MSC DataTransferFailure is not shown, but has been described informally.)

2.5 The Testing and Test Control Notation

This section describes the *Testing and Test Control Notation version 3* (TTCN-3) [ETS02a, ETS03b, GHR$^+$03]. Unlike the *DejaGnu* test library [SE04] which is based on the *Tool Command Language* (TCL) [Ous04] and the *JUnit* test framework [GMB04] which is based on *Java* [Sun04], TTCN-3 is a language designed especially for testing. TTCN-3 has been developed and standardised by the *European Telecommunications Standards Institute* (ETSI). Even though it is a successor of the special purpose *Tree and Tabular Combined Notation* (TTCN) which is part of CTMF, TTCN-3 supports various kinds of tests, with an emphasis on black-box tests for distributed systems [SPVG01, Sza02, EYL02, SS03]. TTCN-3 can not only be used for test specification but due to the broad tool support [Dan04, DaV04, Tel04, Tes04, Ope04] also for test implementation.

As depicted in Figure 2.20, TTCN-3 consists of a textual core notation [ETS02a] and several presentation formats like the MSC-based *Graphical Presentation Format for TTCN-3* (GFT) [ETS03a] or the *Tabular Presentation Format* (TFT) [ETS02b]. Furthermore, TTCN-3 allows —in addition to its own data model— to use data described by other languages. This eases testing of systems which were implemented using, e.g., the *Abstract*

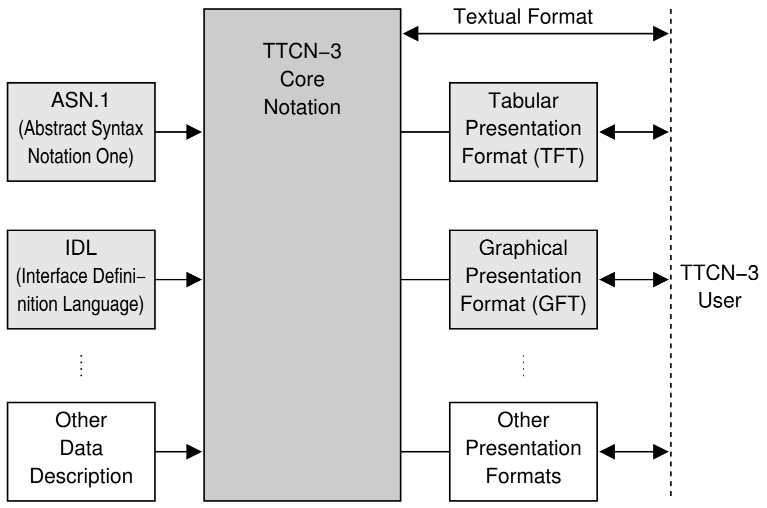

Textual Format
TTCN–3 Core Notation
ASN.1 (Abstract Syntax Notation One)
Tabular Presentation Format (TFT)
IDL (Interface Definition Language)
Graphical Presentation Format (GFT)
TTCN–3 User
Other Data Description
Other Presentation Formats

Figure 2.20: Overall Picture of TTCN-3

Syntax Notation One (ASN.1) or the *Common Object Request Broker Architecture* (CORBA) [OMG04a] *Interface Definition Language* (IDL).

In the following, the concepts of TTCN-3 and its core notation will be explained by two test suites for functional testing of an Inres Initiator protocol entity implementation. The test system has to play the role of an Inres Responder entity and of an Initiator user, since the SUT is accessed via the *Service Access Point*s ISAP and MSAP.

For communicating with the SUT, types and values of the exchanged messages must be defined. This will be demonstrated in the next section. Afterwards, in Section 2.5.2, it will be explained how test behaviour is described in TTCN-3. Since TTCN-3 uses a syntax which is similar to ordinary programming languages like, e.g., *C++* [ISO98], it is assumed that the reader is familiar with common constructs like comments, (structured) data types, assignments or parentheses for delimiting blocks.

2.5.1 Data and Type Definition

TTCN-3 uses *modules* as a top-level structure. Figure 2.21 shows the module inresUserDefinitions. It contains only the definition of the new type UserPDU (Line 2) and a constant value (Line 3). The definitions in this module describe the payload data which can be transferred by an Inres user through the Inres service. In this case, floating point numbers are transferred.

```
1 module inresUserDefinitions {
2   type float UserPDU; // Type of actual data to transfer
3   const UserPDU somePayload:=0.42; // Payload of type UserPDU
4 }
```

Figure 2.21: TTCN-3 Module for Defining Data of the Inres User

Definitions inside modules can be *imported* by other modules. Figure 2.22 demonstrates in lines 2–4, how module inresDefinitions imports selectively the definition of type UserPDU from module inresUserDefinitions.

For further sub-structuring of modules, TTCN-3 allows *grouping* of definitions. Figure 2.23, which is —as indicated by the line numbers— a continuation of module inresDefinitions, provides an example for the **group**

```
1 module inresDefinitions {
2   import from inresUserDefinitions {
3     type UserPDU; // Type of actual data to transfer at User Level
4   }
```

Figure 2.22: Importing Definitions from the Inres User Module

```
 5  group InitiatorUserDefinitions {
 6    // The messages
 7    type record ICONreq {};
 8    type record IDATreq { UserPDU iData };
 9    type record ICONconf {};
10    type record IDISind {};
11
12    // The port type
13    type port InresSAP message {
14      out ICONreq, IDATreq;
15      in ICONconf, IDISind;
16    }
17  }
```

Figure 2.23: Definitions for the Initiator and Initiator User Communication

statement. Grouping has no effects on namespaces, but, e.g., an **import** statement may refer to a whole group. Furthermore, groups aid the human reader to identify elements which are related to each other, especially if a descriptive name for the group (as in Line 5) is chosen.

In group InitiatorUserDefinitions, messages which are exchanged between the Inres Initiator entity and the Initiator user are defined (lines 7–10). Message types are defined using ordinary data types, e.g. an empty record for messages without parameters. In this case, just the type information itself is used to carry information. For messages with parameters, field elements of a record may be used for carrying the parameters. For example, in Line 8 of Figure 2.23, the IDATreq message has a payload of type UserPDU.

Furthermore, *port* types can be defined to specify communication points with respect to type and direction of messages which are exchanged through them. In lines 13–16, InresSAP is defined as a port type. It will be used later-on as *Point of Control and Observation* (PCO) for interfacing the ISAP *Service Access Point*. The **message** keyword declares that the port is used for message-based communication.[14] The **out** keyword specifies that the messages ICONreq and IDATreq can be sent by the test system, whereas **in** denotes that messages ICONconf and IDISind can only be received.

Figure 2.24 shows a group with definitions for the communication between an Inres Responder and the Medium service. At that interface (MSAP), only the two messages MDATreq and MDATind are exchanged (definitions in lines 38–45). The payload of these messages is more complex. It is of type InresPDU, which is defined as a record type in lines 31–35. The last two fields of this record are marked **optional**, which means that they may

[14]TTCN-3 supports also procedure-based communication. However, the corresponding syntax is not described in this thesis, since the semantics does not differ significantly from message-based communication.

```
18  group ResponderDefinitions {
19    type enumerated InresPDUType { CR(1), CC(2), DR(3), DT(4), AK(5) };
20    type enumerated SequenceNumber { zero(0), one(1) };
21
22    function toggle(SequenceNumber number) return SequenceNumber {
23      if (number==zero) {
24        return one;
25      }
26      else {
27        return zero;
28      }
29    }
30
31    type record InresPDU {
32      InresPDUType iPDUType,
33      SequenceNumber seqNo optional,
34      UserPDU iData optional
35    }
36
37    // The messages
38    type record MDATreq { InresPDU mData };
39    type record MDATind { InresPDU mData };
40
41    // The port type
42    type port MediumSAP message {
43      out MDATreq;
44      in MDATind;
45    }
46  }
```

Figure 2.24: Definitions for the Responder and Medium Communication

be omitted. While the type of the last field (UserPDU) has been previously imported (Line 3 of Figure 2.22), the first two fields are defined as enumeration types in lines 19–20. The sequence number is used by the Inres Initiator implementation to identify InresPDUs. Since sequence numbers are binary, they are changed by toggling. To enable a test case to check whether the IUT changes a sequence number correctly, the inresDefinitions module provides not only the type definition but also a *function* to toggle a value of type SequenceNumber. As depicted in lines 22–29, in TTCN-3, this can be written down in a style of an ordinary programming language. (More detailed examples of behaviour specification are provided in Section 2.5.2.)

The group in Figure 2.25 contains *template* definitions. Templates ease specification of test data which is sent or received. While sending requires concrete values, receiving of data may be eased by the possibility to specify value ranges or to use wildcards. Templates allow to define fully specified values as well as a set of expected values based on matching mechanisms, like regular expressions for strings or using “?” to describe any value.

```
47     group TemplateDefinitions {
48       template MDATind ConnectionRequest:={
49         mData:={ iPDUType:=CR, seqNo:=omit, iData:=omit }
50       }
51       template MDATreq ConnectionConfirmation:={
52         mData:={ iPDUType:=CC, seqNo:=omit, iData:=omit }
53       }
54       template MDATreq DisconnectionRequest:={
55         mData:={ iPDUType:=DR, seqNo:=omit, iData:=omit }
56       }
57       template MDATind DataTransfer(UserPDU data, SequenceNumber no):={
58         mData:={ iPDUType:=DT, seqNo:=no, iData:=data }
59       }
60       template MDATreq DataAcknowledgement(SequenceNumber number):={
61         mData:={ iPDUType:=AK, seqNo:=number, iData:=omit }
62       }
63     }
```

Figure 2.25: Definition of Templates

The template definition in lines 48–50 describes an MDATind service primitive which encapsulates an Inres Connection Request PDU. A Connection Request PDU does not require a sequence number or a payload. Thus, these two optional fields are omitted by using the keyword **omit**. Templates may also be parameterised as shown in lines 57–62, where only the iPDUType is fixed, the values for the remaining fields may be provided later-on by using parameters.

The module inresDefinitions does not contain any templates using matching mechanisms. In next section, examples for wildcards are provided. Furthermore, the usage of so called *inline templates* as an alternative to the given template definitions will be demonstrated there.

Finally, Figure 2.26 depicts the definition of a *component* type which is later-on used as *Test System Interface*, i.e. all ports through which the test system accesses the SUT. For testing an Initiator implementation, ports of type InresSAP and MediumSAP are required. Lines 64–67 of Figure 2.26 denote that a component of type InresSystemType contains instances of these port types named ISAP and MSAP.

```
64     type component InresSystemType {
65       port InresSAP ISAP;
66       port MediumSAP MSAP;
67     }
68   } // End of module inresDefinitions
```

Figure 2.26: Definition of the Test System Interface.

2.5.2 Behaviour Definition

After data types and values have been described, they shall be used in a test suite. The basic behavioural concepts of TTCN-3 are introduced by a test suite which is designed for a local test architecture. Thereafter, a test suite for a distributed test architecture is presented to explain TTCN-3 support for distributed testing.

The local test architecture which is used in the first example is depicted in Figure 2.27. It contains just one test component, the *Main Test Component* (MTC). The MTC emulates an Initiator user and an Inres Responder entity. Thus, it consists of the port ISAP and MSAP which can be regarded as PCOs according to CTMF. Thus, queues are used for receiving.

In addition to the ports of the MTC, the whole TTCN-3 test system has an interface to the SUT, the Test System Interface. For local test architectures, it is usually identical to the MTC's interface. This interface is divided into an *Abstract* and a *Real Test System Interface*. While the abstract one is defined using TTCN-3 (cf. Figure 2.26), the Real Test System Interface has to be implemented outside of TTCN-3 involving, e.g., hardware interfaces. For test execution, the ports of the MTC have to be associated (*mapped*) to the ports of the Test System Interface.

As depicted in Figure 2.28, the test suite for the local test architecture is located in a module of its own (Line 1). To gain access to all common definitions of message types, templates and port types for Inres, the inres-Definition module is imported in Line 2 and in lines 3–5, the user defined payload is selectively imported from module inresUserDefinitions. Lines 6-11 demonstrate the declaration of *module parameters*, which can be passed to a module when actually using it. Furthermore, it is possible to provide de-

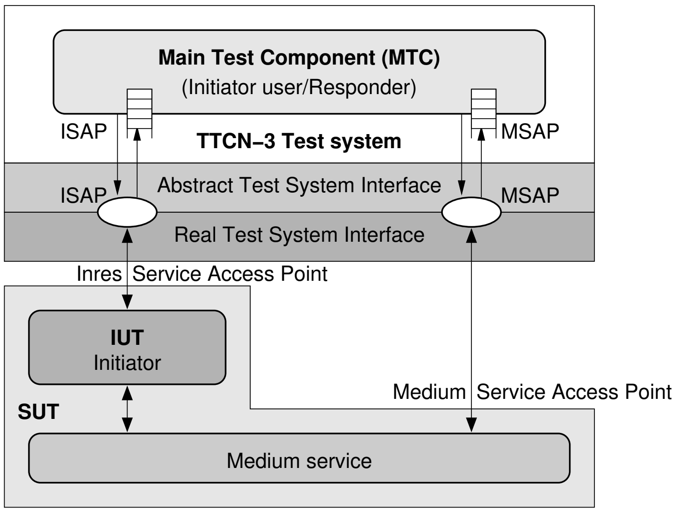

Main Test Component (MTC)
(Initiator user/Responder)
ISAP
TTCN–3 Test system
MSAP
ISAP
Abstract Test System Interface
MSAP
Real Test System Interface
Inres Service Access Point
IUT
Initiator
SUT
Medium Service Access Point
Medium service

Figure 2.27: TTCN-3 Test System for a Local Inres Test Architecture

```
module inresLocal {
  import from inresDefinitions all;
  import from inresUserDefinitions {
    const somePayload;
  }
  modulepar {
    integer transmissions:=100;
    float maxRetransmissionTime:=20.0;
    float maxExecutionTime:=2000.0;
    SequenceNumber initialSequenceNumber:=one;
  }
```

Figure 2.28: Inres Test Suite for Local Test Architecture: Module Parameter

fault values for them. In the inresLocal module, the following parameters are used: transmissions for the number of data transmissions that shall be performed for testing the SUT's data transmission capabilities; an upper limit for the duration of retransmissions (maxRetransmissionTime); an upper limit for the overall execution time of a test case (maxExecutionTime); and the initialSequenceNumber of the Initiator implementation when transmitting a PDU via the Medium service.

Figure 2.29 shows the TTCN-3 **testcase** construct. It embraces the behaviour description of test case bulkDataTransfer. The purpose of this test case is to test the repeated correct transmission of data from ISAP to MSAP.

In Line 12, the test case parameter iterations is declared which is used inside the test case to determine the number of repetitions for transferring data. Furthermore, the **runs on** keyword specifies that this test case runs on a test component which is of type InresSystemType. The component on which a **testcase** executes is implicitly considered as MTC. Moreover, the component type of the MTC is also taken as type of the Test System Interface.

Next, in lines 13–15, local variables are defined: a counter is used to track the number of retransmissions of lost PDUs, which are detected by deviations from the expectedSequenceNumber. The latter is initialised using a module parameter. To be able to extract the actual sequence number from a received MDATind PDU, variable receivedPDU is used to store such PDUs.

In order to guarantee the termination of a test case, a *timer* can be used to detect that the SUT does not respond anymore. The definition of such a timer T is demonstrated in Line 16. TTCN-3 represents time as floating point numbers (type **float**) and uses *seconds* as time unit.

When specifying a test case, it is desirable to focus on the expected behaviour only. To avoid cluttering up a test case, it is possible to shift the description of any invalid or unexpected behaviour into a so called *altstep*. An altstep can be activated as a *default* behaviour for treating any observed events which are not explicitly specified by a test case. Line 17 depicts the activation of the default failOrInconc. Since that default also handles time-

```
12  testcase bulkDataTransfer(integer iterations) runs on InresSystemType {
13    var integer counter;
14    var SequenceNumber expectedSequenceNumber:=initialSequenceNumber;
15    var MDATind receivedPDU;
16    timer T;
17    var default failOrInconcDefault:=activate(failOrInconc(T));
18    map(self:ISAP, system:ISAP);
19    map(self:MSAP, system:MSAP);
20    Preamble();
21    for(var integer i:=1; i<=iterations; i:=i+1) {
22      ISAP.send(IDATreq:{somePayload});
23      counter:=1;
24      T.start(maxRetransmissionTime);
25      alt {
26        [ ] MSAP.receive(DataTransfer(somePayload, expectedSequenceNumber)) {
27          T.stop;
28          MSAP.send(DataAcknowledgement(expectedSequenceNumber));
29           expectedSequenceNumber:=toggle(expectedSequenceNumber);
30        }
31        [counter<=4] MSAP.receive(DataTransfer(somePayload,?))
32                                  -> value receivedPDU {
33          MSAP.send(DataAcknowledgement(receivedPDU.mData.seqNo));
34          counter:=counter+1;
35          repeat;
36        }
37        [counter>4] MSAP.receive(DataTransfer(somePayload,?)) {
38          setverdict(fail);
39          stop;
40        }
41      }
42    }
43    setverdict(pass);
44    Postamble();
45    deactivate(failOrInconcDefault);
46  }
```

Figure 2.29: Inres Test Suite for Local Test Architecture: Test Case

outs, the local Timer T, which has been defined in the preceding line, is passed as an additional parameter. The result of the **activate** statement is stored as a handle in variable failOrInconcDefault to be able to deactivate that default if it is not desired anymore.

To complete the test case set-up, the test component's ports need to be *mapped* to the ports of the Test System Interface. This is performed using the **map** statement in lines 18 and 19.

Before the test behaviour which realises the test purpose can be executed, the Inres Initiator implementation must perform a connection establishment. This is achieved in Line 20 by calling the function Preamble.

Then, the actual test behaviour starts. For obtaining iterations repetitions of the behaviour in the enclosed block, an ordinary **for** loop is used (Line 21).

In Line 22, a *stimulus* is send via port ISAP to the SUT using the **send** statement. The data to transfer is specified using an *inline template*: According to its definition in Figure 2.23, an IDATreq message requires just one parameter. It is provided using the ":{...}" inline template notation.

After the stimulus has been sent, the counter for counting retransmission is set to 1 and Timer T is started (lines 23–24). The duration of the timer is determined by the module parameter maxRetransmissionTime.

The **alt** block in lines 25–41 describes possible *alternative* observations and resulting reactions. Each of the blocks which are prefixed by a "[...]" guard defines a branch of an alternative.

The first alternative (lines 26–30) is that a message with correct payload and expectedSequenceNumber is received at port MSAP. This is specified using a **receive** statement and the DataTransfer template. The reception of a message which matches this template is a correct transmission scenario, thus the Timer T is stopped (Line 27). Next, the message needs to be acknowledged to the SUT via port MSAP as shown in Line 28. Since the data transfer was successful, expectedSequenceNumber is then switched to the next valid sequence number by calling the function toggle.

The second branch of the alternative (lines 31–36) is guarded by the boolean expression counter<=4 which must evaluate to true to enable the branch. The expression assures that the maximum number of retransmissions has not been reached, yet. The DataTransfer template used in the **receive** operation contains a "?" wildcard as parameter (Line 31). This matches any sequence number. The semantics of an **alt** construct is that the alternative branches are evaluated from top to bottom and the first matching one is taken. Thus, the above wildcard will never be applied to an expected sequence number, because that would have been already matched by the first alternative branch. Hence, only unexpected sequence numbers are matched. In this case, the Medium service lost the acknowledgement; thus, it has to be resent. This happens in Line 33 by sending an acknowledgement for the received seqNo which is extracted from the receivedPDU. (While in Line 26, the value of the received message has been discarded, in Line 32, it is stored in receivedPDU due to the "**-> value**" construct appended to the preceding **receive** statement.) Finally, the counter of retransmissions incremented. The **repeat** statement in Line 35 is responsible for a re-evaluation of the alternative, i.e. the alternative starts again from top.

The last alternative of the **alt** statement (lines 37–40) is also guarded and thus matches only if the SUT resends a PDU more than four times. This is invalid behaviour, hence, the **setverdict** statement (Line 38) is used to assign a **fail** *verdict* to the result of the test case. Furthermore, the test terminates immediately by using the **stop** statement (Line 39).

If none of the alternative branches matches, the **alt** block is re-evaluated from the beginning until an event occurs which matches a branch. To achieve a deterministic evaluation of alternatives, TTCN-3 freezes the communica-

```
47  altstep failOrInconc(timer T) runs on InresSystemType {
48    [ ] ISAP.receive(IDISind:{}) {
49      setverdict(inconc);
50      stop;
51    }
52    [ ] ISAP.receive {
53      setverdict(fail);
54      stop;
55    }
56    [ ] MSAP.receive {
57      setverdict(fail);
58      stop;
59    }
60    [ ] T.timeout {
61      setverdict(fail);
62      stop;
63    }
64  }
```

Figure 2.30: Inres Test Suite for Local Test Architecture: Altstep

tion ports and state of timers and evaluates a *snapshot* of that situation. This avoids race conditions due to events occuring during evaluation of an alternative. If a branch matches, it is taken and behaviour continues just after the **alt** block unless the branch contains a **repeat** statement.

If all iterations of the **for** loop were successful, the test verdict can be set to **pass** (Line 43) and the connection can be released by calling the function Postamble (Line 44). Finally, the activated default failOrInconcDefault can be deactivated as shown in Line 45.

The **alt** statement in Figure 2.29 did not show all possible alternative branches which are handled by the test case. Additional branches are contributed by the default failOrInconc which is activated in Line 17. Figure 2.30 provides the definition of this default.

Defaults are defined using the **altstep** statement. If altsteps involve communication operations or component timers, they require the specification of a component type on which they may be executed (**runs on** in Line 47). Local timers which are handled by an altstep can be passed as a parameter.

Altsteps are only evaluated if the branches of the regular **alt** statement did not match, i.e. altsteps are appended at the end of an **alt** statement. The given altstep is quite simple. The first alternative (lines 48–51) treats reception of an IDISind message which is specified using an empty inline template. This behaviour is valid since the IUT may decide to release a connection if it considers the Medium service as too unreliable. Thus, an *inconclusive* verdict is set in Line 49. An inconclusive verdict denotes behaviour which is valid but due to which it was not able to reach the test purpose. For this reason, test execution is terminated using the **stop** statement. All other branches of the altstep lead to a **fail** verdict. These cases occur if a message was not consumed by one of the preceding branches and thus is consumed by either **receive** statement in Line 52 or 56. The **timeout** statement in Line 60 catches the expiration of Timer T. This is an indication that the SUT does not respond anymore.

Figure 2.31 presents the pre- and postambles that are called in test case bulkDataTransfer. Such supplemental behaviour is defined in ordinary *functions*. Since they contain also communication operations, the component type on which they may be executed has to be specified using the **runs on** statement. Apart from that, Figure 2.31 does not contain any unexpected TTCN-3 statements. Though, it shall be noted that —except due to defaults— no verdicts are set in the pre- and postambles. The intent is just to establish or respectively release a connection. Nevertheless, the ability of the IUT to perform this correctly has also to be tested. It is assumed that this has been ensured before by the test cases connectionEstablishment and connectionRelease. Their behaviour is not presented here.

Finally, the *Module Control Part* is shown in Figure 2.32. The control part can be used to determine ordering and conditions for executing test cases. In the example, first a variable of type **verdicttype** is defined (Line 90). Then,

```
function Preamble() runs on InresSystemType {
  timer T;
  var default failOrInconcDefault:=activate(failOrInconc(T));
  ISAP.send(ICONreq:{});
  T.start(maxRetransmissionTime);
  alt {
    [ ] MSAP.receive(ConnectionRequest) {
       MSAP.send(ConnectionConfirmation);
       repeat;
    }
    [ ] ISAP.receive(ICONconf:{}) {
       T.stop;
    }
  }
  deactivate(failOrInconcDefault);
}

function Postamble() runs on InresSystemType {
  MSAP.send(DisconnectionRequest);
}

testcase connectionEstablishment() runs on InresSystemType { /* ... */ }

testcase connectionRelease() runs on InresSystemType { /* ... */ }
```

Figure 2.31: Inres Test Suite for Local Test Architecture: Pre-/Postamble

```
 89    control {
 90      var verdicttype myVerdict;
 91      log("Starting test execution ... ");
 92      myVerdict:=execute(connectionEstablishment(), maxExecutionTime);
 93      if (myVerdict==pass) {
 94        myVerdict:=execute(connectionRelease(), maxExecutionTime);
 95      }
 96      if (myVerdict==pass) {
 97        myVerdict:=execute(bulkDataTransfer(transmissions), maxExecutionTime);
 98      }
 99      log("Test execution terminated");
100    }
101  } // End of module inresLocal
```

Figure 2.32: Inres Test Suite for Local Test Architecture: Module Control

in Line 91, a character string is written to the *test log*. Afterwards, the test case connectionEstablishment is executed (Line 92). This is achieved by using the **execute** statement with the name of the test case as first parameter. The optional second parameter (maxExecutionTime) can be used to restrict the duration of the test case execution. If that limit is exceeded, an **error** verdict is set by the runtime system and test execution aborts. The **execute** statement returns the test case's verdict. The subsequent test cases are only executed if the previous ones yielded a **pass** verdict (lines 93–98).

2.5.3 Defining Distributed Tests

The distributed test architecture which is used by the second test suite is depicted in Figure 2.33. Instead of a single test component, it consists of a

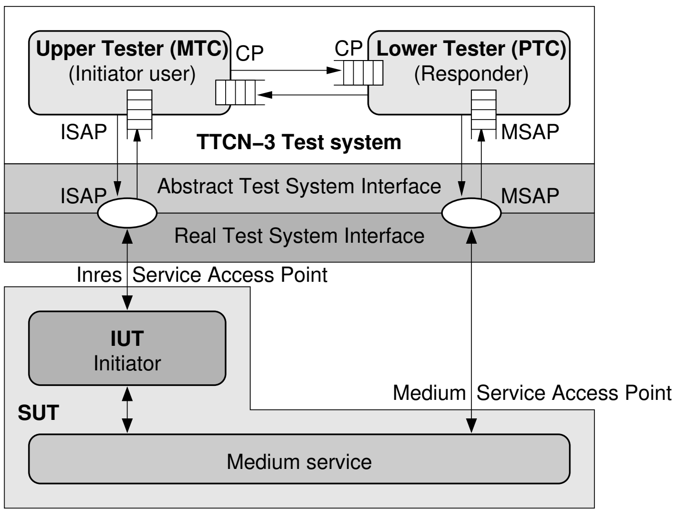


Figure 2.33: TTCN-3 Test System for a Distributed Inres Test Architecture

```
 1 module inresDistributed {
 2   import from inresDefinitions all;
 3   import from inresUserDefinitions {
 4     const somePayload;
 5   }
 6   modulepar {
 7     integer transmissions:=100;
 8     float maxRetransmissionTime:=20.0;
 9     float maxExecutionTime:=2000.0;
10     SequenceNumber initialSequenceNumber:=one;
11   }
12
13   // Definitions related to Communication TC<−>TC
14   type port CoordinationPoint message {
15     inout boolean; // Used for coordination
16   }
17
18   group ComponentDefinitions {
19     // Upper Tester
20     type component InitiatorUserType {
21       port InresSAP ISAP;
22       port CoordinationPoint CP;
23       timer T;
24     }
25
26     // Lower Tester
27     type component ResponderType {
28       port MediumSAP MSAP;
29       port CoordinationPoint CP;
30     }
31   }
```

Figure 2.34: Additional Types for Distributed Inres Test Architecture

Main Test Component (MTC) and one *Parallel Test Component* (PTC).[15] The MTC plays the role of an Initiator user, the PTC emulates a Responder. To achieve their common goal, both *Test Components* (TCs) run concurrently and communicate with each other via the coordination points CP. While the Test System Interface remains the same as for the local architecture, the type of the test components has changed: the MTC consists of the ports ISAP and CP, the PTC of the ports MSAP and CP.

The beginning of the test suite for the distributed test architecture is shown in Figure 2.34. Lines 2–11 are the same as for the local test suite. However, an additional port type has to be defined for the CoordinationPoint (lines 14–16). For simplicity, the coordination between the two TCs is performed by exchanging boolean values in both directions (**inout**) as shown in Line 15.

[15]Note that TTCN-3 makes no assumptions where the TCs are actually located, i.e. both TCs may be located together on the same test system node or distributed on two nodes of a test system. TTCN-3 gives the test implementor the freedom to decide on this.

```
32  testcase bulkDataTransfer(integer iterations)
33            runs on InitiatorUserType system InresSystemType {
34    var default failOrInconcDefault:=activate(initiatorFailOrInconc());
35    var ResponderType responder:=ResponderType.create;
36    map(self:ISAP, system:ISAP);
37    map(responder:MSAP, system:MSAP);
38    connect(self:CP, responder:CP);
39    responder.start(bulkDataTransferResponder(iterations));
40    initiatorUserPreamble();
41    CP.send(boolean:true);
42    CP.receive(boolean:true);
43    for(var integer i:=1; i<=iterations; i:=i+1) {
44      ISAP.send(IDATreq:{somePayload});
45      T.start(maxRetransmissionTime);
46      CP.receive(boolean:true);
47      CP.send(boolean:true);
48      T.stop;
49    }
50    all component.done;
51    setverdict(pass);
52    initiatorUserPostamble();
53    deactivate(failOrInconcDefault);
54  }
```

Figure 2.35: Inres Test Suite: Behaviour of Main Test Component

Additional component types are required for the two TCs. For the MTC, the InitiatorUserType is defined in lines 20–24. It consists of an ISAP port instance for communicating with the SUT and a CP port instance for coordination with the PTC. Additionally, Line 23 demonstrates that a component may also contain variables or, e.g., a timer which is shared by any test behaviour (test cases, functions, altsteps) that executes on the same component instance. The ResponderType definition in lines 27–30 is intended for the PTC. It consists of an MSAP and a CP port instance.

Now, the test case can be specified as given in Figure 2.35. By definition, a TTCN-3 **testcase** runs on the MTC component. Thus, the component type specified via the **runs on** keyword is InitiatorUserType. Since for distributed testing, the Test System Interface differs from the MTC component type, it has to be specified separately using the **system** keyword (Line 33).

Like for the local test architecture, a default is activated in Line 34. Next, in Line 35, a new instance of the PTC of type ResponderType is created and a reference to this component is stored in variable responder.

Test set-up continues by mapping the own ISAP port to the Test System Interface (Line 36). In the following line, the MTC maps the PTC's (referenced by responder) MSAP port to the Test System Interface. Then, the MTC *connects* its own CP port to the PTC's CP port by using the **connect** statement in Line 38. Finally, the behaviour of function bulkDataTransfer-

```
55  function bulkDataTransferResponder(integer iterations)
56              runs on ResponderType {
57    var integer counter;
58    var SequenceNumber expectedSequenceNumber:=initialSequenceNumber;
59    var MDATind receivedPDU;
60    var default failDefault:=activate(responderFail());
61    responderPreamble();
62    CP.receive(boolean:true);
63    CP.send(boolean:true);
64    for(var integer i:=1; i<=iterations; i:=i+1) {
65      counter:=1;
66      alt {
67        [ ]  MSAP.receive(DataTransfer(somePayload,expectedSequenceNumber)) {
68          MSAP.send(DataAcknowledgement(expectedSequenceNumber));
69          expectedSequenceNumber:=toggle(expectedSequenceNumber);
70          CP.send(boolean:true);
71          CP.receive(boolean:true);
72        }
73        [counter<=4] MSAP.receive(DataTransfer(somePayload,?))
74                                        -> value receivedPDU {
75          MSAP.send(DataAcknowledgement(receivedPDU.mData.seqNo));
76          counter:=counter+1;
77          repeat;
78        }
79        [counter>4] MSAP.receive(DataTransfer(somePayload,?)) {
80          setverdict(fail);
81          stop;
82        }
83      }
84    }
85    setverdict(pass);
86    responderPostamble();
87    deactivate(failDefault);
88  }
```

Figure 2.36: Inres Test Suite: Behaviour of Parallel Test Component

Responder is *started* on the created responder component. The set-up of the distributed test components is now completed.

The communication behaviour of test case bulkDataTransfer is basically comparable to the one which can be obtained from the corresponding test case for the local test architecture by restricting on behaviour related to the ISAP port, i.e. the MTC sends just the IDATreq stimulus (Line 44). Though, as shown in lines 41–42 and 46–47 of Figure 2.35, additional handshake messages are used to synchronise the MTC's behaviour with the PTC: the first handshake takes care that one test component does not start before the other is ready. Via the second handshake, the PTC informs the MTC that it has received the correct data and thus, the MTC may cancel its Timer T (Line 48) and shall proceed with sending more data.

Line 50 contains a further addition in comparison to the local test case: the **all component.done** statement waits for all PTCs to terminate. A PTC terminates either due to a **stop** statement or if the function which was used to start a PTC finishes. Furthermore, when a TC terminates, its verdict contributes to the global test verdict. Each TC maintains its own local verdict. Thus, to obtain a final global verdict, the local verdicts of all TCs need to be merged. For this, but also for setting a local verdict with the **setverdict** statement in general, special overwriting rules apply. The rules define that a verdict may only get worse, but never be upgraded. For example, a pass verdict may be downgraded by an inconclusive or fail verdict. But, once an inconclusive or fail verdict has been set, it cannot be upgraded anymore by setting, e.g., a pass verdict. The descending order of verdicts is **none**, **pass**, **inconc**, **fail**. The **none** verdict is the initial value of a local verdict. A special **error** verdict can only be set by the test system itself to indicate a severe internal error.

The behaviour for the PTC which runs on type ResponderType is shown in Figure 2.36. Like for the MTC, the behaviour of the PTC is comparable to the local test case restricted to the MSAP port. The handshake communication operations in lines 62–63 and 70–71 are symmetrical to the ones of the MTC.

As shown in Line 34 of Figure 2.35 and Line 60 of Figure 2.36, each TC activates its own default altsteps. Their definition is depicted in Figure 2.37.

```
altstep initiatorFailOrInconc () runs on InitiatorUserType {
  [ ] ISAP.receive(IDISind:{}) {
    setverdict(inconc);
    stop;
  }
  [ ] ISAP.receive {
    setverdict(fail);
    stop;
  }
  [ ] T.timeout {
    setverdict(fail);
    stop;
  }
}

altstep responderFail() runs on ResponderType {
  [ ] MSAP.receive {
    setverdict(fail);
    stop;
  }
}
```

Figure 2.37: Inres Test Suite for Distributed Test Architecture: Altsteps

Since Timer T is now part of the InitiatorUserType component instance, it needs not to be passed as parameter into the altstep, but can be used directly. Care has to be taken to avoid blocking of a TC. For example, the **stop** statement in Line 107 of Figure 2.37 leads to a termination of the PTC. As a result, the MTC may then wait infinitely for a handshake message of the PTC. In the sample test suite, this is solved by the MTC Timer T which eventually expires if the PTC has stopped. With respect to termination,

```
function initiatorUserPreamble() runs on InitiatorUserType {
  var default failOrInconcDefault:=activate(initiatorFailOrInconc());
  T.start(maxRetransmissionTime);
  ISAP.send(ICONreq:{});
  ISAP.receive(ICONconf:{});
  T.stop;
  CP.send(boolean:true);
  CP.receive(boolean:true);
  deactivate(failOrInconcDefault);
}

function responderPreamble() runs on ResponderType {
  var default failDefault:=activate(responderFail());
  alt {
    [ ] MSAP.receive(ConnectionRequest) {
      MSAP.send(ConnectionConfirmation);
      repeat;
    }
    [ ] CP.receive(boolean:true);
  }
  CP.send(boolean:true);
  deactivate(failDefault);
}

function initiatorUserPostamble() runs on InitiatorUserType {
  CP.receive(boolean:true);
  CP.send(boolean:true);
}

function responderPostamble() runs on ResponderType {
  MSAP.send(DisconnectionRequest);
  CP.send(boolean:true);
  CP.receive(boolean:true);
}

testcase connectionEstablishment()
         runs on InitiatorUserType system InresSystemType { /* ... */ }

testcase connectionRelease()
         runs on InitiatorUserType system InresSystemType { /* ... */ }
```

Figure 2.38: Distributed Inres Test Suite: Post-/ Preambles

```
  control {
    var verdicttype myVerdict;
    log("Starting test execution ... ");
    myVerdict:=execute(connectionEstablishment(), maxExecutionTime);
    if (myVerdict==pass) {
      myVerdict:=execute(connectionRelease(), maxExecutionTime);
    }
    if (myVerdict==pass) {
      myVerdict:=execute(bulkDataTransfer(transmissions), maxExecutionTime);
    }
    log("Test execution terminated");
  }
} // End of module inresDistributed
```

Figure 2.39: Distributed Inres Test Suite: Module Control

TTCN-3 treats the MTC differently than the other PTCs. If the MTC terminates, all PTCs are automatically terminated as well.

The pre- and postambles of the TCs are provided in Figure 2.38. Like the test cases and altsteps, they are also split variants of the local test suite and make as well additional use of coordination messages. The Module Control Part shown in Figure 2.39 is reused from the local test suite without any modifications.

2.5.4 Test Implementation

TTCN-3 can not only be used for test specification, but also for test implementation. The TTCN-3 core notation can be translated into executable code by tools. However, TTCN-3 test suites are abstract, thus additional system specific information needs to be added. To ease test implementation across platforms, two different sets of interfaces are standardised.

The *TTCN-3 Runtime Interface* (TRI) [ETS03c] provides interfaces for accessing a *Platform Adaptor* and a *SUT Adaptor* from within a TTCN-3 runtime system. A Platform Adaptor provides standardised access to the underlying execution platform, e.g. to timers based on an operating system's clock. Thus, the Platform Adaptor facilitates an easy implementation of TTCN-3 timers. An SUT Adaptor is responsible for implementing the Real Test System Interface through which the actual bit strings of a message are exchanged.

The *TTCN-3 Control Interface* (TCI) [ETS03d] provides several interfaces through which a TTCN-3 runtime system can access, e.g. logging facilities, request the creation of (remote) PTCs, or encode/decode abstract TTCN-3 message definitions into actual bit strings and vice versa. For sending a message, first the TCI can be called to obtain a bit string which is then send using the TRI. For receiving, the order is reverse.

2.6 Summary

In this chapter, foundations of testing have been explained and an introduction into the *Conformance Testing Methodology and Framework* (CTMF), which describes a black-box testing approach, has been given. The problem of testing non-functional, in particular real-time properties has been discussed. For specifying hard real-time properties in a formal manner, an overview on the *Message Sequence Chart* (MSC) language was given. Supplementary, an MSC extension to enable the specification of periodic real-time requirements has been proposed. Furthermore, the Inres protocol was introduced as an example of a protocol for communication systems. It is used as a case study in the subsequent chapters. Finally, an introduction into the *Testing and Test Control Notation version 3* (TTCN-3) for specifying and implementing distributed functional tests was provided.

Chapter 3

*TIMED*TTCN-3

The development of the *Testing and Test Control Notation* (TTCN-3) concentrated on functional testing. Thus, some major concepts needed for real-time testing are missing. This chapter tries to close this gap by proposing *TIMED*TTCN-3 as a real-time extension for TTCN-3. *TIMED*TTCN-3 introduces a new test verdict to judge non-functional behaviour. Absolute time is supported as a means to measure time and to calculate durations. The execution of statements can be delayed for defining time dependent test behaviour. The notion of absolute time benefits from support for the specification of clock synchronisation for test components. Finally, a means for the online and offline evaluation of real-time properties is provided.

The structure of this chapter is as follows: First, in Section 3.1, the need of a TTCN-3 real-time extension is motivated. Section 3.2 gives a first impression of *TIMED*TTCN-3 by providing a real-time test case example which is used to explain the *TIMED*TTCN-3 features. Then, in Section 3.3, verdicts for non-functional behaviour are discussed. Section 3.4 describes the time extensions that are part of *TIMED*TTCN-3. Afterwards, in Section 3.5, two evaluation methods for real-time properties are presented. For supporting graphical test case specification, the *Graphical Presentation Format for* TIMED *TTCN-3* (*TIMED*GFT) is briefly introduced in Section 3.6. The tabular presentation of *TIMED*TTCN-3 is discussed in Section 3.7. Finally, a summary and a comparison to related approaches (Section 3.8) are given. This chapter is based on a joint work published in [DGN02, DGN03].

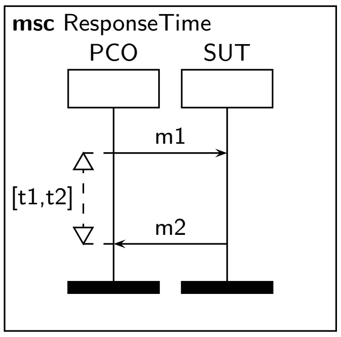
3.1 Specifying Real-Time Tests with TTCN-3

Even though TTCN-3 was not designed for real-time testing, it has nevertheless been used for specifying real-time tests [DST04]. Even its predecessor, the *Tree and Tabular Combined Notation* (TTCN-2), has been successfully used to control external devices for performance testing [GKS00].

However, using standard TTCN-3 for specifying real-time tests has some limitations. For being able to assess time properties within TTCN-3, its timer construct has to be used intensively. The drawback of using timers is that readability of test specifications suffers significantly, because timers are simply not intended for testing real-time properties. Furthermore, timers are always local to a test component, thus, it is impossible to test real-time properties which are imposed on events that occur at different components.

The deteriorated readability of a test case which uses timers for assessing real-time properties shall be illustrated by an example [HKN01]. The test purpose is to test that the response time between stimulating the SUT with message m1 and receiving message m2 is within the interval [t1,t2] seconds. (A corresponding MSC is shown Figure 3.1a.)

The resulting timer-based TTCN-3 test case is shown in Figure 3.1b. The timer Tmin is used to verify that message m2 is not received too early after sending message m1 in Line 3. This is checked by the first **alt** block in lines 5–10: if timer Tmin expires (Line 6), everything is fine and execution may continue with the second **alt** block (lines 11–19). However, if instead message m2 is received (Line 7), this is too early and the **fail** verdict is set (Line 8). The second **alt** block (lines 11–19) is reverse: if message m2 is received (Line 12) prior to a timeout, the test passed and the still running timer Tmax is stopped (line 13–14). However, if timer Tmax expires, the test failed (lines 16–17), because the upper bound real-time requirement was violated.



(a) Real-Time Requirement

```
 1  timer Tmin, Tmax;
 2  Tmax.start(t2);
 3  PCO.send(m1);
 4  Tmin.start(t1);
 5  alt {
 6    [] Tmin.timeout;
 7    [] PCO.receive(m2) {
 8       setverdict(fail);
 9     }
10  }
11  alt {
12    [] PCO.receive(m2) {
13       Tmax.stop;
14       setverdict(pass);
15     }
16    [] Tmax.timeout {
17       setverdict(fail);
18     }
19  }
```

(b) TTCN-3 Test Case

Figure 3.1: Testing a Response Time Requirement Using TTCN-3 Timers

The example demonstrates that it is possible to use TTCN-3 for testing simple real-time requirements. However, the resulting test case is very clumsy: in addition to the two lines which are required for specifying the functional behaviour of stimulus and expected observation, 17 further lines are necessary for testing the non-functional behaviour. Moreover, it is very hard to identify the actual properties which shall be tested, because the test case is bloated and functional and non-function behaviour description is mixed.

Furthermore, when using timers, the measurement of durations is influenced by the TTCN-3 snapshot semantics and by the order in which **receive** and **timeout** operations are ordered in the **alt** statement. TTCN-3 makes no assumptions about the duration for taking and evaluating a snapshot. Thus, exact times cannot be measured using ordinary timers.

As a result, it can be summarised that timers shall be used for detecting or provoking the absence of signals and to take care that a test case eventually terminates if it is blocked for some reason, but not for specifying real-time requirements. Thus, for distributed real-time testing, other concepts for dealing with time are preferable. A possible solution, which combines the ease of TTCN-3 with real-time concepts, is *Timed*TTCN-3.

3.2 An Inres-based Example

The concepts of *Timed*TTCN-3 will be explained by a test suite for the *Inres* protocol. The test suite is designed for the distributed test architecture which has been described on pages 38–38. To enhance the readability of this chapter, a copy of Figure 2.33 is provided in Figure 3.2.

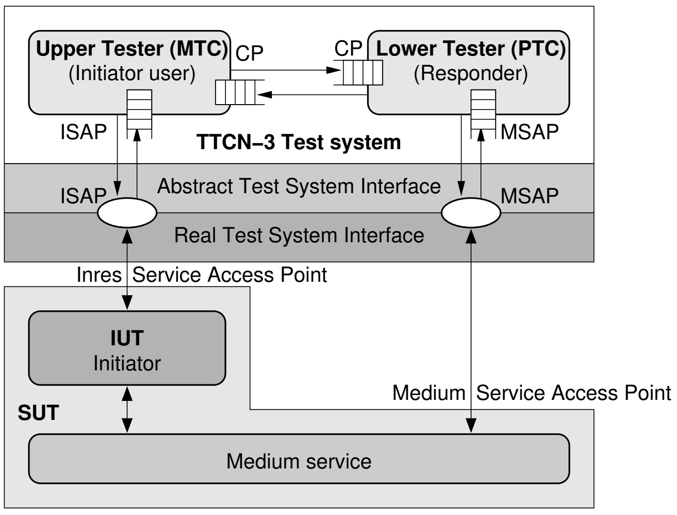


Figure 3.2: Distributed Test Architecture for Inres

```
1  module inresRTdistributed {
2    import from inresDefinitions all;
3    import from inresDistributed all;
4    modulepar {
5      integer transmissions:=100;
6      float maxExecutionTime:=2000.0;
7      SequenceNumber initialSequenceNumber:=one;
8    }
```

Figure 3.3: Inres *Timed*TTCN-3 Test Suite: Import and Module Parameter

The *Implementation Under Test* (IUT) is an *Initiator* implementation. The *Upper Tester* (UT) function plays the role of an Initiator user and the *Lower Tester* (LT) function plays the role of a Responder entity. The UT has a direct connection with the IUT via port ISAP, whereas the LT only has indirect access to the lower interface of the IUT via a *Medium Service* using port MSAP. UT and LT coordinate themselves via the coordination points CP.

The provided example test case is based on the test suite which has been presented in Section 2.5. Thus type, template, altstep, pre- and postamble definitions are imported from that TTCN-3 modules as depicted in lines 2–3 of Figure 3.3. Likewise, the module parameters are specified in lines 4–8.

Moreover, it is assumed that the IUT has passed the related functional test cases. Otherwise, it is not meaningful to perform the real-time test case described in this chapter. The following real-time test case does, e.g., not care about the functional requirement how often lost data packets are retransmitted by the IUT, but does acknowledge any number of retransmissions. Also, the LT does not inform the UT about successfully received data.

The test case is designed with the purpose to test the real-time properties *latency* and *mean inter-arrival time* for the exchange of 100 data packets. The principle control flow and message exchange is presented by the MSC in Figure 3.4. The test case starts with a preamble that establishes a connection between UT and LT. Afterwards, UT and LT synchronise in order to ensure that both tester functions are in a correct state to execute the test body. The test body includes the sending of 100 data packets from UT to LT. The LT must always acknowledge the correct reception of each data packet. Otherwise, the IUT will retransmit the data packet or, after unsuccessful retransmissions, release the connection. However, this is not shown in Figure 3.4. At the end of the test body, UT and LT synchronise again and perform a postamble to release the connection.

This MSC depicts also the latency real-time requirement which is imposed on sending IDATreq and receiving MDATind. It has to be within the open interval (1ms, 5ms). Furthermore, a constraint has been put on the test system itself in order to prevent a queueing of messages at the SUT: The

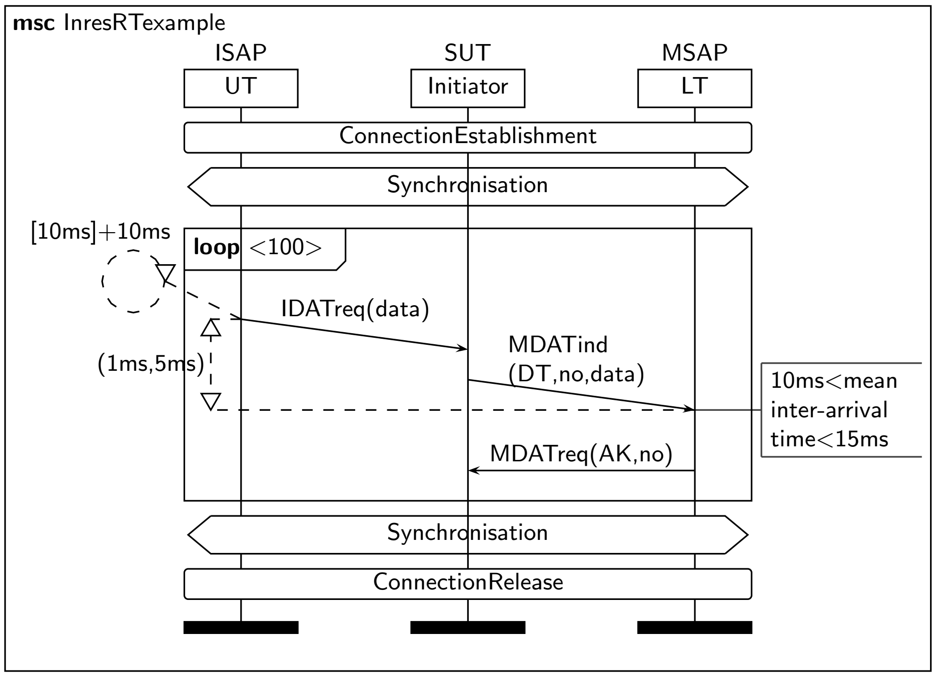


Figure 3.4: MSC for the Inres Test Case Example

IDATreq stimuli shall be sent periodically every 10ms. Moreover, this allows a reasonable testing of inter-arrival times. Since mean inter-arrival time is a soft real-time property, it cannot be expressed using MSC and is hence just indicated by an MSC comment attached to the MDATind reception. Nevertheless, *Timed*TTCN-3 allows to test that the mean time between consecutive arrivals of message MDATind is within 10ms and 15ms.

The *Timed*TTCN-3 code for the behaviour of the *Main Test Component* (MTC) is shown in Figure 3.5. In the example, the MTC is the UT, i.e. it plays the role of an Initiator user. Lines 9 and 10 provide the interface of the test case, i.e. test case name, formal parameters, component types for the MTC (**runs on** clause), and Abstract Test System Interface (**system** clause). Lines 11–14 describe variable declarations, a default activation, and the mapping of MTC ports onto ports of the Abstract Test System Interface.

The creation of the LT component, the mapping of LT ports onto ports of the Abstract Test System Interface, the connection of LT and MTC ports, and the start of the LT component are specified in lines 15–18. The preamble initiatorPreamble is called in Line 19 and a timer T is started (Line 20) just to take care that the test case eventually terminates in case of a blocked component. The initial synchronisation by means of an UT-initiated handshake with boolean synchronisation messages is shown in lines 21 and 22. The time for sending the first data packet is determined in Line 23.

```
 9 testcase inresRTexample(integer iterations)
10           runs on InitiatorUserType system InresSystemType {
11   var float sendTime:=0.0;
12   var default failOrInconcDefault:=activate(initiatorFailOrInconc());
13   var ResponderType responder:=null;
14   map(self:ISAP, system:ISAP);
15   responder:=ResponderType.create(self.timezone); // Create in same timezone
16   map(responder:MSAP, system:MSAP);
17   connect(self:CP, responder:CP);
18   responder.start(responderBehaviour(iterations));
19   initiatorUserPreamble();
20   T.start(maxExecutionTime); // Detect blocking of test case
21   CP.send(boolean:true);
22   CP.receive(boolean:true);
23   sendTime:=self.now+5.0; // Send for the first time in 5.0s from now
24   for(var integer i:=1; i<=iterations; i:=i+1) {
25     resume(sendTime); // Wait until 'sendTime'
26     log(TimestampType:{self.now, self.timezone, idatreq}); // Log timestamp
27     ISAP.send(IDATreq:{self.now}); // Piggyback send time
28     sendTime:=sendTime+0.01; // Send periodically every 10ms
29   }
30   CP.send(boolean:true);
31   CP.receive(boolean:true);
32   T.stop;
33   all component.done;
34   setverdict(pass);
35   initiatorUserPostamble();
36   deactivate(failOrInconcDefault);
37 }
```

Figure 3.5: Inres *Timed*TTCN-3 Test Suite: Test Case Description

The body of the test case consists of the **for** loop specified in lines 24–29. The loop body is repeated iteration times and specifies that a data packet is sent every 0.01s, i.e. 10ms (lines 25–28).

After that, the test case continues with the final synchronisation (lines 30 and 31) and stopping of Timer T (Line 32). Then, the MTC waits for the Responder component to terminate (Line 33) and sets the **pass** verdict in Line 34. Finally, the postamble initiatorPostamble is called (Line 35) and the default is deactivated.

The LT plays the role of a Responder entity. Its behaviour is specified by the *Timed*TTCN-3 function shown in Figure 3.6. The function can be structured into three parts and is very similar to the structure of the MTC test case (Figure 3.5). The first part consists of declarations (lines 39–41 of Figure 3.6), a default activation (Line 42), the call of the preamble responderPreamble (Line 43), and the initial synchronisation (lines 44 and 45).

The second part is the test body which consists of a **for** loop (lines 46–65). The loop body is repeated iterations times and includes an **alt** statement with

```
38  function responderBehaviour(integer iterations) runs on ResponderType {
39    var SequenceNumber expectedSequenceNumber:=initialSequenceNumber;
40    var float receiveTime:=0.0, sendTime:=0.0;
41    var MDATind receivedPDU;
42    var default failDefault:=activate(responderFail());
43    responderPreamble();
44    CP.receive(boolean:true);
45    CP.send(boolean:true);
46    for (var integer i:=1; i<=iterations; i:=i+1) {
47      alt {
48        [ ] MSAP.receive(DataTransfer(?, expectedSequenceNumber))
49                            −> value receivedPDU {
50            receiveTime:=self.now; // Get current time. Next, log time stamp:
51            log(TimestampType:{receiveTime, self.timezone, mdatind});
52            sendTime:=receivedPDU.mData.iData; // Extract send time
53            // Call latency online evaluation:
54            if (evalLatencyOnline(sendTime, receiveTime, 0.001, 0.005)==conf) {
55              setverdict(conf); // Real−time requirement violated
56            }
57            MSAP.send(DataAcknowledgement(expectedSequenceNumber));
58            expectedSequenceNumber:=toggle(expectedSequenceNumber);
59        }
60        [ ] MSAP.receive(DataTransfer(?,?)) −> value receivedPDU {
61            MSAP.send(DataAcknowledgement(receivedPDU.mData.seqNo));
62            repeat;
63        }
64      }
65    }
66    CP.receive(boolean:true);
67    CP.send(boolean:true);
68    setverdict(pass);
69    responderPostamble();
70    deactivate(failDefault);
71  }
```

Figure 3.6: Inres *TIMED*TTCN-3 Test Suite: Responder Behaviour

two alternatives. The first alternative (lines 48–59) describes the expected message exchange: A correct data packet is received (lines 48 and 49), the current time is retrieved and recorded in the test log (lines 50 and 51), the send time is extracted from the received message (Line 52). Then, the latency is evaluated (line 54): if the latency requirement is violated, the new test verdict **conf** (Section 3.3) is set (Line 55). Finally, the data packet is acknowledged (Line 57) and the sequence number of the next correct data packet is computed (Line 58). The second alternative describes the case when the previous acknowledgement got lost and, therefore, the previous data packet is retransmitted by the IUT. The reception of the retransmitted data packet is described in Line 60 and its re-acknowledgement is specified in Line 61. The **repeat** statement in Line 62 causes the re-evaluation of the entire **alt** statement, i.e. the test component waits for the reception of the next correct data packet or another retransmission.

The third part of function responderBehavior describes the final synchronisation (lines 66 and 67), the setting of the **pass** verdict (Line 68), and the call of responderPostamble (Line 69). Finally, the default is deactivated in Line 70 and the component terminates.

The test case specifies the expected message exchange only. Erroneous and unexpected responses received from the SUT are considered to be handled by defaults which are activated in Line 12 in Figure 3.5 and Line 42 in Figure 3.6. The behaviour of the default is provided by imported altsteps which were defined in Section 2.5.3.

The *Timed*TTCN-3 code in Figure 3.5 and Figure 3.6 includes the real-time extensions **self.now**, **resume**, **self.timezone**, the new verdict **conf**, a modified syntax for the **log** statement, and a new parameter for the **create** operation. These extensions are explained in the following sections.

3.3 Non-Functional Verdicts

In standard TTCN-3, the verdicts indicate basically whether a test case was successful (**pass**), inconclusive (**inconc**), or erroneous (**fail**) with respect to functional requirements. By introducing the possibility to test non-functional requirements, additional information concerning the test outcome is needed: A test case may *pass* with respect to both functional and non-functional behaviour, or it may *pass* only with respect to the functional behaviour while the non-functional requirements are violated.[1]

Since non-functional behaviour can be observed only in combination with functional behaviour on which the non-functional requirements are imposed, it is not meaningful to make any statements on non-functional test results if the functional behaviour is not conforming to the functional requirements.

Even in case of a *functional inconclusive*, no statement can be made on non-functional test results, since an inconclusive case may have other non-functional requirements than the pass case which is subject of testing. Hence, distinctive verdicts are just needed in case of a *functional pass*. In contrast to the functional verdicts, a *non-functional inconclusive* verdict is not needed, since a non-functional requirement is either fulfilled or not.

Besides the existing **pass** verdict which is in *Timed*TTCN-3 used to indicate a *functional pass* with an associated *non-functional pass*, *Timed*TTCN-3 introduces the new verdict **conf** (as abbreviation for *conforming*) to indicate a *functional pass* with an associated *non-functional fail*. Due to the introduction of the new verdict, the existing overwriting rules for verdicts are

[1]In the following, the terms *functional pass*, *non-functional pass*, etc. are used to describe the test outcome with respect to functional and non-functional behaviour.

Current value of verdict	New verdict assignment value				
	none	pass	conf	inconc	fail
none	none	pass	conf	inconc	fail
pass	pass	pass	conf	inconc	fail
conf	conf	conf	conf	inconc	fail
inconc	inconc	inconc	inconc	inconc	fail
fail	fail	fail	fail	fail	fail

Table 3.1: *Timed*TTCN-3 Overwriting Rules for the Test Verdicts

refined as given in Table 3.1.[2] The new verdict **conf** is inserted between the verdicts **pass** and **inconc**. This makes the usage of the verdicts downwardly compatible: existing, pure functional test suites or re-used altsteps which set the **pass** verdict do not change the non-functional result.

An example for the usage of the new **conf** verdict can be found in Figure 3.6: If the latency requirement is violated (checked by the **if** statement in Line 54), **conf** is assigned to the local verdict of the Responder test component (Line 55). Due to the overwriting rules of *Timed*TTCN-3, a **conf** verdict will not be overwritten by the **setverdict**(**pass**) statement at the end in Line 68. The existing verdict **fail** is still available to express a *functional fail* (cf. Figure 3.11, lines 109 and 121).

In accordance to TTCN-3, each test component maintains its own local verdict. The local verdicts contribute to the global verdict which is calculated from the local ones based on the overwriting rules shown in Table 3.1.

3.4 Time Concepts

As discussed in Section 3.1, TTCN-3 supports just *timers* for handling time, which are clumsy and influenced by the snapshot semantics. Furthermore, TTCN-3 has no concept of *absolute time*, i.e. a test component cannot read and use its local system time. In real-time testing, absolute time is necessary to check relationships between observed test events and to coordinate test activities. In case of synchronised clocks in a distributed test environment, the system time may be exchanged among test components to check real-time requirements that cannot be measured locally. Moreover, absolute system time may then be used for the timely coordination of test activities.

As a consequence of these considerations, *Timed*TTCN-3 has the concept of *absolute time* in order to support real-time testing. In case of a distributed test environment, the test cases may define requirements for the synchronisation of clocks of different test components.

[2]In the special case of a tester malfunction which may, e.g., lead to a wrong real-time measurement, the **error** verdict will be set by the test system.

3.4.1 Absolute Time

Absolute time is related to *clocks* that provide the actual value of time. *Timed*TTCN-3 assumes that each test component has access to such a clock, but makes no assumptions about the number and the synchronisation of these clocks.[3] Furthermore, *Timed*TTCN-3 assumes that the resolution of such clocks and the speed of the tester are adequate for the real-time requirements which are subject of testing.

For the handling of time values either a new type is needed, or the time values have to be mapped onto an existing basic type. Due to numerous possible time representations, e.g. the *Unix* approach to count the seconds since 1.1.1970 [IEE96] or a structured type with fields for year, month, day, hour, etc., a common new type for time values is not easy to define. Furthermore, a time type should support arithmetics and comparisons to evaluate time stamps.

For simplicity, *Timed*TTCN-3 uses the existing **float** type and follows the Unix approach, i.e. time is counted in seconds and the absolute time is represented by the number of seconds since a fixed point in time. In contrast to the Unix scheme, *Timed*TTCN-3 does not define a fixed starting point for the time measurement. But for performing measurements of time during the test run, the point in time at which a test run starts should at least be included in the domain of valid time points. For that, *Timed*TTCN-3 supports the usage of absolute time by the operations **now** and **resume**:

now is used for the retrieval of the current *local time*. The local character of the **now** operation is reflected by its application to the **self** handle, i.e. **self.now** is the expected call statement for the **now** operation. The operation **now** returns a float value that equals the current absolute time when the operation is called. The mapping of the float value onto a concrete daytime (i.e. year, month, day, hour, etc.) is considered to be outside the scope of *Timed*TTCN-3 and has to be provided by the test equipment, e.g. in form of additional conversion functions.

resume provides the ability to delay the execution of a test component. The argument of the **resume** operation is considered to be an absolute time value, i.e. the point in time when the test component shall resume its execution. If required, a relative time can easily be specified by using the current time as reference time, e.g., waiting for 3 seconds can be described by **resume**(**self.now** + 3.0). A **resume** operation has no effect if the specified time has already been passed before the

[3]From a conceptional point of view, synchronised test components share the same clock, even though in a real implementation, the clocks of distributed test components have to be synchronised by using a synchronisation protocol [Lam78, Lam90, RSB90, IET92] or by radio controlled clocks as, e.g., provided by the *Global Positioning System* (GPS) [LAK99].

operation is executed. Though, this case might be mentioned in the test log, because it is an indication that the test system is too slow.

An example for the usage of the absolute time extension **self.now** is shown in Figure 3.5. The current time is retrieved in Line 23. It is used to calculate the sending time of the first data packet. The sending time is used by the **resume** operation in Line 25: The test component will resume when the specified time is reached.

3.4.2 Synchronisation of Clocks

Time values are observed and used locally by the test components. Furthermore, time values that are observed at different test components may be exchanged and used for further computations and comparisons. But this only makes sense if the clocks of the involved test components are synchronised. The synchronisation mechanism itself is outside the scope of *Timed*TTCN-3 and should be guaranteed by the test equipment, but requirements for clock synchronisation may very well be expressed in *Timed*TTCN-3. These requirements may be used by a *Timed*TTCN-3 runtime system to distribute test components in a manner that they either share clocks physically or clock synchronisation procedures for the test devices are applied.

Timezones

Most specification and implementation languages either support *local time* or *global time*. Local time means that each behavioural entity, e.g., a *Specification and Description Language* (SDL) process [ITU99a] or a TTCN-3 test component, has its own local time. Global time means that all behavioural entities share the same *global* time. Global time is perfect for the purpose of real-time testing, because all test components have by definition the same global time and are synchronised.

However, neither local nor global time are realistic assumptions for real-time testing situations. A real-time test environment typically consists of several devices. If synchronisation among two or more test components is required to reach the goal of a test case, the components have to be executed either on the same device or on synchronised devices.

The developer of a real-time test specification should not care about synchronisation procedures and the distribution of test components her- or himself, but she or he can support later implementation by identifying test components which have to be synchronised. For this purpose, *Timed*TTCN-3 supports the *timezones* concept.

A timezone is an optional attribute that can be assigned to a test component when the component is created. Test components with the same timezone attribute are considered to be clock-synchronised, i.e. they have the same absolute time. A test component can only have one timezone attribute. Components without timezone attributes are considered to be not synchronised with any other component.

The timezones concept is implemented in the *Timed*TTCN-3 language by a designated enumeration type with the reserved name **timezones**. The user has to specify the timezone attribute values by defining the **timezones** type in the module definitions part of a *Timed*TTCN-3 module. This type has an implicit member of name **none** which indicates no clock synchronisation. The usage of an enumeration type only makes sense if the number of timezones is finite and known. For specifying real world test scenarios, this is a realistic assumption.

In *Timed*TTCN-3, the timezone attribute is an optional parameter of the **execute** statement and the **create** operation. The timezone attribute of an MTC is assigned when using the **execute** statement. Attributes of all other test components are assigned by means of the **create** operation.

The flexibility of the timezones concept can be improved by making the timezones visible to the test components. This is implemented in *Timed*TTCN-3 by means of a special **timezone** function which returns the timezone of the component that called the function. In the case of a non-synchronised test component, the value **none** is returned. Like the **now** operation, the **timezone** operation is always applied to the **self** handle of a test component, i.e. **self.timezone** is the expected manifestation of the **timezone** operation. The timezone information may be exchanged among test components to check if synchronisation conditions are satisfied, or it may be used to create several synchronised components.

The usage of the timezone concept is shown in figures 3.5, 3.7, and 3.10. Figure 3.7 presents the definition of timezones Goettingen, Luebeck, and Berlin. In the test case example, the MTC is created by the **execute** statement in Line 127 of Figure 3.10 and is assigned the timezone attribute Goettingen. The behaviour of the MTC is shown in Figure 3.5: The MTC creates the test component *responder* (Line 15 of Figure 3.5) and assigns its own timezone which is obtained using **self.timezone** to the new component, i.e. MTC and *responder* are considered to be synchronised. A *Timed*TTCN-3 runtime environment may use this information to ensure this synchronisation condition.

```
type enumerated timezones {
  Goettingen, Luebeck, Berlin
}
```

Figure 3.7: Definition of Timezones

3.5 Evaluation of Real-Time Properties

While functional behaviour is basically tested by using sequences of **send** and **receive** operations, real-time requirements can be tested by relating particular points in time to each other. The essence of the various real-time requirements can be broken down to the relationship of points in time [Koy91]. Mathematical formulae can be used to evaluate whether the points in time of interesting events fulfil a certain real-time requirement or not.

To obtain those points in time, (existing) functional TTCN-3 test cases are instrumented by statements which generate time stamps according to the test purpose. *Timed*TTCN-3 implements this approach by making use of the possibility to read absolute time values (Section 3.4) which serve as time stamps. The mathematical formulae which are applied on the collected time stamps can be coded as ordinary TTCN-3 functions. Those *evaluation functions* return a judgement which indicates whether a real-time requirement is fulfilled or not. *Online* or *offline evaluation* of time stamps is possible:

Online evaluation is needed if it is not possible to separate functional and non-functional requirements, i.e. a non-functional property directly influences the behaviour of a test case. In such a case, evaluation of non-functional observations must be performed during the test run in order to react on the result of the evaluation. Online evaluation has the drawback of cluttering the test case and possibly slowing down the performance of the test case which may be undesirable for time-critical test cases.

Offline evaluation may be used if the non-functional requirements which are subject of testing have no influence on the reaction of a test case. In this case, the test case just needs to be instrumented by statements that log the relevant time stamps. The non-functional requirement itself can be specified separately. Based on the time stamps in the log file, the non-functional properties can be evaluated when the test run has finished. Offline evaluation has the advantage of having a low impact on the performance of a test case, since only time stamps have to be logged during the test run. Moreover, it does not clutter up the functional test case with code needed for specifying non-functional requirements.

3.5.1 Online Evaluation

For performing online evaluation, the relevant time stamps have to be evaluated during the test run, e.g. by calling a special evaluation function with time stamps as actual parameters.

```
function evalLatencyOnline(float timeA, float timeB,
                           float lowerbound, float upperbound) return verdicttype {
  var float latency:=timeB−timeA;
  if ((latency<upperbound) and (lowerbound<latency)) {
    return pass; // Non−functional pass
  }
  else {
    return conf; // Non−functional fail
  }
}
```

Figure 3.8: *Timed*TTCN-3 Online Evaluation Function

In a distributed test architecture, non-functional requirements may involve time stamps which have been collected by different test components. In this case, the evaluating component needs to obtain time stamps from other components. To achieve this, time stamps can either be piggybacked[4] in the payload of some SUT signals or be communicated directly among test components by using coordination messages. For implementing online evaluation, the new concepts of *Timed*TTCN-3 which have been introduced so far, are sufficient.

In the test case example (Section 3.2), online evaluation is used to check the fulfilment of a *latency* requirement (Line 54 in Figure 3.6). In case of a violation, the local test verdict of the Responder test component is set to **conf** (Line 55).

The online evaluation of the latency requirement involves time stamps of several test components. Hence, the remote time stamps have to be transferred to the evaluating component. Since in the example, the additional connection between ports CP is solely used for functional coordination of the components, a piggyback approach is used: The evaluation function is called inside the Responder test component (Line 54 in Figure 3.6). The **receive** operation for the MDATind signal is local to this component (Line 48). Thus, the related time stamp can be easily obtained locally by calling **self.now** and storing it in variable receiveTime (Line 50). The corresponding **send** operation is performed by the MTC and the associated time stamp is hence piggybacked to the payload of the IDATind signal (Line 27 in Figure 3.5).[5]

The Responder test component extracts the piggybacked time stamp from the received signal and assigns it to variable sendTime (Line 52 in Figure 3.6). Afterwards, the online evaluation function evalLatencyOnline (Line 54) is called. The actual parameters of this evaluation function call are the send

[4]Piggybacking is only possible if the payload is not changed by the SUT. This property has to be tested previously by means of functional testing.

[5]In the Inres example, the payload of the IDATind signal is of type **float**. In the more general case, the **float** value has to be encoded into the particular payload type.

and receive time as well as the boundaries 1ms and 5ms which describe the incarnation of the latency real-time requirement.

The evaluation function evalLatencyOnline (Figure 3.8) checks the mathematical formula related to latency: lowerbound $< t_{receive} - t_{send} <$ upperbound (Line 78). Depending on the result, the function returns either a **pass** or a **conf** verdict (lines 79 and 82) which may be used by the calling entity for further decisions.

3.5.2 Offline Evaluation

When using offline evaluation, the evaluation function is called after test execution. *Timed*TTCN-3 offers a means to record time stamps in a log file during a test run in order to evaluate them afterwards. In this case, the final test verdict is a composition of the functional test verdict which has been determined during test run and of the result of the subsequent offline evaluation. To facilitate offline evaluation of real-time requirements, *Timed*TTCN-3 refines the existing log file concept of TTCN-3.

TTCN-3 assumes that one global or several local log files exist and provides a **log** statement to enable logging of comments. [ETS02a] does not specify the number of log files, the logging mechanism is not described, and neither module control nor test components can access the global or local log files. However, for an efficient offline evaluation, module control and test components need access to the log files and the content and structure of the log files has to be specified more formally.

The Log File Concept

A *Timed*TTCN-3 log file is basically a list of values of arbitrary TTCN-3 types. A log file is of type **logfile** and it is possible to handle log file references as variables or to pass them as parameters into functions.

Each *Timed*TTCN-3 test component has its own local log file. A local log file is initialised when the owning component is created. When test execution finishes, i.e. the MTC terminates, the local log files are automatically merged into a global one. *Timed*TTCN-3 does not specify the internal mechanisms that are needed for storing and maintaining log files[6], but defines four functions for accessing the entries of a log file (Table 3.2).

Logging of Events

*Timed*TTCN-3 refines the TTCN-3 **log** statement in order to write information into log files. But while in TTCN-3, the argument type of the **log** statement is a fixed string, in *Timed*TTCN-3, the argument can be the value

[6]The mechanisms for storing and maintaining log files are considered to be implementation specific and therefore outside the scope of *Timed*TTCN-3.

Operation name	*Return type*	*Function*
first(*sortkey, template*)	**boolean**	Select and sort log file by *sortkey* and move to first matching entry in the log file
next(*template*)	**boolean**	Move to the next matching entry
previous(*template*)	**boolean**	Move to the previous matching entry
retrieve	type of *sortkey* used as parameter of **first**	Retrieve entry from current log file position

Table 3.2: Overview of *Timed*TTCN-3 Log File Operations

of any arbitrary valid type. For offline evaluation, usually a structured data type containing a time stamp field is appropriate. A corresponding offline evaluation function will only consider log file entries of that special type in order to judge the fulfilment of the real-time requirement.

Log File Operations

For retrieving entries of a log file, *Timed*TTCN-3 offers means for sorting a log file by a certain field of the log file's entries. Since a log file may contain values of arbitrary types, sorting and retrieving is only possible for a certain type which has to be specified. According to the order which is imposed by sorting, the first, the next or the previous log file entry may be retrieved. For this purpose, *Timed*TTCN-3 uses an internal cursor which points to an entry in the log file. This cursor can be moved and the value at the current cursor position may be retrieved.

The operation **first** serves two purposes: It selects the entries of the log file by their type and sorts them. In addition, it moves the cursor to the first matching entry in the log file. The first parameter of **first** specifies the element which is used as a sorting key.[7] This is done using the TTCN-3 template notation: A "?" indicates the field which is used as sorting key, all other fields must be set to "-". The type of the template is used to restrict the type of entries which are considered by the log file operations presented in Table 3.2. The second parameter can be used to search for a certain value among the entries, i.e. the internal cursor is moved to the first entry that matches the second parameter. The same matching mechanisms which are available for TTCN-3 **receive** statements apply.

The operations **next** and **previous** place the internal cursor to the next matching entry before or after the current cursor position. The order to which **next** and **previous** relate to is imposed by the sorting which resulted from the first parameter of the operation **first**. The parameter of **next** and **previous** is used in the same way as the second parameter of **first**. More complex search operations may be build from these basic search

[7]For the correspondig type, relational operators, like $<$ or $==$, must be defined.

Operation name	*Return type*	*Description*
getlog	**logfile**	Get log file
getverdict	**verdicttype**	Get global verdict
setverdict(*verdict*)	–	Set global verdict

Table 3.3: Overview of *Timed*TTCN-3 Operations for Test Run Handles

operations. The three operations **first**, **next**, and **previous** return **true** when the matching entry is found in the log file, otherwise **false**. The value of the last matched entry, i.e. the value at the current cursor position, can be retrieved by the **retrieve** operation. (The return value is undefined if **first**, **next**, or respectively **previous** returned **false**.) Since the operation **first** restricts the type of the entries, **retrieve** returns values of the same type which was specified by the first parameter of **first**.

The Test Run Handle

For the handling of global log files, *Timed*TTCN-3 introduces the concept of a *test run handle* and thus changes the return type of the **execute** statement: A test run handle is basically a reference of type **testrun** which is returned by the **execute** statement and which gives access to the results of a test run, i.e. the test verdict and global test log.

The operations which can be applied on a test run handle are shown in Table 3.3. The **getlog** operation is used to retrieve the log file of a test run. The operations **getverdict** and **setverdict** are used to retrieve and set the global verdict after a test run. The overwriting of the final test run verdict might be necessary if an offline evaluation shows that a non-functional requirement is not fulfilled. For the **setverdict** operation the same overwriting rules as defined in Section 3.3 apply.

Local Handling of Log Files

*Timed*TTCN-3 allows also to apply the **getlog** function to **self** handles, i.e. a test component may access its own log file in order to perform a local offline evaluation after the collection of time stamps.

The smooth interworking of all *Timed*TTCN-3 concepts for offline evaluation shall be explained by means of the test case example in Section 3.2. For logging test events, the data types shown in Figure 3.9 have been defined: values of type TimestampType will be logged. Its field values describe the log time (Line 86), the timezone of the logging component (Line 87). and the type of the message which causes the log event (Line 88). The message type is described by an ordinary enumeration type Messages (lines 91–93).

Local log file entries are written by the MTC before sending an IDATreq message (Line 26 in Figure 3.5) and by the Responder test component after the reception of a correct MDATind message (Line 51 in Figure 3.6).

```
type record TimestampType {
  float logtime,
  timezones componentzone,
  Messages messagename
}

type enumerated Messages {
  idatreq, mdatind
}
```

Figure 3.9: Data Types Used for Offline Evaluation in the Inres Example

```
123  control {
124    var testrun myTestrun; // Variable for testrun handling
125    var logfile myLog; // Variable for testlog handling
126    var verdicttype myVerdict; // Variable for global verdict
127    myTestrun:=execute(InresRTexample(transmissions), Goettingen);
128    myVerdict:=myTestrun.getverdict; // Retrieval of verdict
129    if (myVerdict==pass) {
130      myLog:=myTestrun.getlog; // Retrieval of testlog
131      myVerdict:=evalMeanInterArrivalTimeOffline(mdatind, Goettingen,
132                        0.01, 0.015, transmissions, myLog); // Offline evaluation
133      myTestrun.setverdict(myVerdict); // Change of testrun verdict
134    }
135  }
136  } // End of module inresRTdistributed
```

Figure 3.10: *TIMED*TTCN-3 Control Part for the Offline Evaluation

Figure 3.10 shows the *TIMED*TTCN-3 module control part of the Inres example real-time test suite. The control part starts with variable declarations for the handling of a test run, a log file, and a verdict value (lines 124–126). The test case inresRTexample is executed with Goettingen as timezone attribute for the MTC (Line 127). The **execute** statement returns a test run handle which is assigned to variable myTestrun. The verdict is retrieved from the test run and stored in variable myVerdict (Line 128).

If myVerdict is **pass** (checked in Line 129), the log file is retrieved (Line 130) and the offline evaluation function evalMeanInterArrivalTimeOffline is called (lines 131 and 132). The actual parameters for the evaluation function are the message identifier mdatind, for which the mean inter-arrival time should be checked, the timezone value Goettingen for the identification of relevant log file entries, the time bounds of 10ms and 15ms that determine the bounds of the requirement to be checked, the module parameter for the number of transmissions that specifies the number of relevant time stamps to examine, and the reference to the log file to be evaluated. At the end, the result of the offline evaluation is assigned to the final verdict of the test run (Line 133).

```
 94 function evalMeanInterArrivalTimeOffline(Messages messageId, timezones zone,
 95            float lowerbound, float upperbound, integer count, logfile timelog)
 96            return verdicttype {
 97   var float timeSum:=0.0, averageArrivalTime;
 98   var TimestampType stampA, stampB;
 99   if (timelog. first (TimestampType:{?,−,−},
100                    TimestampType:{?, zone, messageId})==true) { // Search
101     stampA:=timelog.retrieve; // Get current time stamp entry
102     for (var integer i:=2; i<=count; i:=i+1) {
103       if (timelog.next(TimestampType:{?, zone, messageId})==true) { // Search
104       stampB:=timelog.retrieve; // Get current time stamp entry
105       timeSum:=(stampB.logtime−stampA.logtime)+timeSum;
106       stampA := stampB;
107       }
108       else {
109         return fail; // Wrong number of messages indicates functional problem
110       }
111     }
112     averageArrivalTime:=timeSum/(int2float(count−1));
113     if ((averageArrivalTime<upperbound)
114         and (lowerbound<averageArrivalTime)) {
115       return pass; // Non−functional pass
116     }
117     else {
118       return conf; // Non−functional fail
119     }
120   }
121   return fail; // Wrong number of messages indicates functional problem
122 }
```

Figure 3.11: *Timed*TTCN-3 Offline Evaluation Function

The offline evaluation function evalMeanInterArrivalTimeOffline is shown in Figure 3.11. It implements the mathematical formula for mean inter-arrival time based on the collected time stamps t_i, namely $\left[\sum_{i:=2}^{n}(t_i - t_{i-1})\right]/(n-1)$ and subsequently verifies that the mean inter-arrival time falls within the interval (lowerbound, upperbound).

In order to iterate through the time stamps of mdatind messages, the operations **first** (lines 99 and 100 in Figure 3.11) and **next** (Line 103) are used. Since the **first** operation in lines 99 and 100 sorts the log file by the logtime field, the time stamp entries are matched in ascending order. If **first** or **next** fails, the log file contains less matching time stamps than expected. This is an indication for a non-conforming behaviour of the SUT. Hence, evaluation is aborted with a **fail** verdict (lines 109 and 121).

The **retrieve** operation (lines 101 and 104) yields the value of the last successfully matched entry, which is used to calculate the mean inter-arrival time. Based on the final value of the calculation, the function returns either **pass** or **conf** (lines 115 and 118).

3.5.3 Equivalence of On- and Offline Evaluation

The presented mechanisms for on- and offline evaluation of real-time properties are equivalent as long as it is possible to re-identify time stamps stored in a log file. This can be assured if distinguishable time stamps are generated, e.g. by using message names as labels like in the example (cf. Figure 3.9).

In contrast to online evaluation, offline evaluation has the advantage to separate functional and non-functional requirements. It is even possible to use the time stamps contained in a log file for the assessment of several different real-time requirements.

```
function evalLatencyOffline(Messages messageIdA, Messages messageIdB,
                            timezones zone, float lowerbound, float upperbound,
                            integer count, logfile timelog) return verdicttype {
  var TimestampType stampA, stampB;
  var float latency;
  if (timelog.first(TimestampType:{?,−,−},
       TimestampType:{?, zone, messageIdA})==true) { // Search message A
    stampA:=timelog.retrieve; // Time stamp for message A, i.e. send
    if (timelog.next(TimestampType:{?, zone, messageIdB})==true) {
      stampB:=timelog.retrieve; // Time stmap for message B, i.e. receive
      latency:=stampB.logtime−stampA.logtime;
      if (not((latency<upperbound) and (lowerbound<latency))) {
        return conf; // Non−functional fail
      }
      for (var integer i:=2; i<=count; i:=i+1) {
        if (timelog.next(TimestampType:{?, zone, messageIdA})==true) {
          stampA:=timelog.retrieve; // Time stamp for message A, i.e. send
          if (timelog.next(TimestampType:{?, zone, messageIdB})==true) {
            stampB:=timelog.retrieve; // Time stamp for message B, i.e. receive
            latency:=stampB.logtime−stampA.logtime;
            if (not((latency<upperbound) and (lowerbound<latency))) {
              return conf; // Non−functional fail
            }
          }
          else {
            return fail; // Wrong number of messages: functional problem
          }
        }
        else {
          return fail; // Wrong number of messages: functional problem
        }
      } // End of for loop
      return pass; // Non−functional pass
    }
  }
  return fail; // Wrong number of messages: functional problem
}
```

Figure 3.12: *Timed*TTCN-3 Offline Evaluation Function for Latency

In the Inres example, time stamps are logged for the send (Figure 3.5, Line 26) and receive event (Figure 3.6, Line 51). Hence, it is possible to evaluate the latency real-time requirement not only online as demonstrated in the Inres test case example, but also offline. This is shown in Figure 3.12.

The structure of the latency offline evaluation function is quite similar to the offline evaluation function for the mean inter-arrival time (Figure 3.11). In lines 6–8 of Figure 3.12, the time stamp for the first send event is obtained. Then the time stamp for the corresponding receive event[8] is retrieved (lines 9 and 10). If the latency inequations do not hold for that pair of events, the evaluation function terminates and the verdict **conf** is returned to indicate a non-functional failure (lines 11–14). These steps are repeated for count entries of the log file by the **for** loop in lines 15–32. If all entries are valid, the **pass** verdict is returned (Line 33). In case of any problem, evaluation terminates and the verdict **fail** is returned, because these problems result from non-matching time stamps in the log file. For example, to allow the comparison of a pair of time stamps, they must have the same timezone attribute which is determined by the formal parameter zone.

Figure 3.12 demonstrated that it is possible to replace online evaluation by offline evaluation. Now, it shall be demonstrated how evaluation of the mean inter-arrival time requirement can be performed online.[9]

When trying to evaluate soft real-time requirements online, variables which keep their values between subsequent function calls are needed. Such "static" variables are not supported by TTCN-3. However, a simple workaround for "static" variables is to declare such variables as component variables.

Figure 3.13 depicts in lines 1–7 the definition of a Responder component type which contains additionally three variables (lines 4–6) which serve as "static" variables. The usage of these variables is demonstrated by function evalMeanInterArrivalTimeOnline in lines 9–29.

Variable count is used to count how often the online evaluation function was called. If it is called for the first time, inter-arrival times cannot be computed, since at least two arrivals are required for that. In this case, the other variables are hence only initialised and the **none** verdict is returned (lines 13–17 of Figure 3.13). For all subsequent calls, the inter-arrival time for the last pair of events can be calculated (Line 19) as well as the mean inter-arrival time for all past events (Line 21). The used mathematical formula for calculating after the i^{th} iteration the mean value m_i on-the-fly is: $m_i := \frac{m_{i-1} \cdot (i-1) + x_i}{i}$, where x_i is the sample obtained in the i^{th} itera-

[8]In fact, the function does not care what kind of events it evaluates as long as the message name field of the time stamp matches the parameters messageIdA and messageIdB.

[9]It has to be considered that, e.g. during the i^{th} iteration ($i \in [2 \ldots n]$), the arithmetical mean might be out of bounds. However, after the n^{th} iteration, the situation might have changed due to averaging. Thus, checking mean values after each iteration differs semantically from checking a mean value only after the last iteration.

```
1  type component ResponderTypeWithMemory {
2    port MediumSAP MSAP;
3    port CoordinationPoint CP;
4    var float previousArrivalTime;
5    var float mean;
6    var integer count:=0;
7  }
8
9  function evalMeanInterArrivalTimeOnline(float arrivalTime, float lowerbound,
10   float upperbound) runs on ResponderTypeWithMemory return verdicttype {
11   var float interArrivalTime;
12   count:=count+1;
13   if (count==1) { // For inter-arrival time, at least two arrivals are required
14     previousArrivalTime:=arrivalTime;
15     mean:=0.0;
16     return none;
17   }
18   else {
19     interArrivalTime:=arrivalTime-previousArrivalTime;
20     previousArrivalTime:=arrivalTime;
21     mean:=mean*int2float(count-1)+interArrivalTime/int2float(count);
22     if ((mean<upperbound) and (lowerbound<mean)) {
23       return pass; // Non-functional pass
24     }
25     else {
26       return conf; // Non-functional fail
27     }
28   }
29 }
```

Figure 3.13: *Timed*TTCN-3 Mean Inter-arrival Time Online Evaluation

tion. Based on the calculated mean inter-arrival time, either a **pass** or **conf** verdict is returned (lines 22–27).

The two above examples demonstrate that the given real-time requirements may be evaluated using either on- or offline evaluation. Moreover, time stamps from the same log file may be used to assess various real-time properties using offline evaluation functions. While for online evaluation, the time stamps are explicitly passed as parameters to the evaluation function, for offline evaluation, the time stamps have to be re-identified in the log file. Therefore, the specification of offline evaluation functions usually requires more lines of *Timed*TTCN-3 code than online evaluation.

The idea is to provide a module of predefined time stamp type definitions and matching evaluation functions in order to facilitate the usage of *Timed*TTCN-3. In this way, a real-time test case developer just needs to select the appropriate evaluation function from the predefined library and instrument the test case accordingly.

3.6 Graphical Presentation of *Timed*TTCN-3

The *Graphical Presentation Format for TTCN-3* (GFT) [ETS03a] is one of the standardised presentation formats of TTCN-3. It provides an exact way of graphically displaying TTCN-3 behaviour specifications, i.e. test cases, altsteps, functions, and module control. GFT is based on the MSC language [ITU99b]. It uses a subset of MSC and extends this subset with test specific symbols and keywords. Appropriate GFT visualisations exist for all behavioural TTCN-3 statements.

It is desirable to be able to present *Timed*TTCN-3 behaviour specifications graphically as well. This is achieved by *Timed*GFT, a corresponding real-time extension of GFT. *Timed*GFT has been published jointly in [DGN03]. In the following, just a brief introduction on *Timed*GFT and its relation to *Timed*TTCN-3 is given. A more elaborated presentation is about to appear in one of the co-authors' PhD thesis [Dai05].

Since GFT already covers all behavioural statements of the TTCN-3 core notation, only the real-time extensions of *Timed*TTCN-3 have to be consid-

***Timed*TTCN–3**		***Timed*GFT**
Concept	Realisation	Presentation
Timezones	new parameter of **create** statement	MyTC:=CType.**create**(Berlin)
	new parameter of **execute** statement	MyTestCase(Berlin)
	timezone operation	no special symbol
Local time	**now** operation	no special symbol
	resume statement	@[t+3.0]
Logging	extension of **log** statement	MyTemplate
Log file handling	**first, next, previous** and **retrieve** operations	no special symbols
Test run handling	**getlog** operation	no special symbol
	overwriting of verdicts in control part	myTestrun. **setverdict(fail)**
Non–functional verdict	**conf** verdict	**conf**

Table 3.4: Real-Time Constructs of *Timed*GFT

ered in order to define *Timed*GFT. This is achieved by introducing additional symbols and enriching existing GFT symbols. Table 3.4 provides an overview on the graphical presentation of the *Timed*TTCN-3 extensions.

Figure 3.14 shows a *Timed*GFT representation of the *Timed*TTCN-3 test case example from Figure 3.5, i.e. this *Timed*GFT diagram describes the MTC behaviour. The majority of symbols contained in this diagram are standard GFT symbols: Test components are displayed as MSC instances, ports are depicted similarly to instances but with a dashed life line. Variable declarations, mapping and connecting of ports are displayed as MSC action boxes. For default activation and deactivation, creation and start of components, variants of the action symbol are used. Function call is visualised as MSC reference. Operations on ordinary TTCN-3 timers are mapped one-to-one on MSC timer symbols. Communication operations are represented as messages between the test component instance and its port instances. Loops, but also other control flow related statements like, e.g. alternatives, are mapped to variants of MSC inline expressions. Waiting for termination of PTCs, but also setting of verdicts are visualised as MSC condition.

In the remainder, the additions which are contributed by *Timed*GFT to GFT are presented. The description is structured according to the order of appearance of the *Timed*TTCN-3 real-time concepts in Figure 3.14.

3.6.1 Timezones

*Timed*TTCN-3 allows to assign timezones to test components during their creation. Both, GFT and *Timed*GFT depict the creation of test components by the same create or respectively execute symbol. Thus, the assignment of a timezone is just an additional parameter in a create or execute symbol.

A test component may retrieve its timezone attribute using the **timezone** operation. *Timed*GFT provides no special symbol for the **timezone** operation. Depending on the usage, the **timezone** operation may appear in different symbols. For example, a **timezone** operation will appear in an action box if it is used in an assignment, or a **timezone** operation will be presented within a reference symbol if it defines the actual parameter of an altstep or function call. The *Timed*GFT representation of both, a *Timed*TTCN-3 **create** statement and a **self.timezone** operation, can be found in the upper third of Figure 3.14 where the responder test component is created.

3.6.2 Absolute Time

For the **now** operation, no special symbol is provided by *Timed*GFT. Instead, the **now** operation appears in different symbols depending on its usage, e.g. as inscription in action boxes or log symbols. In Figure 3.14,

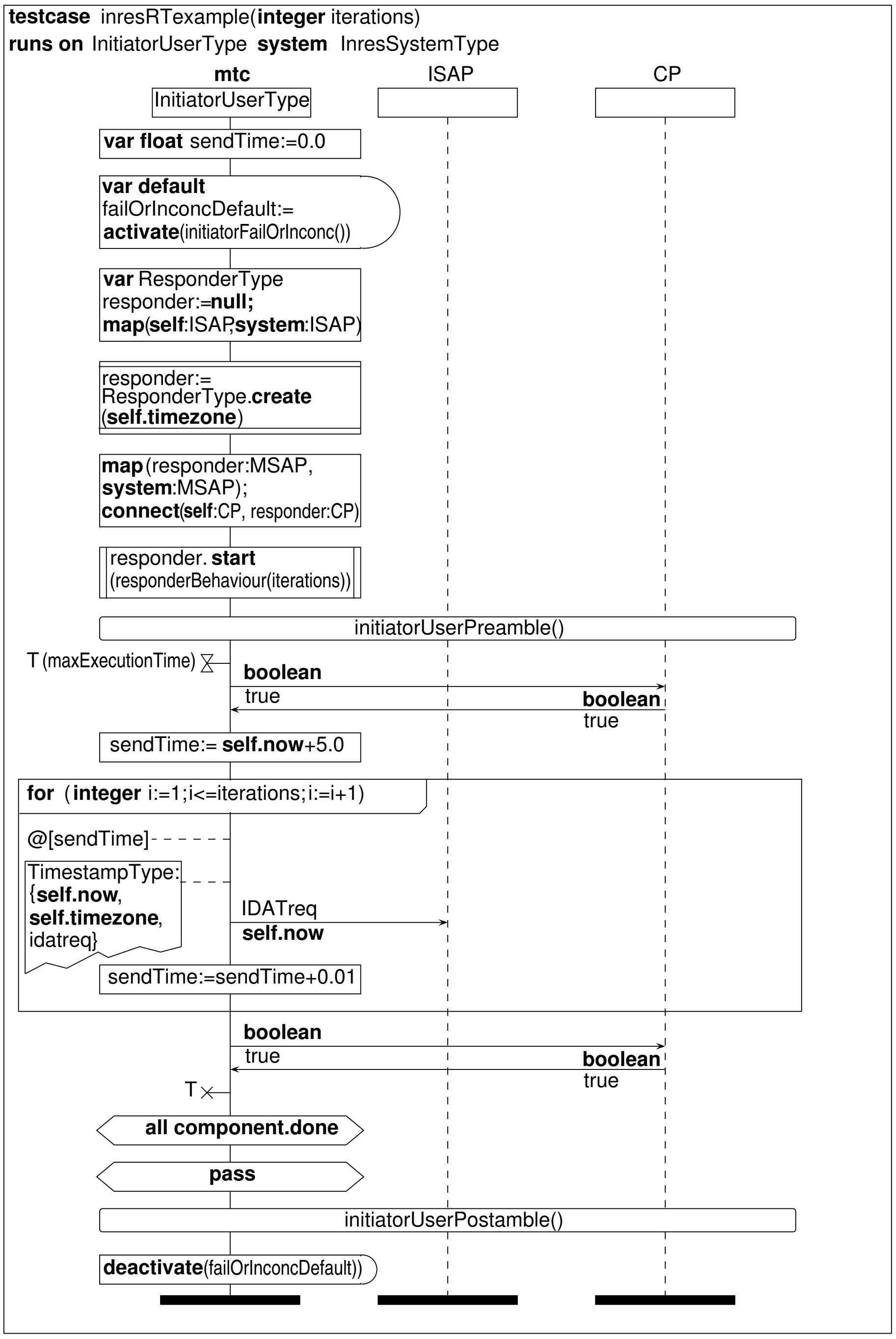

testcase inresRTexample(**integer** iterations)
runs on InitiatorUserType **system** InresSystemType
mtc
InitiatorUserType
ISAP
CP
var float sendTime:=0.0
var default failOrInconcDefault:= **activate**(initiatorFailOrInconc())
var ResponderType responder:=**null;** **map**(**self**:ISAP,**system**:ISAP)
responder:= ResponderType.**create** (**self.timezone**)
map(responder:MSAP, **system**:MSAP); **connect**(**self**:CP, responder:CP)
responder. **start** (responderBehaviour(iterations))
initiatorUserPreamble()
T (maxExecutionTime)
boolean true
boolean true
sendTime:= **self.now**+5.0
for (**integer** i:=1;i<=iterations;i:=i+1)
@[sendTime]
TimestampType: {**self.now**, **self.timezone**, idatreq}
IDATreq **self.now**
sendTime:=sendTime+0.01
boolean true
boolean true
T
all component.done
pass
initiatorUserPostamble()
deactivate(failOrInconcDefault))

Figure 3.14: *Timed*GFT Representation of Test Case from Figure 3.5

the **self.now** operation is used in an assignment, in a **log** inline template, and as message template parameter. Therefore, it appears in an action box, inside the log symbol, and as message parameter.

For presenting the **resume** statement graphically, *Timed*GFT adopts the absolute time constraint symbol of MSC. (In Figure 3.14, the visualisation of **resume**(sendTime) can be found just inside the beginning of the **for** loop.) But, contrary to MSC, the dashed time line is attached directly to an instance, not to an event. The reason is that the **resume** statement is a statement on its own and not related to other events.

3.6.3 Logging

In the GFT standard, **log** statements are presented in action boxes. Nevertheless, *Timed*GFT introduces a new log symbol which resembles a paper-based log. The reason for this new symbol is that places in the test behaviour, where time and other information are collected in the log file, shall be emphasised. Figure 3.14 presents below the resume symbol the new symbol which logs a time stamp of type TimestampType using an inline template.

3.6.4 Test Run and Log File Handling

In the module control part[10], *Timed*TTCN-3 grants access to the global test verdict and the log file by a test run handle which is returned by the **execute** statement. Like for the **now** operation, the context of the **getlog** operation determines the symbol in which it is presented. If the **getlog** operation is, e.g., used in an assignment, it will be presented inside an action box.

The *Timed*TTCN-3 functions **first**, **next**, **previous**, and **retrieve** have no special *Timed*GFT presentation. Just like for the **getlog** operation, their presentation depends rather on the context in which they are applied.

3.6.5 Verdict Handling

The handling of verdicts in *Timed*GFT is almost identical to their handling in GFT. When setting a local verdict inside a test component, the argument of the **setverdict** operation is presented in an MSC condition symbol.

However, the presentation of the **setverdict** operation differs from GFT if the global verdict is set. This is used inside the module control part when applying the **setverdict** operation to a test run handle. In this case, *Timed*GFT displays not just the argument of the **setverdict** operation, but the complete *Timed*TTCN-3 statement including the test run handle to which the **setverdict** operation is applied. Otherwise, the relation between verdict and test run handle would not be clear.

There exists no special symbol to emphasise the **getverdict** operation. The context determines rather the symbol in which it is presented.

[10]For the graphical presentation of a module control part, GFT provides a control diagram, which includes one control instance only.

3.7 Tabular Presentation of *Timed*TTCN-3

In addition to GFT, the set of standardised presentation formats for TTCN-3 consists of the *Tabular Presentation Format* (TFT) [ETS02b]. It allows to represent TTCN-3 as tables which resemble the *Tree and Tabular Combined Notation* (TTCN) [ISO97b], the predecessor of TTCN-3. In contrast to GFT, not only the behavioural elements of the TTCN-3 core notation can be visualised using TFT, but in particular also data type and template definitions. No modifications of TFT are required for the tabular presentation of *Timed*TTCN-3 test suites. This is discussed in the following paragraphs.

3.7.1 Behaviour

In TFT, the actual behaviour description of test cases, functions, altsteps, and module control are presented textually as ordinary TTCN-3 core language inside the **behaviour** area of the corresponding table. Just static information like formal parameters, component type, and local variables definitions are displayed in special sections of the table. Hence, all behavioural add-ons of *Timed*TTCN-3, like **self.now**, **resume**, **first**, **retrieve**, **getlog**, are presented as textual core language. This includes also creation of clock synchronised components, since this is just an additional parameter of the **create** operation or respectively the **execute** statement, which are both part of the behaviour description. Therefore, no extension of TFT's behavioural tables is necessary. Nevertheless, to get an impression of how a behavioural TFT table looks like, Table 3.5 provides the tabular presentation of the *Timed*TTCN-3 Inres example test case from Figure 3.5. All additional statements of *Timed*TTCN-3 are contained as textual notation inside the behavioural area in the lower half of the table.

3.7.2 Types and Values

In contrast to behaviour representation, type, template, and constant definitions are presented by TFT more closely to the tabular appearance of TTCN and thus, much more detached from the TTCN-3 core language. However, since *Timed*TTCN-3 time stamps are ordinary data types, their definition can be visualised using the standard tables for presenting data types in TFT. This is also valid for the definition of the reserved enumeration type **timezones**. Furthermore, the template definitions which are used for matching log file entries using **first**, **next**, and **previous** can be displayed as ordinary TFT template tables, too. Thus, although TFT uses very specialised tables for types and values, they need not to be modified for representing *Timed*TTCN-3 definitions.

Testcase			
Name	inresRTexample(**integer** iterations)		
Group			
Purpose			
System Interface	InresSystemType		
MTC Type	InitiatorUserType		
Comments			
Local Def Name	**Type**	**Initial Value**	**Comments**
sendTime	float	0.0	
failOrInconcDefault	default	**activate** (initiatorFailOrInconc())	
responder	ResponderType	null	
Behaviour			

map(**self**:ISAP, **system**:ISAP);
responder:=ResponderType.**create**(**self.timezone**); // Create in same timezone
map(responder:MSAP, **system**:MSAP);
connect(**self**:CP, responder:CP);
responder.**start**(responderBehaviour(iterations));
initiatorUserPreamble();
T.**start**(maxExecutionTime); // Detect blocking of test case
CP.**send**(**boolean**:**true**);
CP.**receive**(**boolean**:**true**);
sendTime:=**self.now**+5.0; // Send for the first time in 5.0s from now
for(**var integer** i:=1; i<=iterations; i:=i+1) {
   resume(sendTime); // Wait until 'sendTime'
   log(TimestampType:**self.now**, **self.timezone**, idatreq); // Log timestamp
   ISAP.**send**(IDATreq:**self.now**); // Piggyback send time
   sendTime:=sendTime+0.01; // Send periodically every 10ms
}
CP.**send**(**boolean**:**true**);
CP.**receive**(**boolean**:**true**);
T.**stop**;
all component.done;
setverdict(**pass**);
initiatorUserPostamble();
deactivate(failOrInconcDefault);			

Detailed Comments			

Table 3.5: TFT Representation of Test Case from Figure 3.5

3.8 Summary

In this chapter, *Timed*TTCN-3, a real-time extension for TTCN-3 has been introduced. Its usage was demonstrated by applying it to testing of real-time properties which were imposed on the Inres protocol. By introducing absolute time for test components, *Timed*TTCN-3 allows to wait until an absolute point in time and to collect time stamps. Time stamps may be evaluated online during a test run or offline after a test run. Offline evaluation allows to separate the description of functional and non-functional requirements which are subject of a test case. For offline evaluation, *Timed*TTCN-3 offers a flexible log mechanism with local and global log files. In particular, *Timed*TTCN-3 supports to store and retrieve time stamps in or respectively from log files. The log mechanism also enables an evaluation of non-functional properties which are not real-time related. For example, failure rates for transmitted data packets can be checked offline by logging correct as well as erroneous message receptions without any time information. *Timed*TTCN-3 can also be used for distributed test architectures, since it supports the specification of synchronisation requirements for clusters of clock-synchronised test components. This allows to compare time stamps captured at different, but clock-synchronised test components.

Even though *Timed*TTCN-3 was developed with having hard real-time requirements in mind, it can also be used to test soft real-time requirements as long as they apply to a discrete set of events. An example was provided, which demonstrates the assessment of statistical properties like mean inter-arrival times.

Furthermore, it is was shown how *Timed*TTCN-3 can be displayed using the existing TTCN-3 presentation formats: *Timed*GFT has been introduced as an extension of GFT which allows a graphical specification and presentation of real-time test cases. For the tabular presentation format (TFT), it was demonstrated that no extension is necessary for the representation of *Timed*TTCN-3 real-time test suites.

For supporting *Timed*TTCN-3, only a few changes to the TTCN-3 language are needed. They have been documented and submitted to ETSI as change requests for TTCN-3 and GFT [Neu02, Dai03]. *Timed*TTCN-3 is downwardly compatible to TTCN-3. Hence, existing test suites may be reused and instrumented for generating time stamps. The only break of compatibility is due to the modified return type of the execute statement. Thus, the control part of an existing TTCN-3 module might require modifications, which can be performed automatically by a tool, however.

The formal semantics of the new *Timed*TTCN-3 constructs was not presented. Indeed, most *Timed*TTCN-3 extensions can be explained by an add-on to the existing formal semantics of TTCN-3. Only the concept of absolute time in combination with the notion of clock-synchronised components and the delaying of execution requires a new real-time semantics. These features allow the description of time dependencies among test components, i.e. absolute time values influence the behaviour in different test components. For all other statements, the existing, untimed semantics [ETS03b] remains valid.

*Timed*TTCN-3 is also intended to be used as real-time test implementation language. Thus, a *Timed*TTCN-3 runtime system benefits from real-time support by the *TTCN-3 Runtime Interface* (TRI) and *TTCN-3 Control Interface* (TCI). For example, the TRI Platform Adaptor might provide access for reading the local clock of a test component. Since logging facilities and test component creation are provided by the TCI, the log file handling and the timezone concept of *Timed*TTCN-3 may benefit from an enhanced TCI. Currently, it is, e.g., under discussion to define an *Extensible Markup Language* (XML) [W3C04] format for TTCN-3 log files, which would facilitate an automated access to the log file content.

From the methodological aspect, it is suggested to provide a *Timed*TTCN-3 module of predefined time stamp type definitions and evaluation functions to ease development of real-time tests. Further support for real-time test case development, like computer aided real-time test case generation or creating real-time test cases from patterns, is subject of the remaining chapters of this thesis.

Related Work

Real-time extensions for TTCN as well as a formalised evaluation of log files in general are not new. For example, [UHPB03] presents an approach for a formalised analysis of log files: Log files of a distributed system are obtained by monitoring and later-on transformed into a formal SDL [ITU99a] representation which can be checked by a model checker. Unfortunately, this approach involves several activities which cannot be specified and automatically executed as part of a test case. Furthermore, it does not address real-time properties but trustworthiness requirements.

For testing real-time requirements, two extensions for TTCN-2, the predecessor of TTCN-3, have been proposed in the past: *PerfTTCN* and *RT-TTCN*.

PerfTTCN [SSR97] extends TTCN-2 with concepts for performance testing. These concepts are: (1) performance test scenarios for the description of test configurations which include, e.g., load generator components for fore- and background load, (2) traffic models for the description of discrete and continuous streams of data, (3) measurement points as special observation points, (4) measurement declarations for the definition of metrics to be observed at measurement points, (5) performance constraints to describe the performance conditions that shall be met, and (6) performance verdicts for the judgement of test results.

The PerfTTCN concepts are introduced mainly on a syntactical level by means of new TTCN tables. Their semantics is described in an informal manner only and in some cases the implementation of these concepts is not clear. For example, traffic models may be declared inside a PerfTTCN test suite but they turn out as comments for a PerfTTCN compiler because their implementation is outside the scope of the language. The traffic models may serve as input parameters for external load generator components and it can be discussed, if such information should be part of the compilable test suite or part of accompanying documents.

[GKS00] describes how the concepts of PerfTTCN can be implemented without any language extensions: For example, external load generators may be controlled from within TTCN-2 by control messages which are passed to the load generating device by an adaptor port. This approach can also be used in *Timed*TTCN-3 to establish some background load to obtain a realistic environment with definable, reproducible loads for the SUT. Alternatively, [VGDS04] proposes an extension for TTCN-3 to call external command line utilities from within a test case. This facility may be used to call a software-based load generator. Finally, it is possible to implement explicitly a load generator on a test component using *Timed*TTCN-3 statements.

The second real-time extension proposed for TTCN-2 is RT-TTCN [WG97, WG99]. It is intended for testing hard real-time requirements. On the syntactical level, RT-TTCN supports the annotation of TTCN-2 statements with a time interval for earliest and latest execution times. On the semantical level, the TTCN-2 snapshot semantics has been refined. In addition, RT-TTCN has been mapped onto timed transition systems [HMP91].

Even though the RT-TTCN time extension is introduced formally and looks very simple, it turned out that its usage is not that simple. The time points associated to a statement are relative to the occurrence of previous events and define a time interval in which the statement is activated, i.e. can be executed. The handling of these activation intervals is not intuitive, especially, if several statements can be executed, i.e. their activation intervals overlap. For the user it seems to be more natural to record execution times and to compare their values afterwards than to define activation intervals for TTCN-2 statements.

Experiments give evidence that *Timed*TTCN-3 covers most of the PerfTTCN and RT-TTCN features while being more intuitive in usage. Moreover, the *Timed*TTCN-3 extensions are more unified than the other extensions by making full use of the expressiveness of TTCN-3.

Chapter 4

Generation of Real-Time Test Cases

Manual test development is a time consuming task. Furthermore, manual activities are error-prone. In testing, this may lead to situations where it has to be decided whether a fail verdict was assigned due to defects in the *Implementation Under Test* (IUT) or due to errors made during test development. Thus, test development benefits from tool support.

Figure 4.1 depicts the test process described in Section 2.1.1 and indicates the different areas which may profit from tooling. If a formal specification for the IUT exists, corresponding test purposes may be automatically derived from the specification, e.g. based on some coverage criteria. For *Specification and Description Language* (SDL) specifications [ITU99a], this approach has been successfully applied [SEG$^+$98, KJG99, Koc01, Sch03]. However, SDL allows only to describe functional behaviour. A comparable, industrial strength formal language for the specification of real-time behaviour did not prevail, yet. The area of computer aided derivation of real-time test purposes based on formal specifications is therefore not covered in this thesis.

However, for the next activity, namely real-time test case development, a popular formalism exists. Real-time test purposes can be formalised as *Message Sequence Chart*s (MSCs) and used as input for a tool. Thus, computer aided generation of *Timed*TTCN-3 real-time test cases is possible. A suitable approach for this is presented in this chapter.

The remaining testing activities can as well be automated by tools. Making an *Abstract Test Suite* executable can be performed by compilers [Dan04, Tel04, Tes04, Ope04], and test execution can be supported by runtime systems which implement the *TTCN-3 Runtime Interface* (TRI) and *TTCN-3 Control Interface* (TCI). The formalised evaluation of test log files using *Timed*TTCN-3 evaluation functions is described in the previous chapter.

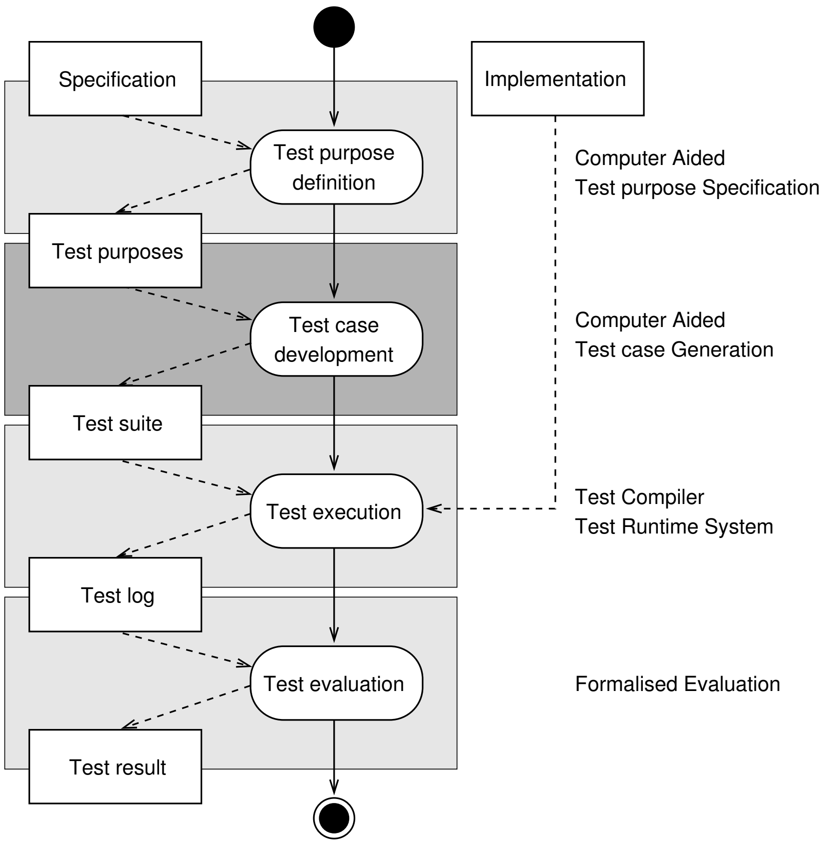


Figure 4.1: Possible Tool Support for Black-Box Testing Activities

This chapter is about generating *Timed*TTCN-3 real-time test cases from formal MSC real-time test purposes. In Section 4.1, it is explained how real-time test purposes can be specified by means of MSC. Then, in Section 4.2, an interpretation of real-time test purpose MSCs and the automated transformation into *Timed*TTCN-3 real-time test cases are presented. This includes both, established concepts for deriving functional test cases and a novel approach for obtaining real-time test cases. The next section discusses real-time test generation from test purpose MSCs for distributed test architectures. Finally, a summary of this chapter is given in Section 4.4. This chapter is based on the author's work that has been published as part of [DGN03, GNS+02].

4.1 MSC-based Test Purpose Specification

The usage of MSC [ITU99b] as a graphical language for test purpose specification, but also for test behaviour visualisation is popular and has been thoroughly investigated by several authors [GHN93, GW98, KJG99, BRS01, BBJ$^+$02, Koc01, Sch03, Ebn04]. However, these authors only treated the usage of MSC for describing pure functional behaviour, but not for testing real-time properties.

4.1.1 Test Purpose vs. Test Behaviour Visualisation

MSC-based real-time test purpose specification has to be distinguished from visualising real-time test behaviour using MSC or *Timed*GFT respectively. The difference between test purposes[1] described in form of MSCs and a test behaviour visualised by *Timed*GFT diagrams are the different levels of abstraction and points of view. The relations between test purposes and the behaviour of test cases are shown in Figure 4.2. *Timed*GFT is described in Section 3.6, thus this chapter is only concerned with the left half of the figure below.

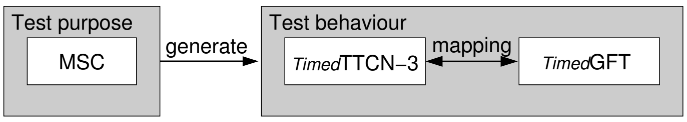


Figure 4.2: MSC Test Purposes, Test Behaviour and *Timed*GFT Diagrams

An MSC test purpose is an abstract description of a test. It describes the test from the perspective of the *System Under Test* (SUT) and makes no assumptions about the implementation of the test, e.g. the used test architecture. Only after providing additional information about the test architecture, it is possible to generate *Timed*TTCN-3 test behaviours from MSC test purpose specifications. In contrast, test behaviour descriptions define tests from the perspective of the test system. They are written for a specific test architecture and include all the activities to coordinate the test components and evaluate the test result.

While test behaviour visualisation requires extension of MSC like those provided by GFT, MSC is very well suited for the formal specification of test purposes. Industrial tools exist which allow an automated generation of TTCN-2 test cases from MSC test purposes [GKSH99, BBJ$^+$02]. These experiences from functional testing serve as a foundation for the *Timed*TTCN-3 real-time test case generation approach presented in this chapter.

[1] In the following, the terms *test purpose* and *test case* are used to denote also *real-time test purposes* and *real-time test cases*. However, to emphasise the real-time aspect, the latter terms are still used in some places of this chapter.

4.1.2 An Inres-based Example

In the approach presented in this chapter, the common practise to specify test purposes by *system level* MSCs is used. A system level MSC does not show internals of an SUT, but has just one designated instance representing the SUT. All other instances correspond to the different interfaces of the SUT. Together with the contained message exchange from and to the interfaces of the SUT, this information can be used to derive corresponding test cases. By taking MSC time annotation into account, even real-time test cases can be generated this way.

This shall be demonstrated by an example. The IUT is an Initiator implementation of the *Inres* protocol. The SUT can be accessed by using the interfaces ISAP and MSAP. The test purpose is to test for each of 100 data transfers which are initiated consecutively every 10ms that the latency is below 5ms. For doing this, first, a connection needs to be established, and after the test, the connection has to be released.

A formalisation of this test purpose in form of an MSC is shown in Figure 4.3. The MSC describes the test purpose from the point of view of the SUT, i.e. only the required information exchange at the ISAP and MSAP service access points is shown. The test purpose includes no assumptions about implementation of the test system, e.g. the number of test components and the required synchronisation among test components are not specified.

The *Timed*TTCN-3 module containing the real-time test case inresRTexample shown in Figure 4.6 is generated automatically from the test purpose

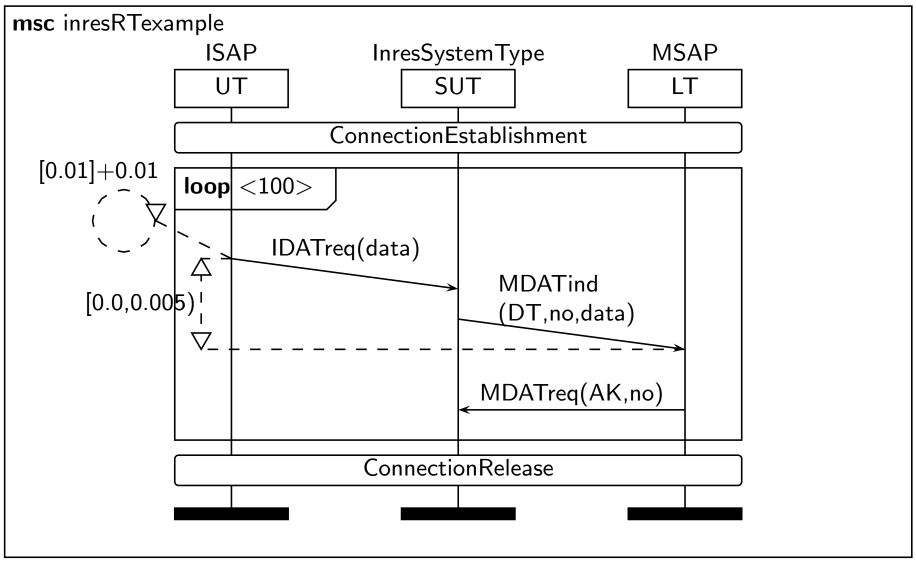


Figure 4.3: Real-Time Test Purpose for Inres

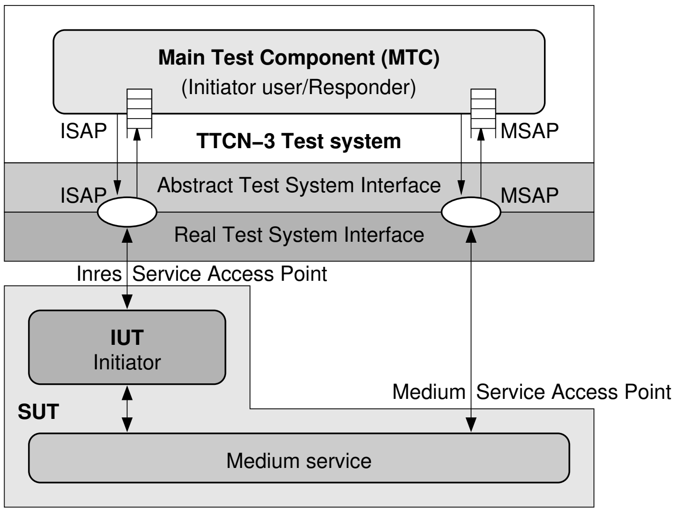

Main Test Component (MTC)
(Initiator user/Responder)
ISAP
TTCN–3 Test system
MSAP
ISAP
Abstract Test System Interface
MSAP
Real Test System Interface
Inres Service Access Point
IUT
Initiator
SUT
Medium Service Access Point
Medium service

Figure 4.4: Local Test Architecture Used for Test Case Generation

example in Figure 4.3 and the local test architecture provided in Figure 4.4.[2] The test architecture consists of the MTC only, which controls both interfaces of the SUT.

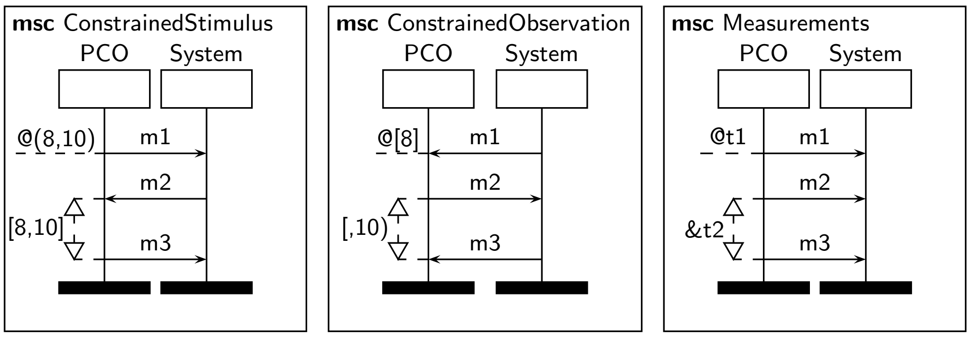
4.2 Test Generation from MSC Test Purposes

In the following, it is explained how MSCs specifying real-time test purposes can be interpreted and used for automatically generating *Timed*TTCN-3 test cases. For a concise presentation, this section is restricted to test generation for non-distributed test architectures.

4.2.1 Interpretation of MSC Test Purposes

Even when using the formal representation of MSC for expressing a test purpose, different interpretations of a test purpose MSC are possible. For example, the approach described in [BBJ$^+$02] generates one test case for each of the traces which are possible due to the partial order semantics of MSC. In contrast, the approach presented in this chapter extracts just one single representative *path* from the test purpose MSC, by taking the queue semantics of *Timed*TTCN-3 into consideration. (More details are explained in Section 4.2.2.) However, e.g., the interpretation of *non-SUT instances*, i.e. all MSC instances except for the one named SUT, as interfaces of the SUT to its environment is a common interpretation of test purpose MSCs.

[2]Figure 4.4 depicts the same test architecture which is described on Page 32. The figure is a copy of Figure 2.27 and just provided here to enhance the readability of this chapter.



(a) Time Constraints (b) Time Constraints (c) Time Measurements

Figure 4.5: Time Constructs Applicable in Real-Time Test Purpose MSCs

The usage of MSCs for generating *real-time* test cases is a novel approach. Thus, no common interpretation of real-time test purpose MSCs exists, yet. Though, the proposed interpretation of MSC time annotations used in test purposes is intuitive and straightforward. Their meaning is discussed in the following by referring to the MSCs explained in Figure 2.11 on Page 21. For a better readability, a copy of them is provided in Figure 4.5.

In real-time test purpose MSCs, it is allowed to attach MSC time constraints and measurements to the communication events along the non-SUT instances. Time constructs attached to events on the SUT instance axis are not observable and thus make no sense for test specification.[3] Therefore, such time constructs cannot be translated.

While the meaning of MSC time measurements (Figure 4.5c) in test purpose descriptions is straightforward (namely, observation of the point in time when an event occurs and its storage in a *Timed*TTCN-3 variable), MSC time constraints can be used with two different aims: They may either describe a timely stimulation of the SUT, i.e. a real-time requirement rather on the test system (*time constrained stimulus*), or a response from the SUT that shall arrive within a certain period of time (*time constrained observation*). Both cases look similar, but can be distinguished as follows:

Time constrained stimuli have to be performed by the test system if absolute time constraints are attached to send events, or relative time constraints are attached to a pair of events, where the second event is a send event[4] (cf. Figure 4.5a).

Time constrained observation have to be performed by the test system if absolute time constraints are attached to receive events, or relative

[3]The same is valid for non-communication events, like MSC actions.

[4]The type of the first event involved in the relative time constraint is irrelevant, since the time constraint is essentially imposed on the second event.

time constraints are attached to a pair of events, where the second event is a receive event (cf. Figure 4.5b).

In the MSC test purpose example (Figure 4.3), both types of relative time constraints are used. The cyclic real-time requirement constraint can be unrolled to a sequence of relative time constraints which require to send a stimulus every 10ms.[5] The second time constraint is attached to the messages IDATreq and MDATind and describes thus a time constrained observation of MDATind.

With respect to MSC time measurements and MSC time constraints which are used for time constrained observation, it has to be noted that their occurrence in a real-time test purpose MSC results in a similar test behaviour: In both cases, the time when the involved event occurred has to be retrieved by the test system. But, only time constrained observations allow to attach actual boundary values of a real-time requirement to the graphical MSC symbols. In contrast, an actual real-time requirement cannot be specified by MSC time measurements, because they only allow to specify names for observations, but do not impose concrete requirements on time values. As a result of these considerations, MSC time constraints are the preferred means for specifying real-time requirements in test purposes. Nevertheless, MSC time measurements may be useful for gathering time information, which is reused later-on, e.g. as part of other MSC time constraints.

Based on the given interpretation of real-time constructs in test purpose MSCs, suitable transformation rules for obtaining *Timed*TTCN-3 test cases are presented in the remainder of this chapter. These rules have been implemented by the author in a prototype tool which accepts machine processable textual MSC format as input and generates *Timed*TTCN-3 core notation as output. It is based on a tool described in [Ebn04] which allows to transform pure functional MSC test purposes into TTCN-3. Thus, before the transformation of real-time concepts is explained in Section 4.2.3, the underlying transformation of functional concepts is briefly described, first.

4.2.2 Transformation of Functional Concepts

The transformation of static aspects of an MSC test purpose into TTCN-3 is simple. The name of the MSC is taken as test case name and as a suffixed module name. (In the example, lines 1 and 4 of Figure 4.6 are derived from the MSC name inresRTexample in Figure 4.3.)

The interfaces of the SUT are described by non-SUT MSC instances. They are mapped to TTCN-3 ports, i.e. the MSC instance type is taken as name of a TTCN-3 port instance. (The name of those instances is ignored, though.)

[5]In test purpose MSCs, the data language of *Timed*TTCN-3 is used, thus 10ms has to be written as 0.01.

The type of the SUT instance is taken as TTCN-3 component type for the Abstract Test System Interface. (The **runs on** InresSystemType specification in Line 4 of Figure 4.6 is generated from Figure 4.3. As well, the TTCN-3 port instances used for sending and receiving in lines 18, 19, and 21 of Figure 4.6 are derived from the MSC instance types in Figure 4.3.)

For the mapping of MSC events to TTCN-3 statements, only the events along non-SUT instances are relevant. MSC send events on those instances are mapped to TTCN-3 **send** operations and MSC receive events are mapped onto TTCN-3 **receive** operations.[6] The MSC message names are expected to refer to TTCN-3 data types or, in case of procedure-based communication, to signature definitions. MSC message parameters refer to TTCN-3 templates or define inline templates. (The generated communication operations are located in lines 18, 19, and 21 of Figure 4.6).

For the component types and contained ports, but also for message types and templates, corresponding definitions are required. However, these cannot be generated from MSC test purposes. They have to be specified manually or imported from other TTCN-3 modules. For the Inres example, the type definition modules presented in Section 2.5.1 might be imported.

In addition to the generation of basic communication operations from test purpose MSCs, timers, actions, and local conditions on interface instances are supported. MSC timer events have a one-to-one mapping to TTCN-3 timer operations. MSC actions can be used to specify additional test behaviour in form of TTCN-3 statements contained in action boxes. Local setting conditions are used for the specification of test verdicts, i.e. they are translated into a **setverdict** operation. Non-local conditions in MSC test purposes can be used to specify synchronisation points explicitly, i.e. force a specific order of test events.

An MSC test purpose specification may also include references and the inline expressions **alt**, **loop**, and **par**. Each usage of a reference or an inline expression is considered as one single event, which in case of references or inline expressions may include partially ordered events. This means, these constructs are synchronisation points, even though this violates the official MSC semantics [ITU99b] which assumes a weak sequential composition. Indeed, many test case specifiers regard this as counter-intuitive anyway. Hence, the corresponding TTCN-3 constructs generated by the suggested transformation rules do not allow that sort of interleaving. The transformation algorithm maps MSC references to TTCN-3 function calls (lines 8 and 23 of Figure 4.6), MSC **alt** inline expressions to TTCN-3 **alt** statements[7], MSC **par** inline expressions to TTCN-3 **interleave** statements, and

[6]The generation of procedure-based communication behaviour is possible as well, since both MSC and TTCN-3 distinguish between the two types of communication.

[7]This means, each branch of an MSC alternative must start with an observation, otherwise it describes a non-deterministic test case since late choice is only possible in MSC.

```
 1 module inresRTexampleModule {
 2   import from evaluationLibrary all;
 3
 4   testcase inresRTexample() runs on InresSystemType {
 5     var float sendTime1:=−1.0;
 6     var integer iterator1:=0;
 7     var default otherwiseFailDefault:=activate(otherwiseFail);
 8     ConnectionEstablishment();
 9     for ( iterator1 :=0; iterator1 <100; iterator1:=iterator1+1) {
10       if (sendTime1==−1.0) {
11         sendTime1:=self.now+0.01;
12       }
13       else {
14         resume(sendTime1);
15         sendTime1:=sendTime1+0.01;
16       }
17       log(TimestampType:{self.now,”IDATreq1”});
18       ISAP.send(IDATreq:{data});
19       MSAP.receive(MDATind:{DT,number,data});
20       log(TimestampType:{self.now,”MDATind2”});
21       MSAP.send(MDATreq:{AK,number});
22     }
23     ConnectionRelease();
24     deactivate(otherwiseFailDefault);
25     setverdict(pass);
26     stop;
27   }
28
29   control {
30     var testrun myTestrun;
31     var logfile myLog;
32     var verdicttype myVerdict;
33     myTestrun:=execute(inresRTexample());
34     myVerdict:=myTestrun.getverdict;
35     if (myVerdict==pass) {
36       myLog:=myTestrun.getlog;
37       myVerdict:=evalMultipleDelaysOffline(”IDATreq1”,”MDATind2”,
38                                            0.0, incl ,0.005, excl ,myLog);
39       myTestrun.setverdict(myVerdict);
40     }
41   }
42 } // End of module inresRTexampleModule
```

Figure 4.6: *Timed*TTCN-3 Test Case Generated from Figure 4.3

MSC **loop** inline expressions to TTCN-3 **for** statements (Line 9). For the latter, optional guarding conditions containing boolean expressions may be used as termination criteria of a loop. HMSCs can be used to structure and concatenate test cases.

For the calculation of the control flow of TTCN-3 test cases, the *Autolink* approach [SEG+98, GKSH99, Koc01, Sch03] is used as generation algorithm.

The algorithm computes a so called *path* from the partially ordered set of MSC events. Such a path specifies a set of traces which includes no nondeterminism due to non-receiving events, but considers all interleavings due to receiving events. This path representation takes the port queue semantics of TTCN-3 into consideration. A path can be visualised in form of a tree, where branching is related to alternative receiving events. The TTCN-3 generation algorithm computes a TTCN-3 test case that allows to test all sequences of events described by the corresponding path.

TTCN-3 default activation and deactivation are automatically added at the top and bottom of a generated test case (lines 7 and 24 of Figure 4.6). The altstep otherwiseFail is also automatically generated, it just sets the **fail** verdict for any unexpected event (lines 3–12 of Figure 4.7). The **pass** verdict is automatically assigned at the end of the test case just before the **stop** statement (lines 25 and 26 of Figure 4.6). Inconclusive behaviour has to be added manually by adding it either as an altstep or to the test purpose MSC itself using a local setting condition to assign the **inconc** verdict. However, the latter solution is not in the spirit of a test purpose, anymore.

4.2.3 Transformation of Real-Time Concepts

In Section 4.2.1, it has been explained that time constrained stimuli have to be distinguished from time constrained observation. Thus, for transforming MSC time constraints into *Timed*TTCN-3 real-time test cases, the different usages of time constructs in MSC test purposes have to be treated separately.

Time Constrained Observations

For real-time test case generation, time constrained observations contained in test purpose MSCs are translated to *Timed*TTCN-3 by creating time stamps for the observed events. For offline evaluation, the time stamps are stored in the log file produced by the test case. In the online evaluation approach, time stamps are stored in ordinary variables and compared during the test run.

In the following, only offline evaluation is discussed. Online evaluation is intended to be used in tests that react based on the observed real-time properties. Hence, to make reasonable use of online evaluation, it would be necessary to specify how to deal with the outcome of the online evaluation. Indeed, a test purpose should abstract from that. Apart from that, all considerations can be generalised to online evaluation, because both approaches are based on the generation of time stamps. Furthermore, only relative time constraints are considered, since they are more relevant than absolute ones.

The example test purpose in Figure 4.3 contains a latency real-time requirement, i.e. a time constrained observation. Since the constraint is attached to the messages IDATreq and MDATind, the *Timed*TTCN-3 statements for

```
 1 module evaluationLibrary {
 2
 3   altstep otherwiseFail () {
 4     [ ] any port.receive {
 5       setverdict(fail)
 6       stop;
 7     }
 8     [ ] any timer.timeout {
 9       setverdict(fail)
10       stop;
11     }
12   }
13
14   type record TimestampType {
15     float logtime,
16     charstring id
17   }
18
19   type enumerated IntervalBoundaryType { excl, incl };
20
21   function evalLatencyOffline(charstring messageIdA, charstring messageIdB,
22                     float lowerbound, IntervalBoundaryType lowerboundarytype,
23                     float upperbound, IntervalBoundaryType upperboundarytype,
24                     logfile timelog) return verdicttype {
25     // ...
26   }
27 } // End of module evaluationLibrary
```

Figure 4.7: Library with Time Stamp and Evaluation Function Definitions

generating time stamps (lines 17 and 20 of Figure 4.6) are placed directly before and after the associated communication operations (lines 18 and 19).

The generated time stamps contain the value of the local clock and a label of type **charstring**. The value of the local clock may be obtained by the **self.now** statement. The label is used to identify time stamps afterwards.[8] The value of the label is generated from the message name used in the MSC plus a consecutive number to distinguish between different occurrences of the same message. The type definition for the TimestampType is provided by the predefined *Timed*TTCN-3 module evaluationLibrary (Figure 4.7) in lines 14–17. The necessary **import** statement is added automatically at the top of the *Timed*TTCN-3 module containing the generated real-time test case (Line 2 of Figure 4.6).

The module control part is automatically created in addition to the real-time test case (lines 29–41 of Figure 4.6). The structure is identical to the

[8]In Chapter 3, enumerations were used for that purpose. The particular implementation of time stamp labels is irrelevant, as long as they are distinguishable. However, to avoid a cluttered enumeration type, **charstring**s are used in this approach.

one presented in Chapter 3. A call to an offline evaluation function is added for each time constrained observation defined in the MSC test purpose. In the example in Figure 4.6, the evaluation function evalMultipleDelaysOffline is called in lines 37 and 38. The actual parameters are generated from the interval of the real-time constraint in the test purpose MSC and from the same label identifiers which were used inside the related **log** statements.

The called evaluation functions are provided together with the time stamp type definition in *Timed*TTCN-3 libraries. For example, the predefined evaluation function evalMultipleDelaysOffline is used to retrieve and evaluate a sequence of matching pairs of time stamps from a given log file. The two **charstring** parameters of the function identify the time stamps to be compared. The next four parameters define the time interval between two time stamps. Upper and lower bound of the interval are defined by two **float** values. The values incl and excl define whether a boundary is closed or open, i.e. whether the evaluation function uses $\leq$ or $<$ for comparisons. The definition of incl and excl as elements of an enumeration type is shown in Line 19 of Figure 4.7. The last parameter of the evaluation function refers to the log file which is subject of evaluation.

Only a stub of the evalMultipleDelaysOffline evaluation function is depicted in lines 21–26 of Figure 4.7. The actual implementation is very similar to evalLatencyOffline provided on Page 66 in Figure 3.12, except for the fact that evalMultipleDelaysOffline does not evaluate a fixed number of time stamps. Instead, as many matching time stamps as possible are retrieved.[9]

Placement of Time Stamping Statements

In an ideal world, no time passes between a send or receive event and the corresponding time stamp generation, and hence, the time stored in a time stamp is the actual time of an event. For abstract test specification, this assumption might be valid. However, since executable test cases are implemented on real hardware, some time passes between both statements. Thus, there is a choice of putting a **self.now** operation before or after a time constrained event derived from a test purpose. For **receive** operations, the **self.now** operation has to be put after the **receive**, because a **receive** operation is blocking. (In Figure 4.6, this is shown in lines 19–20, where the **self.now** operation is contained in a **log** statement.) But for a **send** operation, there is the option to put a **self.now** operation before or after.

In the Inres example, the first event of the latency time constraint relates to sending the message IDATreq. Thus, the **log** statement associated to the **send** operation may be inserted before or after the **send** operation in

[9]The reason is that the problem of predicting from the test purpose MSC the number of time stamps which are generated during a test run is undecidable.

Line 18 of Figure 4.6. In the first case, the observed duration would be slightly longer than in the second case. Therefore, the choice of placement should depend on whether the time constraint is used to prescribe a *minimal duration* or a *maximal duration*.

If the time constraint has only an upper bound (or the lower bound is zero as, e.g., [0.0,0.005) in Figure 4.3), a maximal duration is specified, i.e. the SUT shall not exceed the upper bound. In this case, choosing the placement which yields the shorter observed duration might result in a **pass** verdict even though the actual duration was longer and slightly violated the real-time constraint. Therefore, the **self.now** operation shall be placed just before the **send** operation as shown in lines 17–18 of Figure 4.6. In this case, a slightly larger duration is observed and one can be sure that if even this larger duration meets the real-time requirement, the actual duration fulfils the real-time requirement in any case.

The opposite considerations hold for testing minimal durations, e.g. intervals like [1ms,), for requiring that an SUT shall not respond too early. In this case, one is on the safe side if a slightly shorter observed duration still meets the real-time requirement, because the actual occurred duration will be slightly larger. Thus, the **self.now** operation shall be placed just after the first event.

In the combined case[10], i.e. neither the lower interval bound is omitted or zero, nor the upper bound is omitted (e.g. [8ms,10ms]), it is a matter of taste which bound to give the priority. If one assumes that encoding for sending and decoding for receiving takes a similar amount of time, the **self.now** operation shall be placed after the **send** operation because it has also to be placed after the **receive** operation. This may lead to a measurement, which is closer to reality since the extra delay introduced to both operations by the test runtime system will eliminate each other.

Time Constrained Stimuli

A time constrained stimulus in an MSC test purpose description is translated into a *Timed*TTCN-3 **resume** statement which is used to schedule the execution of the related **send** operation. A generic *Timed*TTCN-3 skeleton for a time constrained stimulus is shown in Figure 4.8. If the time constraint consists of a single point in time, like [d], this value can be used as relative offset to the **self.now** expression as in Line 3 of Figure 4.8.

If the time constraint is an interval of the form $[t_1, t_2]$, any of the values inside the interval is possible as delay of the **send** operation. This may lead to an infinite number of test cases, which is infeasible in practise. By using test data selection heuristics from functional data testing, like boundary values

[10]Specifying just a single element as interval (e.g. [5ms]) for a time constrained *observation* is not recommended, because it is very unlikely that exactly that value is matched.

```
float sendTime;
myport.receive; / myport.send;
sendTime:=self.now+d;
 ...
resume(sendTime);
myport.send;
```

Figure 4.8: *Timed*TTCN-3 Skeleton for Timed Stimulus

[Mye79, Bei95], an appropriate number of test cases can be selected, e.g. d:=t_1 for testing the extreme lower and d:=t_2 for testing the extreme upper allowed point in time. (In the implemented real-time test case generation tool, just the lower boundary is selected, because it is more likely that an SUT fails to fulfil a real-time requirement if it is stimulated in a fast manner.)

Time Constraints Attached to Inline Expressions and References

For MSC time annotations involving MSC inline expressions and MSC references, several special cases have to be considered. For example, cyclic time constraints in MSC loops using the MSC extension presented in Section 2.3.3 can be treated almost like ordinary relative time constraints: In case of a *time constrained cyclic observation* (Figure 4.9), time stamps are not generated for a pair of two communication events, but for a sequence of a single communication event. Hence, a call to a different predefined evaluation function, working on sequences of a single time stamp only, is necessary. For a *time constrained cyclic stimulus* (e.g. sending message IDATreq every 10ms as in the Inres test purpose example in Figure 4.3), a different set of statements than presented in Figure 4.8 is required: the first execution of the **send** operation has to be performed immediately, while all subsequent executions have to adhere to the cyclic time constraint. This is achieved by the *Timed*TTCN-3 statements given in lines 5 and 10–16 of Figure 4.6.

For MSC inline expressions and references, MSC allows to impose time annotations to the top or bottom of the respective frame. In these cases, the

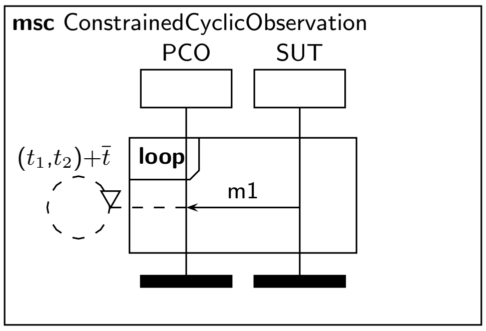
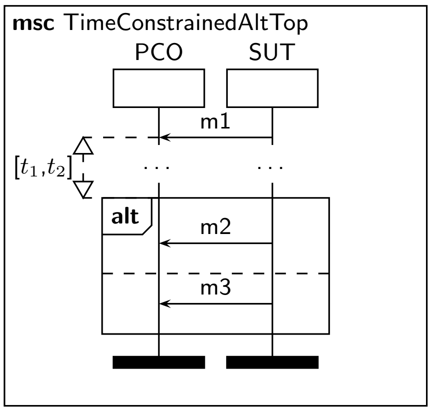


Figure 4.9: Time Constrained Cyclic Observation



(a) Test Purpose

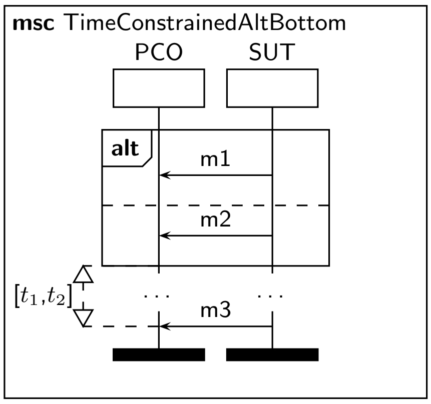
```
 1 port.receive(m1);
 2 log(TimestampType:{self.now, ”m1”});
 3 ...
 4 alt {
 5   [ ] port.receive(m2) {
 6      log(TimestampType:
 7           {self.now, ”Either_m2_or_m3”});
 8   }
 9   [ ] port.receive(m3) {
10      log(TimestampType:
11           {self.now, ”Either_m2_or_m3”});
12   }
13 }
```

(b) Derived Test Case

Figure 4.10: Time Constraint Attached to First Event of Alternative

time annotation refers to the first or last event which actually occurs inside the inline expression or reference, respectively. Thus, for time constrained observation, a time stamp has to be generated for every first or last event which is possible due to alternatives or interleaving. For time constrained stimuli, this is not relevant, because only one path is generated by the tool. Since a magnitude of combinations is possible (e.g. time constraint attached to top/bottom of inline expression/reference, time constrained observation/ time constrained stimulus), just a few examples are given in the following.

In Figure 4.10a, a time constrained observation is attached to the top of an **alt** inline expression. Thus, for every possible first event, a time stamp is created in the generated *Timed*TTCN-3 code (lines 6–7 and 10–11 of Figure 4.10b). In order to be able to evaluate the log file independently from the actual branch, the tool generates the same label (Either_m2_or_m3) in both branches of the alternative.



(a) Test Purpose

```
alt {
  [ ] port.receive(m1) {
  }
  [ ] port.receive(m2) {
  }
}
log(TimestampType:
     {self.now, ”Either_m1_or_m2”});
 ...
port.receive(m3);
log(TimestampType:{self.now, ”m3”});
```

(b) Derived Test Case

Figure 4.11: Time Constraint Attached to Last Event of Alternative

The treatment of time annotations which are attached to the bottom of MSC inline expressions (Figure 4.11a) and references is simpler. In this case, a time stamp is just generated after the *Timed*TTCN-3 **alt** construct as shown in lines 7–8 of Figure 4.11b.

Time annotations attached to the bottom of **loop** inline expressions can be treated in the same way as in Figure 4.11. For annotations attached to the top, it has to be distinguished whether the loop starts with several possible observations (due to partial ordering or an **alt** expression) or with either just one observation or with a stimulus. In the latter cases, a time stamp may just be generated before the actual loop starts. In the first case, the situation is similar to the one described in Figure 4.10. However, the time stamp shall only be generated in the first iteration of the loop. Thus, the first iteration containing the time stamp creation has to be unrolled, while all further iterations may be part of a *Timed*TTCN-3 **for** loop.

4.3 Distributed Test Architectures

In the previous section, test generation for local test architectures was discussed. But, if the SUT is physically distributed, the test system usually has to be distributed, too. Since *Timed*TTCN-3 supports distributed testing as well, basically the same transformation rules are also applicable for generating distributed real-time test cases. However, when testing real-time requirements imposed on events which occur at different test components, a more sophisticated test generation is required.

4.3.1 Generation of Distributed Functional Test Cases

The generation of pure functional test cases for distributed test architectures from test purpose MSCs has been well studied in [GKSH99, Koc01, Sch03]. The therein described *Autolink* tool is able to generate TTCN-2 test cases for distributed test architectures.

According to [GKSH99], basically two additional kinds of information have to be provided to generate distributed test cases from a test purpose MSC:

Test configuration: The test architecture for which the distributed test cases shall be generated has to be known: How many *Parallel Test Components* (PTCs) are used in addition to the *Main Test Component* and how are the *Points of Control and Observation* (PCOs) assigned to the test components?

Synchronisation of Distributed Test Behaviour: Distributed test components need to synchronise their behaviour with each other to achieve their common goal. For generating distributed test behaviour, a tool needs to know at which points of the control flow synchronisation is desired. This has also an influence on the test configuration, because for exchanging coordination messages, test components require further ports in addition to their PCOs.

With respect to the test configuration, the Autolink tool assumes a test architecture where each PCO is assigned to an individual PTC. For each non-SUT instance axis of a test purpose MSC, a PTC is created. The MTC is just responsible for creating the PTCs and for coordinating their activities. Therefore, each PTC involved in synchronisation has an additional port for exchanging coordination messages with the MTC and the MTC has as many port for the coordination with its PTCs. A corresponding distributed test architecture for two PCOs is depicted in Figure 4.12. (Test cases may as well be automatically generated for other distributed test architectures. Just the routing of coordination messages differs in these cases.) The test behaviour for each PTC can be obtained by simply slicing a test purpose MSC vertically and applying for each non-SUT instance separately the transformation rules presented in Section 4.2.2.

For the specification of test behaviour synchronisation, the Autolink tool supports to add shared MSC conditions to the non-SUT instances of a test purpose MSC. In this case, coordination messages are automatically generated for synchronising all PTCs which are associated to a PCO covered by an MSC condition. The MTC is responsible for distributing the coordination messages, i.e. to wait for messages from the involved PTCs which indicate that the behaviour preceding the synchronisation condition has finished and to notify subsequently all involved PTCs to proceed. As a result, the described approach allows to take a test purpose MSC as given in Figure 4.13 as input and to generate coordination messages exchanges as shown in lines 14–19 of Figure 4.14 and in Figure 4.17.

The presented Autolink approach is not only applicable to obtain TTCN-2 test cases, but can also be used for generating distributed TTCN-3 test

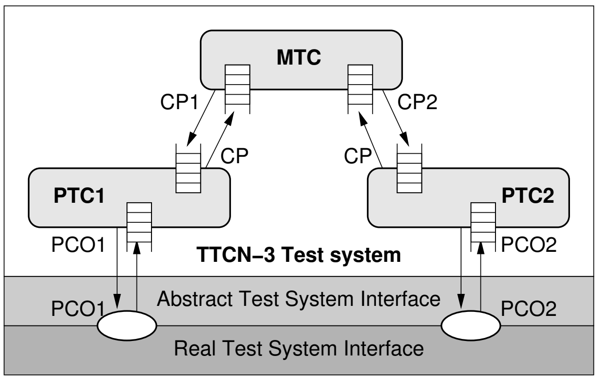


Figure 4.12: Distributed Test Architecture with Two PTCs

cases, because TTCN-3 is semantically downwardly compatible to TTCN-2. Moreover, the enhanced expressiveness of TTCN-3 allows to generate even shorter test cases with respect to the reception of coordination messages by the MTC: Since it is not possible to predict the order in which coordination messages arrive at the MTC, Autolink has to generate TTCN-2 code for all possible permutations of coordination message receptions. In contrast, TTCN-3 supports a shorter specification of permutated message arrival by providing the **interleave** statement. The automatically generated TTCN-3 code for the MTC's coordination behaviour can take advantage of this (lines 14–17 of Figure 4.14).

4.3.2 Generation of Distributed Real-Time Test Cases

Based on the Autolink approach, also distributed real-time test cases can be generated from real-time test purpose MSCs. As long as each real-time requirement can be treated by a PTC on its own, i.e. time annotations in the test purpose MSC are always local to non-SUT instances, the transformation rules described in Section 4.2.3 still apply. But, if test cases shall be generated which involve real-time properties spanning over several PTCs, modified transformation rules are necessary.

Time Constrained Observations

If a time constrained observation spans over several test components, it is only reasonable to compare the time stamps which are generated by distributed test components as long as their local clocks are synchronised. This can be achieved in *Timed*TTCN-3 by creating the concerned PTCs in the same timezone. Offline evaluation can then easily be applied to compare the logged time stamps which refer to clocks from within the same timezone.

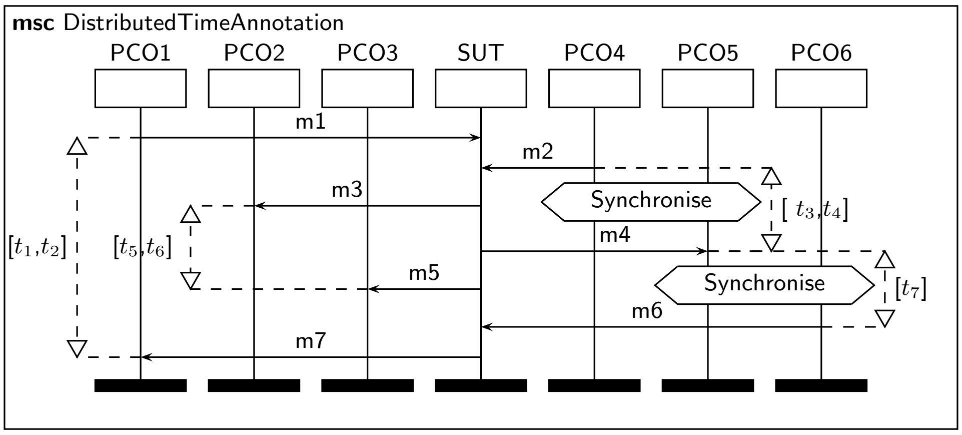


Figure 4.13: Test Purpose with Distributed Time Annotations

```
 1  type enumerated timezones {
 2    tz_PCO2_PCO3, tz_PCO4_PCO5_PCO6
 3  }
 4
 5  testcase DistributedTimeAnnotation() runs on AllCPType system SystemType {
 6    var ptc_PCO1Type ptc_PCO1:=ptc_PCO1Type.create;
 7    var ptc_PCO2Type ptc_PCO2:=ptc_PCO2Type.create(tz_PCO2_PCO3);
 8    var ptc_PCO3Type ptc_PCO3:=ptc_PCO3Type.create(tz_PCO2_PCO3);
 9    var ptc_PCO4Type ptc_PCO4:=ptc_PCO4Type.create(tz_PCO4_PCO5_PCO6);
10    var ptc_PCO5Type ptc_PCO5:=ptc_PCO5Type.create(tz_PCO4_PCO5_PCO6);
11    var ptc_PCO6Type ptc_PCO6:=ptc_PCO6Type.create(tz_PCO4_PCO5_PCO6);
12    var float timeValue:=0.0;
13    // ...
14    interleave {
15      [ ] CP4.receive(float:0.0) {}
16      [ ] CP5.receive(float:0.0) {}
17    }
18    CP4.send(float:0.0);
19    CP5.send(float:0.0);
20    CP5.receive(float:?) −> value timeValue;
21    CP6.send(float:timeValue);
22  }
```

Figure 4.14: Generated Timezone Enumeration and MTC Behaviour

The simplest solution would be to assume that just all PTCs are created in the same timezone. Indeed, this assumption might be too strict. Instead it is sufficient to partition the PTCs into minimal sized sets taking the transitive connection due to distributed time annotations into account.

This shall be illustrated for the test purpose given in Figure 4.13: PCO1 is only involved in the assessment of local time stamps and hence, the responsible PTC needs no clock synchronisation at all. PCO2 and PCO3 are involved in the same time constraint, thus, their PTCs need to be clock synchronised. PCO4 and PCO5 share a time constraint as well as PCO5 and PCO6. Thus, all these three PCOs need to be created in the same timezone.

Such timezone sets can be calculated using simple graph algorithms [CLR90]. The resulting *Timed*TTCN-3 **timezones** enumeration type might look like in lines 1–3 of Figure 4.14. The MTC is responsible for creating the PTCs in the correct timezones as shown in lines 6–11.[11] Further behaviour of the MTC, like connecting its coordination ports with the PTCs, mapping their PCOs to the test system interface and starting the PTCs, is not shown.

Once the individual PTCs are clock synchronised, their behaviour for creating time stamps can be generated by the same transformation rules as described in Section 4.2.3. Two examples for receiving messages m3 and m5 and the corresponding time stamp creation are shown in lines 29–37 of Fig-

[11] Since the MTC itself is not involved in creating time stamps, the MTC may be created in the module control part without any clock synchronisation parameter.

```
23  type record TimestampType {
24    float logtime,
25    timezones componentzone,
26    charstring id
27  }
28
29  function ptc_PCO2Behaviour() runs on ptc_PCO2Type {
30    PCO2.receive(m3);
31    log(TimestampType:{self.now, self.timezone,”m1”});
32  }
33
34  function ptc_PCO3Behaviour() runs on ptc_PCO3Type {
35    PCO3.receive(m5);
36    log(TimestampType:{self.now, self.timezone,”m5”});
37  }
```

Figure 4.15: Generated Time Stamp Type and PTC Behaviour

ure 4.15. The only difference to test generation for local test architectures is that the timezone is additionally part of the TimestampType (lines 23–27), and hence, gets also logged (lines 31 and 36).

Figure 4.16 depicts the module control part which has to be generated for the test purpose in Figure 4.13. It contains calls to the offline evaluation functions related to the time constrained observations described by the test purpose. In Line 70, the evaluation function evalMultipleDelaysOffline is called with **none** as timezone parameter, since the time stamps m1 and m7

```
62  control {
63    var testrun myTestrun;
64    var logfile myLog;
65    var verdicttype myVerdict;
66    myTestrun:=execute(DistributedTimeAnnotation());
67    myVerdict:=myTestrun.getverdict;
68    myLog:=myTestrun.getlog;
69    if (myVerdict==pass) {
70      myVerdict:=evalMultipleDelaysOffline(”m1”,”m7”,none,t1,incl,t2,incl,myLog);
71      myTestrun.setverdict(myVerdict);
72      myVerdict:=evalMultipleDelaysOffline(”m2”,”m4”,tz_PCO4_PCO5_PCO6,
73                                           t3, incl ,t4, incl ,myLog);
74      myTestrun.setverdict(myVerdict);
75      myVerdict:=evalMultipleDelaysOffline(”m3”,”m5”,tz_PCO2_PCO3,
76                                           t5, incl ,t6, incl ,myLog);
77      myTestrun.setverdict(myVerdict);
78    }
79  }
```

Figure 4.16: Generated Module Control Part

```
38  type port CoordinationPoint message {
39    inout float;
40  }
41
42  function ptc_PCO4Behaviour() runs on ptc_PCO4Type {
43    PCO4.send(m2);
44    log(TimestampType:{self.now, self.timezone,"m2"});
45    CP.send(float:0.0);
46    CP.receive(float:0.0);
47  }
48
49  function ptc_PCO5Behaviour() runs on ptc_PCO5Type {
50    CP.send(float:0.0);
51    CP.receive(float:0.0);
52    PCO5.receive(m4);
53    log(TimestampType:{self.now, self.timezone,"m4"});
54    CP.send(float:self.now);
55  }
```

Figure 4.17: Generated Coordination Messages and PTC Behaviour

were created by a PTC which is not clock-synchronised. However, the other calls to this evaluation functions are performed with the respective timezone actual parameter in which the corresponding time stamps were created (lines 72–73 and 75–76).

The implementation of the evalMultipleDelaysOffline evaluation function is not shown. It is similar to the previously presented offline evaluation functions. Though, it has to be assured that the evaluation function is able to cope with the possible orderings of time stamps. For example, the time stamps for the observation of m3 and m5 at PCO PCO2 and respectively PCO3 may occur either in the order m3, m5, or m5, m3 due to interleaving of partially ordered reception events.[12]

For restricting the possible orderings of events, the functional synchronisation mechanisms described in Section 4.3.1 may be used in real-time test purpose MSCs. However, even if such synchronisation conditions are used in the context of time annotations, they restrict actually the functional behaviour of sending and receiving, i.e. the path which is generated by the Autolink approach. Since the generation of a time stamping statement is based on that path, synchronisation conditions need not to be especially considered for real-time test generation.

In the example given in Figure 4.13, a condition has been added to synchronise the behaviour of the PTCs responsible for PCO4 and PCO5. Thus, the

[12]In fact, this has already to be considered when generating test cases for local test architectures. Though, the MSC test purpose example in Figure 4.3 did not allow any reordering. Hence, this problem did not arise.

SUT has to be stimulated by message m2 prior to the observation of message m4.[13] According to the presented synchronisation approach, coordination messages are exchanged between the PTCs and the MTC (lines 45–46 and 50–51 of Figure 4.17). Messages of type **float** are used for synchronisation. The definition of the corresponding CoordinationPoint port type is given in lines 38–40. The MTC waits for the PTCs to reach the synchronisation point and signals subsequently the PTCs to proceed (lines 14–19 of Figure 4.14).

Time Constrained Stimuli

A test purpose MSC may as well contain time constrained stimuli which span over two PTCs. (For example, PCO5 and PCO6 in Figure 4.13.[14]) Like for time constrained observations, the local clocks of the involved PTCs need to be synchronised. Thus, the same considerations on timezones and PTC creation also apply for time constrained stimuli.

A stimulus determined by a relative time constraint must be scheduled by the associated PTC using the *Timed*TTCN-3 **resume** statement. The schedule is based on the point in time when the first event of the relative time constraint occurred. Thus, the necessary time information has to be transferred from the PTC which is responsible for the first event to the stimulating PTC. This can be achieved by using messages exchanged via the existing CoordinationPoint port type. These messages are able to carry a *Timed*TTCN-3 **float** value as payload. For the assumed distributed test architecture, this means that such time information is transferred from one PTC to another via the MTC. The test system must be fast enough to be able to deliver the coordination message to the stimulating PTC in time.

Figure 4.17 shows in Line 54 the sending of a coordination message with the time stamp of the first event to the MTC. The MTC just forwards the received value via port CP6 to the PTC associated to PCO6 (lines 20 and 21 of Figure 4.14). The behaviour of that PTC is depicted in Figure 4.18. In Line 58, the coordination message carrying the previously generated time stamp for the first event is received. Then, in Line 59, the PTC resumes until firstEventTime+t7 is reached. After that, the actual stimulus is sent.

The described approach of transferring time stamps during a test run via coordination messages can also be exploited for generating real-time test

[13]Actually, the path generated using the Autolink approach would posses this property even without the synchronisation condition, since send events are prioritised to minimise the number of interleavings.

[14]In this example, a synchronisation condition is used to assure that this part of the test purpose describes actually a time constrained stimulus and not alternatively a time constrained observation. In case the synchronisation condition is absent, the path generated by the Autolink approach would prioritise the sending of m6 against the reception of m4. As a result, the time constrained stimulus would be turned into a time constrained observation.

```
function ptc_PCO6Behaviour() runs on ptc_PCO6Type {
  var float firstEventTime:=0.0;
  CP.receive(float:?) −> value firstEventTime;
  resume(firstEventTime+t7);
  PCO6.send(m6);
}
```

Figure 4.18: Generated PTC Behaviour for Timed Constrained Stimulus

cases which make use of online evaluation. But, as already discussed in this chapter, online evaluation does not fit very well to the abstraction of test purposes.

4.4 Summary

This chapter presented the generation of *Timed*TTCN-3 real-time test cases from test purpose descriptions which are formalised using MSC. This allows to derive test architecture specific test behaviour from functional and real-time requirements which are specified from the perspective of the SUT. The test development process may be accelerated by automating this test development activity. To achieve this, a tool was implemented that allows to generate *Timed*TTCN-3 real-time test cases for a local test architecture. This tool is based on an existing tool for the generation of functional TTCN-3 test cases from test purpose MSCs.

The underlying transformation rules for obtaining *Timed*TTCN-3 real-time test cases from time annotations in test purpose MSCs have been explained. As an example, they have been applied to a real-time test purpose for testing real-time properties of an Inres protocol implementation. Furthermore, the generation of real-time test cases for distributed test architectures has been discussed. In this context, it has been shown how to deal with real-time properties which span over several, distributed test components. By transferring time stamps during a test run, it is even possible to react online on time stamps gathered at remote test components.

Related Work

Using MSC for test purpose specification and the consecutive test case generation as suggested in this chapter is not new and has already been implemented in several academical and industrial tools like *Autolink*, *Testcomposer*, or *ptk*. Autolink and ptk support the generation of test cases for concurrent test architectures.

Autolink [SEG+98] and Testcomposer [KJG99] support the generation of TTCN-2 test cases either from SDL specifications guided by MSC test purposes or directly from MSC test purposes. However, both tools are only able to generate tests for assessing pure functional requirements.

The ptk tool [BBJ$^+$02] also generates TTCN-2 test cases from MSC test purposes. The latest version is intended to produce TTCN-3 output as well. Moreover, ptk is able to generate test cases which aim at testing real-time requirements by taking non-standard time annotations in test purpose MSCs into account. Using non-standard annotations is a disadvantage in comparison to the approach presented in this chapter. A further deficiency is that the generated test cases use just standard TTCN timers to assess the real-time behaviour of the SUT. This leads to all the problems which have already been discussed in Chapter 3. In contrast to the approach presented in this chapter, it is in particular not possible to test real-time requirements which span over several distributed test component.

Limitations

Generating test cases from test purpose MSCs has some limitations. For example, in the example test purpose for the Inres protocol in Figure 4.3, it is not perceptible that the sequence number changes for each iteration of the loop as well as the sequence number used for acknowledgement. As a result, a tool is not able to generate from such a test purpose MSC a test case which reflects the alterations of the sequence number and the dependencies between the received sequence number and the acknowledged sequence number. In principle, MSC allows to express this, but this would clutter the test purpose MSC. Furthermore, inconclusive cases cannot be generated from a test purpose which describes by definition just a scenario leading to a pass verdict.

Thus, it is desirable to let a test case developer benefit from the knowledge and experience which are contained in the transformation rules used by the described real-time test generation tool. An appropriate pattern-based approach for deriving *Timed*TTCN-3 test cases from test purpose MSCs is presented as part of the next chapter.

Chapter 5

Test Patterns

In this chapter, patterns which can be used for development of tests are presented. Since testing is usually performed against some specification, some of those patterns may not only be useful in the test development phase, but also in the requirements and specification development phase of a system. Using patterns already in the initial phase of system development allows an integrated methodology, where both, test case and implementation development benefit from patterns.

While in the previous chapter an automated test case generation approach was presented, patterns may be used both in a manual and in an automated test generation approach. In a manual development approach, a developer can reuse patterns and needs not to invent the wheel twice. An automated approach may benefit from patterns, because tools may identify patterns automatically and provide further support for the identified pattern. In fact, the transformation rules presented in the previous chapter are already a sort of pattern. This becomes apparent, where solutions in form of *Timed*TTCN-3 skeletons are assigned to problems in form of MSC test purpose snippets. Thus, some aspects of automatic test case generation, e.g. providing the correct evaluation function for a certain real-time requirement in a test purpose, may profit from patterns.

The roots of this work have been published by the author as part of a joint work in [NDG04]. This work was refined and extended during the participation of the author in the *European Telecommunications Standards Institute* (ETSI) Work Item "Patterns in Test Development" (PTD) [ETS04].

This chapter is structured as follows: First, in Section 5.1, some foundations on patterns and their usage are given. This includes a proposal for classifying test patterns and a discussion of test pattern templates. A specialised test pattern template is presented later-on in Section 5.2. In the latter section, *Real-time Communication patterns*, which have been identified by the author, are presented. These patterns ease the specification of real-time requirements. Then, in Section 5.3 it is shown, how these patterns relate to *Timed*TTCN-3 and aid real-time test generation. Finally, a summary of this chapter and some conclusions are given.

5.1 Foundations

Patterns and a *pattern language* were first introduced in the context of architecture for building houses and towns based on the combination of known, proven solutions [AIS$^+$77, Ale79]:

> "*The elements of this language are entities called patterns. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*"
> (C. Alexander et al. in [AIS$^+$77])

While these patterns where not developed with having software development in mind, it turned out that the notion of patterns is valuable for software engineering. [GHJV95] made object-oriented software *design patterns* popular. A more fundamental discussion of software patterns, which is not only restricted to design patterns and also discusses, e.g., the development of new patterns (*pattern mining*), is given in [BMR$^+$96].

In general, patterns are regarded as elements of reusability. A pattern language[1] describes, how these elements can be combined. For developing software, a well known primary source of reusable elements are libraries. But libraries of predefined elements are often not flexible enough, because they lack customizability. In contrast, patterns provide an abstract solution for a generic problem. This abstraction keeps patterns customisable. On the other hand, patterns need to be instantiated before being usable, which requires some more experience and effort than simply reusing a predefined library. A third kind of reusable elements are frameworks, which may be regarded as the opposite of a library, i.e. an application-dependent infrastructure architecture which can be customised by plugging in missing parts. Patterns —at least design patterns— may be considered as a sort of micro framework.

Patterns can be regarded as rules of how to solve a certain problem. A pattern consists of several parts which are described using a fixed template. For the original patterns, C. Alexander defines in [Ale79]:

> "*Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.*"

[1][BMR$^+$96] suggests to use the term *pattern system* instead of the notion of a "language" since it is not possible to construct software solely from patterns as the term "language" might wrongly suggest.

The number of parts of a pattern and the template used for describing a pattern may differ with respect to the domain, for which a pattern applies. However, a pattern usually consists —besides its name— of at least a description of a problem, it's context and the provided solution. From now on, only patterns of relevance for software engineering and especially testing are regarded.

Besides supporting reuse, patterns have an additional valuable advantage: patterns provide a common vocabulary for problem solutions. For software engineering, this benefits not only system design, implementation and test development but also system maintenance, since it is easier to identify known patterns than to understand an uncommon system structure. Both aspects of patterns, reuse and common vocabulary, allow a more efficient construction of systems.

5.1.1 Patterns and Testing

Since the existing patterns, like software design patterns, have proven to reduce development time and to improve maintainability [PUT$^+$01], it is desirable to identify patterns for the testing domain, too. Hence, *test patterns* are considered in this thesis, in particular *Real-time Communication patterns* as introduced in Section 5.2.

The intention is to use test patterns during the test development process to solve recurring problems. Thus, test development time can be reduced and maintenance is eased as described in the preceding section. Furthermore, test patterns improve the comparability of test suites and reproducibility of tests.

Since a system needs to be testable and (at least black-box) testing is performed against some requirements, patterns should be used in an integrated system development methodology. The highest benefit can be achieved by using patterns already in the requirements capture phase. This improves testability and allows utilisation of related test patterns during test development. This will be shown in sections 5.2 and 5.3. Moreover, such an integrated development approach allows even an automated generation and implementation of test cases based on patterns which can be identified by a tool.

5.1.2 Relationship to Other Kinds of Patterns

Before classifying different kinds of test patterns in the following sections, test patterns need to be distinguished from other existing kinds of patterns. A valuable source of patterns is [Hil04]. Most of the patterns mentioned below can be found there. Note that the chosen patterns provide just a small selection. They were chosen either because of their popularity or their relevance for distributed systems and testing.

Design patterns are described in [GHJV95, BMR$^+$96]. They focus on structural aspects of object-oriented software design by identifying common, recurring relationships between classes and their interactions.

Design pattern are described using a template. It consists of the following parts: name, problem, context, forces, solution, resulting context, rationale, related patterns, known uses.

Despite of this template, the pattern description is rather informal. Prose language is used, except for the solutions section where additionally *Unified Modeling Language* (UML) diagrams [OMG03a, OMG03b] are used to describe the proposed classes and their relationship.

Design patterns are abstract in the sense that they are independent of the object-oriented language which is used to actually implement the pattern. Nevertheless, to illustrate the implementation, usually an example of the instantiated pattern is given by means of code excerpts from some object-oriented programming language.

Hence, a design pattern provides just a small framework, which has to be instantiated by coding the contained classes using an object-oriented implementation language, and customising it by inserting the missing parts. In contrast to real object-oriented frameworks, design patterns are smaller and application domain independent.

SDL patterns [Gep01] are tailored to the development of SDL systems, i.e. systems which are specified using the *Specification and Description Language* [ITU99a]. SDL patterns benefit from the formal SDL semantics, which offers the possibility of precisely specifying how to apply a specific pattern, under which assumptions this will be allowed, and what properties result for the embedding context. This includes special symbols, which are introduced for the SDL fragments of the pattern template to distinguish the patterns itself from its context.

The SDL pattern template contains the following parts: name, version, intent, motivation, structure, message scenario, SDL fragment, semantic properties, refinement, cooperative usage, known uses.

Since these patterns are intended for SDL, they are language specific. Moreover, they are specific to a particular domain, namely communication systems. Nevertheless, they abstract from a specific protocol.

Telecommunications Distributed Processing Patterns [DeB95] refer to different stages of the development of a telecommunication system, mainly on analysis and design.

The used pattern template consists of the parts name, context, forces, problem, solution, resulting context, design rationale, author.

The presented patterns are heterogeneous and vary not only with respect to the development stages to which they apply but also in the degree of abstractness and formalisms used: sometimes just prose language is used, but sometimes also informal graphics or (pseudo) code snippets are provided.

Similar to patterns, further kinds of reusable elements of good practise have been published in the past. An example are *idioms*, i.e. language specific code snippets as a solution for recurring problems in a particular language. In software development, the notion of idioms has been used first in [Cop92], where they relate to the *C++* [ISO98] programming language.

Idioms and the patterns presented so far have in common that they can be customised because they have to be instantiated. But in contrast to patterns, idioms do not provide an abstract solution, but a detailed language specific implementation. Nevertheless, idioms are application domain independent. [BMR+96] classifies idioms as a special category of patterns, namely a low-level language-dependent pattern. In this thesis, this classification of idioms as patterns is adopted.

5.1.3 Existing Test Patterns

Besides these patterns which relate to software analysis, design and specification, some patterns which are associated to testing have already been identified and published:

Test design patterns are described as a part of [Bin99]. Another suitable term would be "object-oriented test strategy patterns", since those patterns describe several strategies to derive test cases for testing object-oriented software.

The template used for test design patterns consists of the following parts: name, intent, context, fault model, strategy (test model, test procedure, oracle, automation), entry criteria, exit criteria, consequences, known uses, related patterns.

Test design patterns are abstract in the sense that they describe just an approach of how to obtain test data or test actions. They are not related to how to specify the obtained test data or actions.

Unit Test Patterns are intended for the design and implementation of unit tests, e.g. for the *JUnit* test framework [GMB04].

Several separate unit test patterns (e.g. mock objects [MFC01]), are collected on [Obj04]. They do not constitute a pattern system and no template is used for describing them. Some of them use code snippets for explanation.

Furthermore, in [Cli04], several unit test patterns are presented. They do not make use of a template either, even though all of them share a common format by using an informal diagram and prose text. However, they are categorised into *pass/fail patterns*, *collection management patterns*, *data driven patterns*, *performance patterns*, *process patterns*, *simulation patterns*, *multithreading patterns*, *stress test patterns*. These patterns are abstract in the sense that they are not related to a certain unit test framework. Indeed, they are that abstract and general that some of them apply even to other kinds of testing, like system tests.

In general, the unit test patterns in [Obj04, Cli04] are not as elaborated as other kinds of patterns.

Code review patterns are collected in [Wik04]. These kind of patterns describes patterns for organising reviews of source code which is a kind of static tests.

The template for code review patterns is not very rigorously used. In the most elaborated case, it consists of the following parts: name, problem, context/forces, solution, resulting context, rationale, related patterns.

Code review patterns do not apply to formal artifacts like source code, but to management issues, i.e. they are organisational patterns. Thus, the solutions which they provide are abstract.

Besides the above test patterns, some international standards related to testing contain abstract solutions for recurring problems. Thus, the solutions captured in such standards may be regarded as candidates for patterns.

As an example, the different OSI service types defined in [ISO02] are a kind of communication pattern. A further example is the ISO/IEC standard 9646 *Conformance Testing Methodology and Framework* (CTMF) [ISO97b]: The contained abstract test methods which define different test architectures can be seen as test patterns, because they provide abstract solutions for testing protocol entities of communicating systems. Also the concept of *Protocol Implementation eXtra Information for Testing* (PIXIT) can be regarded as a pattern to decouple test suites from hardware details. Though, since these elements were not developed with having patterns in mind, they do, e.g., not use a pattern template.

In addition to international standards, numerous strategies for deriving test cases (or test data) have been published. Prominent examples are the books [Mye79, Bei95]. Especially the black-box test strategies in [Bei95] are already described using a uniform template which is very close to a pattern description. Hence, it would be possible to re-write those strategies as patterns, just in the same way as it is done in [Bin99].

5.1.4 Classification of Test Patterns

In the previous section, three kinds of test patterns and several candidates for test patterns are mentioned. Before introducing a further kind of test patterns in Section 5.2, a classification of all the various patterns which can be regarded as *test patterns* will be developed in this section.

Different areas of test development may benefit from using patterns. Thus, for the classification of test patterns, several dimensions of test development where patterns are applicable can be identified: the *phase of test development*, the *test goal* and the *scope of testing*[2]. These three dimensions are orthogonally to each other. Nevertheless, some dependencies between them might exist. For example, a pattern for a certain test architecture is applicable for conformance testing only, but not for interoperability testing. Furthermore, using a certain pattern during test design might suggest a specific pattern during test evaluation.

Figure 5.1 shows the dimensions suggested for classifying test patterns. The three dimensions and their subdivision are described in the following.

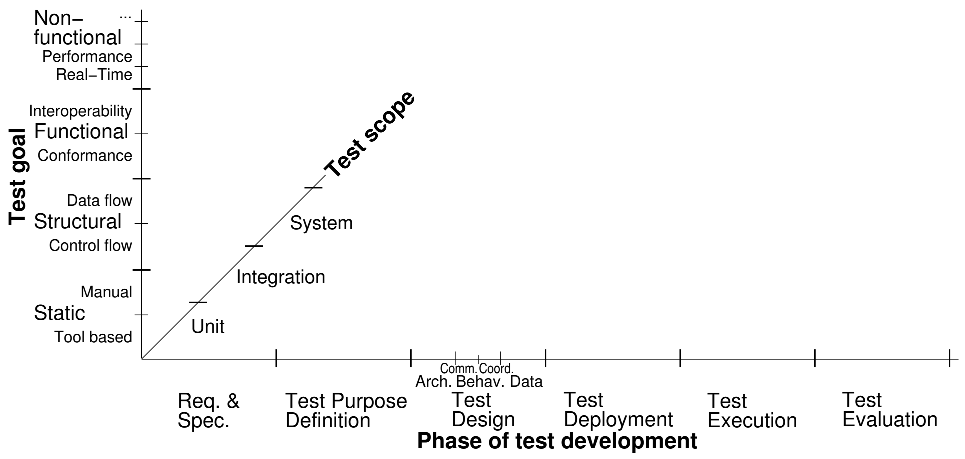


Figure 5.1: Dimensions of Pattern Classification

Phase of test development: The dimension of the *phase of test development* relates to the point in time of the test development process where a test pattern is used. The test development process can be divided into the following phases which are usually consecutive, but may be iterated as part of an incremental or iterative development approach:

[2]The suggested classification is similar to the test classification provided in Section 2.1. Though, the dimension of distribution has been dropped and is replaced by the *test development phase*. Nevertheless, that dimension still exists, but it is often coupled to the test scope: unit tests are usually performed locally, whereas integration and system tests might require a distributed tester. Moreover, local testing may be regarded as a special case of distributed testing.

1. *Testable requirements and specification:* Preceding to the actual test development, testable requirements need to be captured and a testable design needs to be specified. This early stage can already benefit from test patterns. Some of the patterns used in later test phases may be related to patterns used in the requirements, specification or design phases of system development. Thus, using patterns in the early stages of system development may ease test development in later stages.
2. *Test purpose definition:* The stage of identifying and capturing test purposes may also be aided by patterns. When using a formalised test purpose description, e.g. *Message Sequence Chart*s (MSCs), patterns can be used as MSC building blocks. Patterns for test purpose definition may be related to the requirements patterns described in the previous item.
3. *Test design* is the next phase of test development. Since this phase is more complex than, e.g., test purpose definition, it can be further sub-divided according to several aspects:
   (a) *Test architecture:* The architecture of a test system —especially if a distributed test architecture is used— may be complex. Nevertheless, often similar test architectures are used. Thus, this aspect of test design benefits intensively from patterns. Examples for such architectural test patterns are the CTMF test methods which are mentioned in Section 5.1.3.
   Further viewpoints of architectural patterns are: distribution (local or distributed), number of involved ports (PCOs) or respectively test components.
   (b) *Test behaviour:* Besides the architecture, the behaviour of a test is an important part of test design. The development of test behaviour may profit from patterns as well. Such patterns may provide abstract solutions, but also language specific idioms. An example is guarding a test case with a timer to guarantee an eventual termination of the test case—if the solution is only provided as TTCN-3 code, it is an idiom, but the essence can also be described as abstract solution which is valid for any test language which supports timers.
   Besides general behavioural test patterns, further sub-classes of test behaviour patterns can be distinguished:
      i. *Communication:* Test communication patterns provide solutions for the communication between the SUT and test components.
      ii. *Test coordination:* Such patterns apply for distributed testing (which might involve a test architecture provided

by a test architecture pattern). Test coordination patterns provide solutions for coordination procedures between several parallel test components, either by using a designated (e.g. main) test component as coordinator or via a peer-to-peer approach.

(c) *Test data:* A further important aspect of the test design stage is the selection, definition and matching of test data. This may also benefit from patterns. Some of those patterns may be application or protocol dependent (e.g. test data patterns applicable for *Internet Protocol version 6* (IPv6) testing.)

4. *Test deployment:* This kind of patterns relates to making abstract test suites executable. As already described in Section 5.1.3, the concept of PIXIT or the usage of test suite parameters in general are candidates for patterns in the test deployment stage.
5. *Test execution:* Test execution, especially when not fully automated, may also be aided by patterns, e.g. by providing guidance to a manual tester how to perform certain test steps. Nevertheless, TTCN-3 test suites are usually executed automatically. Therefore test execution patterns are not discussed in this thesis.
6. *Test evaluation:* The final phase of the testing process is the evaluation of test results. But, in fact, the individual steps of how to evaluate test results may have already been specified during test design. Therefore, test evaluation patterns are usually tied to some test design patterns. Nevertheless, the actual evaluation is performed after test execution.

A further dimension for the classification of test patterns is the goal of testing:

Test goal: The *test goal* dimensions relates to the type of test. Different test goals may require different test patterns. In contrast to the first dimension, there is no causal order between the particular goals, because the different test strategies are completely different from each other. Nevertheless, e.g., non-functional testing is usually only performed after an item under test has passed the functional tests. The division of this dimension is as follows:

1. *Static testing:* In contrasts to all other test goals, static tests assess an item under test without executing it. Mainly, two different kinds of static testing can be distinguished:

   (a) *Tool based:* Automated tools which assess the source code are used. Those tools perform a semantical analysis of the control or data flow (e.g. the *lint* tool [Dar88]). The flaws which are detected by such tools may be regarded as anti-patterns. Another application of static test tools is to check source code formatting conventions or to calculate source code metrics which give a hint on the code quality.

   (b) *Manually:* Another possibility of static testing is the manual examination of the source code by humans. This is also known as code inspection, review or walkthrough [FW90, GG93, FLS04]. Especially the management of the examination process can be guided by patterns, e.g. those mentioned in Section 5.1.3.

2. *Structural testing:* Structural test approaches have the goal to cover the structure of the item under test during test execution with respect to a certain criteria. In order to create test cases which achieve this, the internal structure of the item under test needs to be known. Therefore, structural tests are usual glass-box tests. Structural testing can be sub-divided with respect to the type of coverage at which the test is aimed:

   (a) *Control flow coverage:* The goal of this type of tests is to cover the control flow of the item under test, e.g. branch coverage or path coverage.

   (b) *Data flow coverage:* This type of tests aims at covering accesses to variables, i.e. the data flow.

   Structural test approaches may benefit from patterns. An example is the test design phase: the glass-box test design strategies described in [Mye79] are candidates for glass-box test patterns.

3. *Functional testing:* The test goal of this type of test is to test the item under test with respect to the functionality it should fulfil. Functional testing is based on the specification which prescribes the behaviour of the item under test. In contrast to structural tests, functional tests do not require any knowledge of internals of the item under test. Therefore, they are usually black-box tests. However, if the item under test is instrumented to obtain coverage measurements, functional tests have to be regarded as grey-box tests.

   The following special types of functional tests can be distinguished:

   (a) *Conformance testing:* This type of test is aimed at testing whether an item under test conforms to its specification based on observable external behaviour. A lot of proven solutions for this goal are known in the area of conformance testing (e.g. [ISO97b]) and thus candidates for conformance test patterns.

   (b) *Interoperability testing:* The goal of interoperability testing is to test several implementations against each other and to observe their inter-working. Patterns for this goal may be, e.g., related to test architectures which are suitable for observing the implementations and injecting or modifying messages exchanged between the implementations.

4. *Non-functional testing:* Like functional tests, non-functional test are usually performed against some requirements. Though, non-functional testing aims at the assessment of non-functional requirements. A variety of different test goals can be related to testing of non-functional properties. Hence, a sub-division of test patterns for non-functional tests is appropriate, e.g.:

   - *Real-time tests:* Real-time test patterns provide solutions for testing hard real-time properties of discrete events, e.g. the response time or the latency of forwarding a signal. Patterns of this class are intensively discussed in Section 5.2.
   - *Performance tests:* In contrast to the previous pattern kind, performance test patterns do not deal with discrete events, but properties of whole streams of data, e.g. the quality of a video stream.

   Many more kinds of non-functional tests which may profit from patterns exist. Examples for further non-functional requirements which might be subject of tests are given in Section 2.2.

The final dimension of the suggested classification of test patterns is the scope of testing:

Test scope: The *test scope* describes the granularity of the item under test. During software development, tests can be performed at different levels of scope. Test patterns for one scope may not be suitable for a different scope. Testing are usually performed in the following order of scopes:

1. *Unit:* The scope of unit testing is the smallest testable unit. As mentioned in Section 5.1.3, patterns which ease designing and implementing unit tests have already been identified.
2. *Integration:* Another kind of test patterns may fall into the class of integration test patterns. Patterns for this scope aid developing tests for strongly connected collections of units, i.e. sub-systems.
3. *System:* The whole system is the scope of system tests. System tests are usually black-box tests, thus the large number of abstract solutions for black-box tests falls also into the category of system test patterns.

The characteristics of the three dimensions used for the classification of patterns differs slightly. With respect to the phase of test development, a test pattern relates usually to exactly one phase (but may be tied to another test pattern from another phase). But for the other dimensions, it is possible that a pattern may be used for, e.g., all scopes of testing or for a set of test strategies, e.g. functional and non-functional testing. Some examples will be given in the following.

Assessment of Pattern Classification

To show that the suggested classification is applicable for test patterns, an exemplary classification of some test patterns will be given in the following. As a result, covered and non-covered areas in the test pattern space become visible.

Figures 5.2–5.8 show the test pattern space which is spanned by the three dimensions *phase*, *goal*, *scope* and where the considered test patterns are located according to the suggested classification. To ease cognition which range of each axis is covered by a pattern, the perpendicular projection of a pattern to the coordinate planes is shown as a shaded area. If necessary, the location of the shaded areas with respect to the axes is additionally indicated by dashed lines.

Test design patterns as described in [Bin99] can be classified as patterns for the test design phase. [Bin99] introduces a large variety of patterns, thus the whole domain of test scopes is covered. Since the patterns are intended for object-oriented systems, they make assumptions and require knowledge of the implementation. Hence, with respect to the test goal dimension, these patterns are classified as structural test patterns (cf. Figure 5.2).

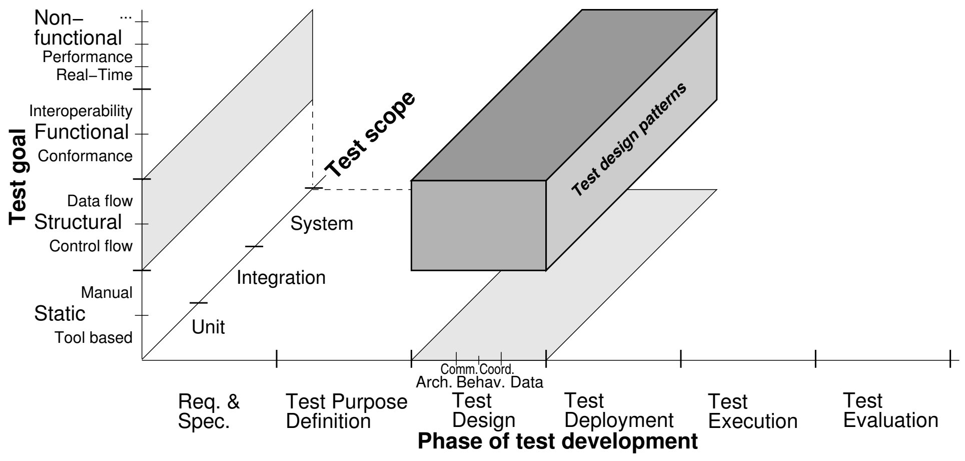


Figure 5.2: Classification of Existing Test Design Patterns

Unit test patterns as surveyed in [Obj04, Cli04] can be classified as test patterns for the test design, test deployment, and test evaluation phase aimed at structural coverage for testing at unit scope (cf. Figure 5.3).

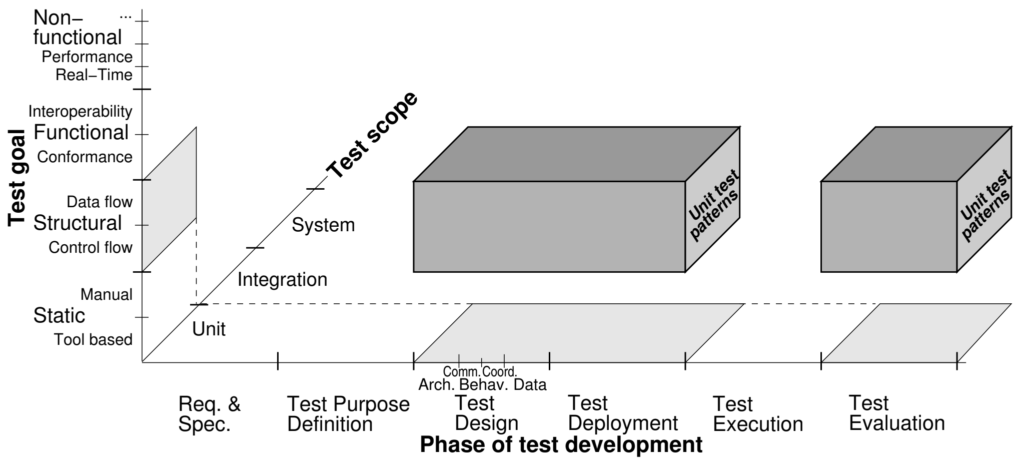


Figure 5.3: Classification of Existing Unit Test Patterns

Code review patterns as collected in [Wik04] can be classified as test patterns for manual static tests. In principle, manual static tests may be performed at any scope level. However, the feasible level depends on the detailedness of the item under test: A coarse design model can be assessed at system level, while the source code of an implementation can be assessed reasonably only on unit or integration test level. Since the patterns in [Wik04] are intended for source code, unit and integration test scope apply for their classification. When classifying static

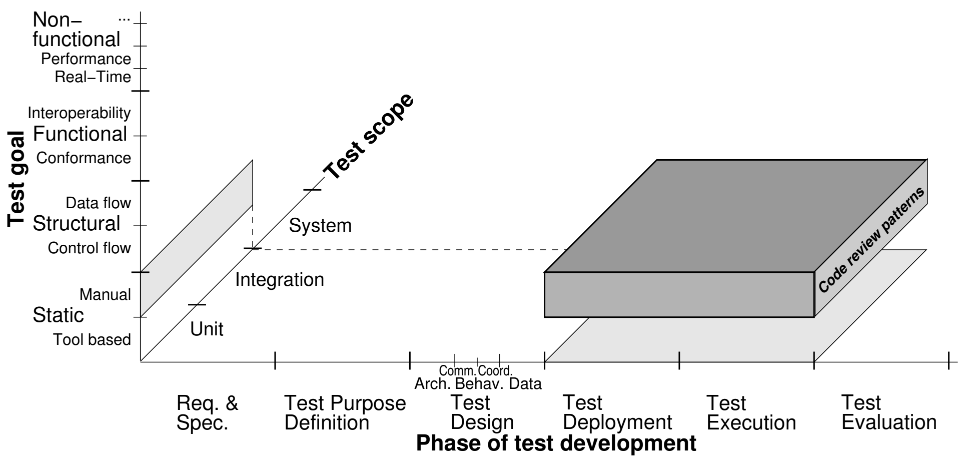


Figure 5.4: Classification of Code Review Test Patterns

test patterns with respect to the dimension of test development phase, it has to be taken into account that static tests are quite different from dynamic tests. Nevertheless, for each of the phases of dynamic testing a counterpart in static tests exists. The code review patterns support setting up a review team and performing the actual review. This maps to the test deployment and test execution phases (cf. Figure 5.4). However, in a more general classification, these patterns may also be regarded as test management patterns.

In addition to the existing test patterns, some test pattern candidates are mentioned either in Section 5.1.3 or 5.1.4. They shall also be classified in the following.

Test methods as standardised in the *Conformance Testing Methodology and Framework* [ISO97b] can be classified as patterns suitable for the test design phase, especially the test architecture sub-class. Since they are described in the context of conformance testing, they have to be regarded as functional conformance test patterns with respect to the test goal axis. The test scope is mainly system level, though, since CTMF distinguishes between *System Under Test* and *Implementation Under Test*, the actual implementation may be also regarded as a sub-system, i.e. integration tests may also apply (cf. Figure 5.5).

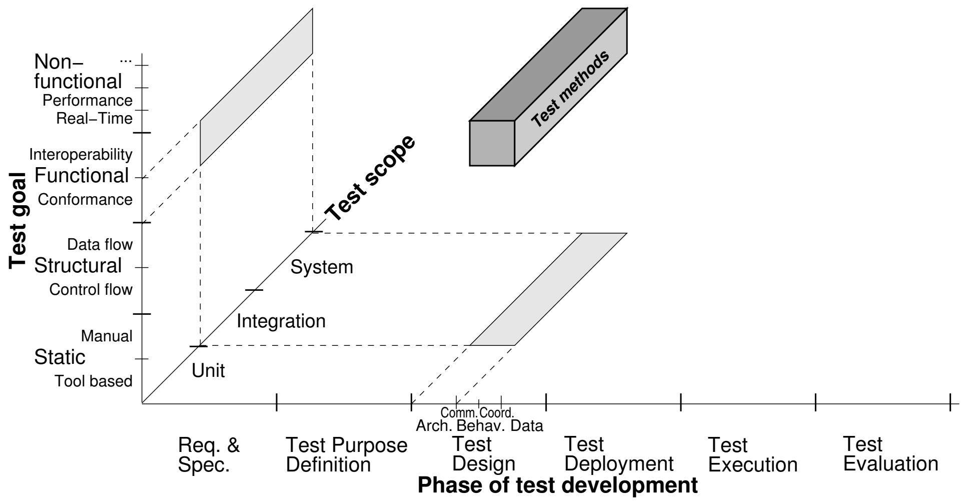


Figure 5.5: Classification of the Test Methods Pattern Candidate

Protocol implementation extra information for testing (PIXIT) are also part of CTMF [ISO97b]. Thus, concerning the test goal and the test scope axis, the same considerations as for test methods hold. However, they ease test deployment. Therefore, they are located on the according location of the test phase axis (cf. Figure 5.6).

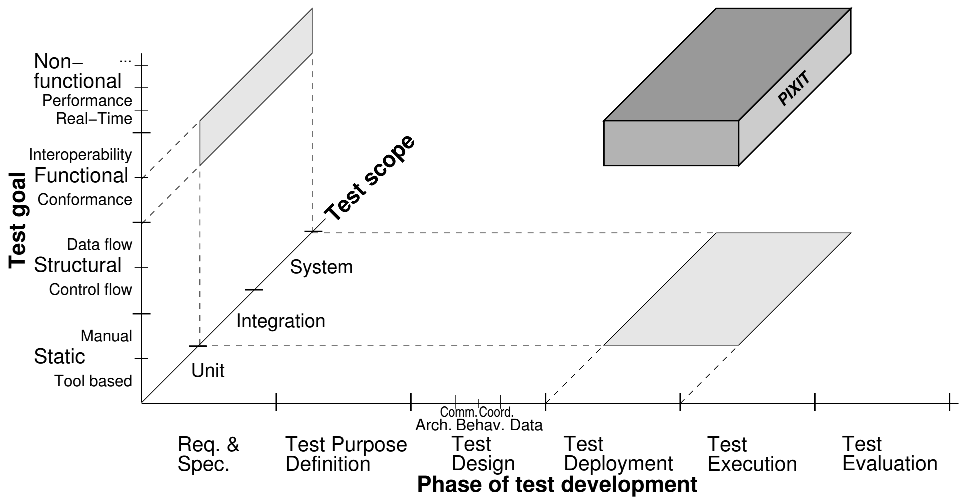


Figure 5.6: Classification of the PIXIT Pattern Candidate

Most idioms, like, e.g., test case guarding timer are patterns for the test design phase, especially test behaviour. Idioms can be useful for all test scopes. They are intended for specifying dynamic tests and thus apply in general for all test goals except static tests (cf. Figure 5.7). Nevertheless, it is possible that a certain idiom relates to exactly one test goal or scope. In this case, the covered volume in the pattern space would reduce.

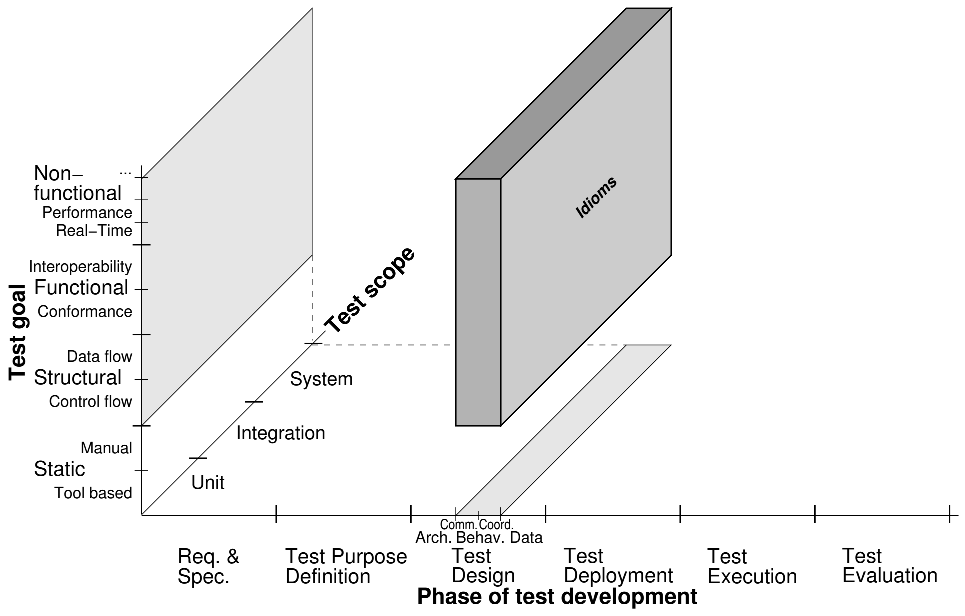


Figure 5.7: Classification of most Test Idioms Pattern Candidates

Test coordination patterns are patterns for the test design stage, especially test behaviour, or to be more precise: the test coordination sub-class. Such patterns may be suitable for a range of test goals, except static tests. Moreover, e.g., real-time tests may require other (time aware) test coordination patterns than non-time critical functional tests. Since units have usually just one interface and thus require just one test component, test coordination patterns are mainly related to system or integration scope (cf. Figure 5.8).

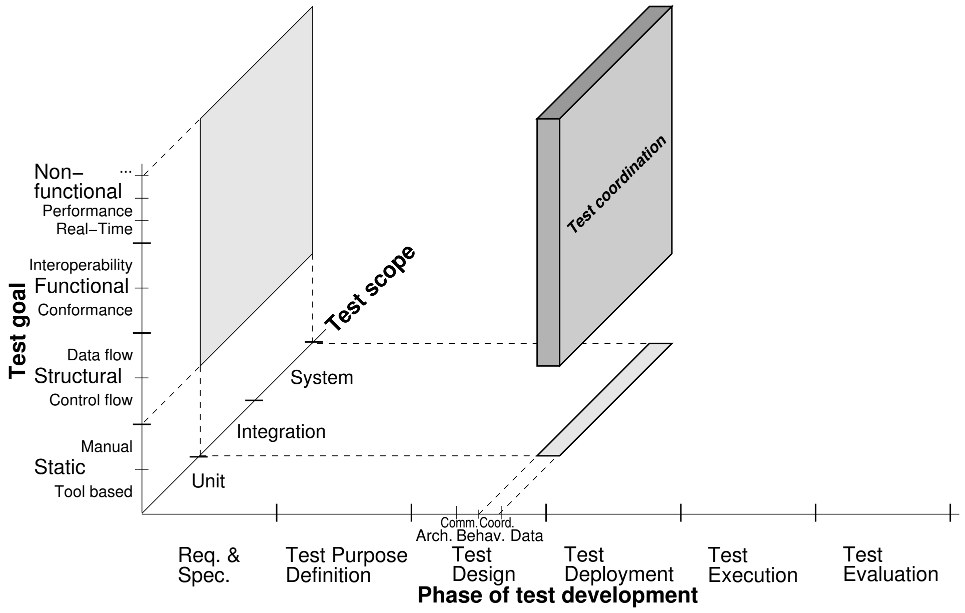


Figure 5.8: Classification of the Test Coordination Pattern Candidates

Adding up the areas which are covered by the discussed patterns and pattern candidates, results in the conclusion that for just a fraction of the test pattern space, patterns have been identified, yet. In order to narrow the existing gaps, a further type of test patterns, which covers the real-time layer of the test goal dimension, is introduced in Section 5.2.

5.1.5 Pattern Templates

Experience has shown that for the notation of patterns, a uniform template is suitable. This eases browsing through a pattern catalogue. Furthermore, a fixed template makes it is easier to compare patterns to each other, especially if several patterns apply to the same context. Thus, it is desirable to use a special template for describing test patterns.

However, when comparing the various pattern templates presented in sections 5.1.2 and 5.1.3, it becomes obvious that one template may not fit all

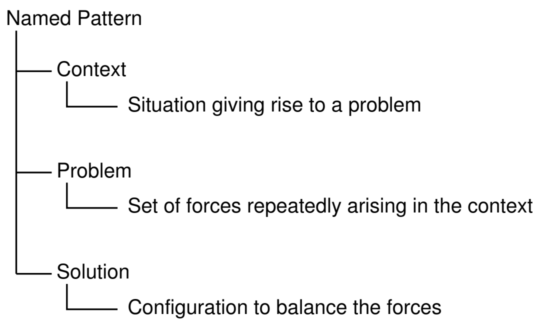

Named Pattern
Context
Situation giving rise to a problem
Problem
Set of forces repeatedly arising in the context
Solution
Configuration to balance the forces

Figure 5.9: Pattern Template Scheme (Slightly modified from [BMR+96])

kinds of patterns. Even when restricting to test patterns, a look at the template for the test design patterns surveyed in Section 5.1.3 shows that a very specialised template is used. While that pattern template is reasonable for that particular purpose, it would be very impractical to describe other kinds of test patterns with that template—especially when keeping the huge test pattern space developed in Section 5.1.4 in mind. (The heterogeneity of the pattern space —and thus of the required pattern templates— stems mainly from the dimension of the phase of test development, because the individual stages differ significantly.)

Nevertheless, it is possible to consider the common essence of pattern templates. Based on this, specialised variants of pattern templates can be derived for the different kinds of test patterns.

In [BMR+96], the core of the common pattern templates is distilled to look like in Figure 5.9. Accordingly, the essence of a pattern template can be summarised as follows:

- As previously mentioned, the pattern should have a *name*, which communicates the core of the pattern. An example is: "Coordinate parallel test components".

- The *context* describes the situation in which the problem solved by the pattern occurs. It is important to be aware that the context itself is not already the problem, but may lead to the problem. The context might also refer to another pattern, from which the context might result. An example is: "Distributed test architecture with more than two test components."

- The *problem* describes a problem which occurs repeatedly in the given context. An example is: "Parallel test components shall synchronise their behaviour." Furthermore, so called *forces* are usually listed in the problem description. Such forces refer to any additional requirements or constraints which have to be challenged when solving that problem.

Such forces may even be contradictory. Examples for forces are: "The number of coordination messages shall be minimal to reduce bottlenecks." or "The existing generic distributed test architecture shall not be modified."

- The *solution* provides the abstract solution, which is able to balance the forces associated to the problem. An example is: "If all existing test components have a connection to the MTC, just use the MTC as a centralised coordinator. If bottlenecks shall be reduced, introduce additional coordination connections only between the components which are to be synchronised."

It makes sense to adhere to this scheme for all of the different kinds of test patterns, because everyone who once used patterns in other domains will immediately feel familiar inside a test pattern catalogue, too. A concrete example of a template for a specialised kind of test pattern will be given in Section 5.2.1.

5.1.6 Usage of Patterns

After a test pattern has been identified (through "pattern mining") it can be written down using a pattern template. Eventually, this will result in a pattern catalogue, which can be sorted with respect to the test pattern classification proposed in Section 5.1.4. This enables a test developer to select and instantiate a pattern from the test pattern catalogue. For finding and instantiating the right pattern, the following steps (taken and modified from [BMR$^+$96]) can be performed:

1. Specify the problem you want to solve. If the general problem has several aspects, divide them into sub-problems. Identify the forces which constrain your problem.
2. Select a pattern category from the pattern classification which applies to your problem.
3. Search the pattern catalogue in the category selected in Step 2.
4. Compare the problem descriptions of the found patterns. Which one matches best your problem and balances your forces best?
5. (a) If the selected pattern provides several variants as solution, select the one which implements best the solution for your problem.
   (b) If you could not find a suitable pattern in steps 3 or 4, go back to Step 2 and select another, probably more general, pattern category. If even that does not yield a satisfactory result, abort

searching for provided patterns. Instead, solve your problem without using patterns or try to mine a new pattern on your own.

6. Instantiate the selected pattern. The actual way of instantiation depends on the pattern itself. Most of the test patterns will probably either be idioms or provide an example using TTCN-3. Thus, implementing a pattern using TTCN-3 should be straightforward.

After this general considerations on test patterns, this thesis will focus in the remaining sections of this chapter on test patterns which are suitable for testing real-time requirements. It will also be explained, how system and test development can be integrated by associating patterns of different development stages to each other.

5.2 Real-Time Communication Patterns

In the previous section, test patterns and their general foundations have been discussed. In this section, test patterns will be discussed in the context of real-time test specification and implementation using *TIMED*TTCN-3.

Even though it is possible to generate *TIMED*TTCN-3 automatically from graphical test purposes as demonstrated in Chapter 4, pattern support for *TIMED*TTCN-3 is beneficial. The use of *TIMED*TTCN-3 can be facilitated and harmonised by providing a common set of test evaluation functions. This would make test results more comparable and avoids misinterpretations due to the use of different or erroneous evaluation functions. Thus, the key issue is the identification of commonly applicable evaluation functions for *TIMED*TTCN-3 test cases. As described in Chapter 3, such functions are used to evaluate relations among time stamps of events, which are observed during a test run. An evaluation function is related to the number of interfaces of the *System Under Test* (SUT), the number of time stamps to be considered and the number of relations among these time stamps. It would be necessary to provide an infinite set of evaluation functions to cover all cases. This is not possible and, therefore, a pattern-based approach is used to identify evaluation functions for the most common cases.

To achieve this, *Real-time Communication pattern*s (RTC-patterns) for expressing real-time requirements are introduced. RTC-patterns are used to describe real-time requirements in form of time relations among communication operations at the interfaces of a communication system. For each of these patterns, evaluation functions can then be provided. By using RTC-patterns during test design or by scanning test specifications for RTC-patterns, it is possible to use predefined evaluation functions in *TIMED*TTCN-3 test descriptions.

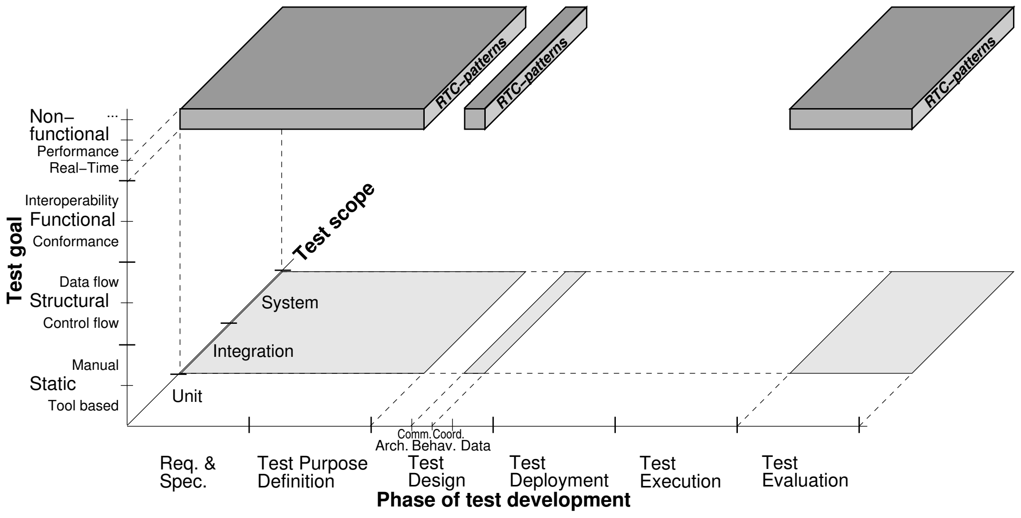


Figure 5.10: Classification of the RTC-patterns

With respect to the pattern classification in Section 5.1.4, RTC-patterns are patterns for real-time tests. Most of the patterns are appropriate for testing at more than one interface, thus the test scope is mainly at integration and system level. Concerning the test development phases, several phases are covered as shown in Figure 5.10: testable requirements and thus also test purpose definition are supported as well as test evaluation. Test behaviour is also affected, as far as the generation of time stamps for communication events is concerned. In comparison to the pattern spaces which are depicted in Section 5.1.4, it becomes visible that RTC-patterns cover completely new domains of the pattern space.

Even though the domain of testing motivated the work on RTC-patterns, these patterns may be of general interest for real-time system development, because RTC-patterns describe requirements, namely real-time requirements. Relating predefined test evaluation functions is just a special application of these patterns. Therefore, RTC-patterns are presented in the remainder of this section independent of the testing domain. Afterwards, their application to testing is explained by providing appropriate test evaluation functions. In general, RTC-patterns abstract from a certain test specification language, but as an example implementation, Section 5.3 shows how they can be instantiated using *Timed*TTCN-3.

In the following, *Message Sequence Chart*s (MSCs) are used to present RTC-patterns for the most common hard real-time requirements [ATM99a, IET90, IET91, IET02]. Since real-time requirements are always related to some functional behaviour on which they are imposed, it is not possible to provide patterns for pure real-time requirements. Therefore, the RTC-patterns contain not only real-time constraints, but also communication events on which the real-time requirements are imposed.

In order to ease specification and testing of distributed real-time systems, it is the intention to provide patterns for testable real-time requirements only. In general, testable requirements can be obtained if the involved events of the system can be observed and stimulated. Thus, it is assumed that the system for which the requirements are specified has appropriate interfaces, i.e. *Points of Control and Observation* (PCOs).

The RTC-patterns are presented as system level MSCs, i.e. each PCO is represented as one MSC instance. The given MSCs make intensively use of the abstraction mechanisms explained in Section 2.3, e.g., the system is described by a single decomposed instance with name System. The internal structure of the system is abstracted by omitting in the System instance header the actual reference to an MSC that decomposes the system behaviour. This way, a black-box view of the system is obtained.

The most common real-time requirements are related to *delay*, *throughput*, *periodic events* and *jitter* respectively. Basically, those requirements describe time relations between one send and one receive event, or the repeated occurrence of a send and a receive event. Depending on the number of PCOs of a system, the RTC-pattern for a certain requirement may look different, i.e. several pattern variants may exist for describing the same real-time requirement in different system configurations. In this thesis, RTC-patterns for systems with one or two PCOs only are provided. They will be listed in the pattern catalogue starting in Section 5.2.2. Based on them, patterns for more PCOs, e.g. for multi-cast, may be derived.

Note that each pattern is described in a self-contained manner. Since some remarks apply for several patterns, they are listed several times. Redundancy in the pattern catalogue is hence by intention.

5.2.1 Real-Time Communication Pattern Template

Before the actual RTC-patterns are described in Section 5.2.2, a template, which is used to describe these patterns, is presented. It is derived from a test pattern template which has been developed by the author as part of the ETSI PTD Work Item [ETS04].

The RTC-pattern template resembles the scheme discussed in Section 5.1.5. For each element of the template, a short description is given. The context of all RTC-patterns is “real-time requirements at integration and system scope” as shown in Figure 5.10. This information is not repeated each time in the context part of the RTC-pattern template. (However, for a more general pattern template, this classification of the current pattern might be a valuable constituent of the context part.)

To ease browsing the pattern catalogue, the pattern template is headed with two horizontal lines surrounding the pattern name. Furthermore, each pattern is started on a seperate page.

Name: The name of the pattern. One or just a few words. If several names for the pattern exist, they shall also be listed here.

Intent: A short (one or two sentences) summary of the pattern, i.e. a description of the problem solved by the pattern, but possibly also the underlying principle of the solution.

Context: The context in which this pattern is applicable, e.g. any conditions which have to be fulfilled before the pattern can be applied. For RTC-patterns, the context is mainly described by an MSC which specifies the involved instances and the required message flow, i.e. the functional behaviour in which an RTC-pattern applies. To provide an abstract description of the context, MSC constructs like decomposed instances and MSC references are used.

Problem: A description and a short discussion of the problem which is actually solved by the pattern.

Roles/Parameters: An enumeration of the different roles of participants involved in the pattern. These participants and further values may be parameters of the provided solution (and also already of the context).

Solution: The detailed description of the abstract solution, i.e. the general idea of the solution as well as any guidelines of how to implement the pattern. For RTC-patterns, the most important element of this part is an MSC which is provided to show where time constraints, i.e. the actual real-time requirements have to be applied. For providing an abstract solution, the same MSC abstraction mechanisms as already used for the context description are applied. Note that test evaluation functions, which may be related to an RTC-pattern, are not provided in this item of the pattern template. They are discussed in a later section. However, in a pure *Timed*TTCN-3 test pattern catalogue, it might be reasonable to provide them as part of the solution item.

Related patterns: Relationship of this pattern to other patterns: Either other patterns which may apply in the same context and provide a different solution but as well patterns which might apply afterwards due to the instantiation of the current pattern.

5.2.2 Pattern Catalogue

The RTC-pattern catalogue consists of patterns for describing three different kinds of real-time requirements: Requirements on delays, throughput, and periodic events.

Patterns for Delays

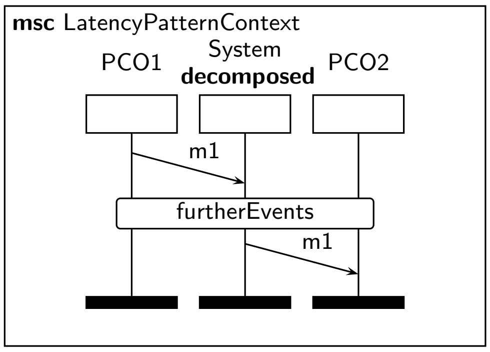
The term *delay* is often used as an umbrella term for both *latency* and *response time* [IET90], since both differ only in the number of PCOs which are involved in the requirement. Hence, patterns for both types of real-time requirements are given in this section.

First, the *Latency* pattern is presented, afterwards two different response time patterns: *Response Time* for a response time requirement on a system and *Response Time PCO* for a response time requirement on a system's environment.

Latency

Intent: Impose a latency real-time requirement on a system forwarding a message from one interface to another.

Context: A system forwards a message from one interface to another.



Problem: A latency real-time requirement shall be imposed on a system forwarding a message from one interface to another. Latency describes the delay which is introduced during the transmission of a signal by a component (the system), which is responsible for forwarding this signal [IET91].

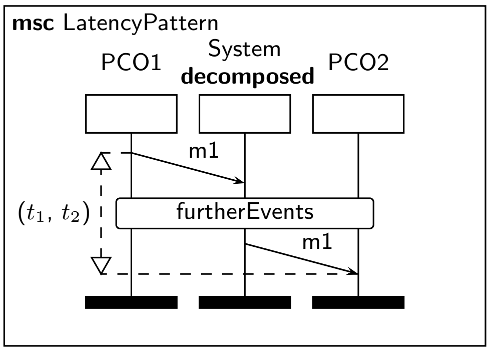
The allowed latency between sending message m1 via PCO1 and receiving it at PCO2 shall be between t_1 and t_2 time units. The delay may be introduced by some further events that may include communication with the system environment (indicated by the MSC reference furtherEvents), the transmission times for message m1, and additional computations inside the system (indicated by the **decomposed** keyword in the heading of the System instance).

Note that even though the same message name is used in this pattern for both transmissions, the actual contents of the forwarded message may differ due to changes introduced by the system, e.g. updated hop counters or processing of the actual payload.

Roles/Parameters:

PCO1:	stimulating interface
PCO2:	observing interface
m1:	stimulus which is forwarded
t_1, t_2:	lower and upper bound for latency real-time constraint

Solution: Add a relative time constraint (t_1, t_2) to the two events of sending message m1 at PCO1 and receiving message m1 at PCO2.

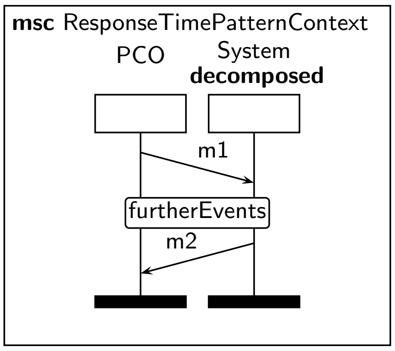


Related patterns: *Throughput Two PCO* usually refers to a *Latency* pattern. Other delay patterns which, however, involve just one interface are *Response Time* and *Response Time PCO*.

Response Time

Intent: Impose a response time requirement on a system processing some data.

Context: A system answers a request on the same interface.



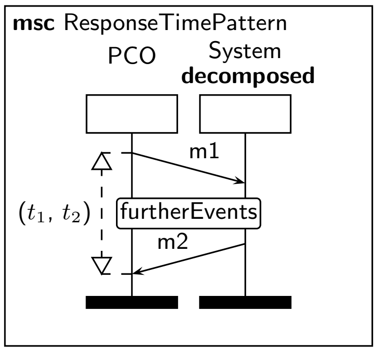
Problem: A response time real-time requirement shall be imposed on a system answering a request.

Response time is a delay requirement where the same PCO is used for sending a message and receiving the corresponding answer. In contrast to the *Latency* pattern, the messages constrained in the response time pattern usually differ significantly, e.g. request (message m1) and response (message m2) in a client-server system. The response time shall be t_1 and t_2 time units between sending message m1 and receiving message m2. The delay is usually introduced due to the behaviour referred to by the MSC reference furtherEvents.

Roles/Parameters:

PCO:	stimulating & observing interface
m1:	request
m2:	answer
t_1, t_2:	lower and upper bound for response time real-time constraint

Solution: Add a relative time constraint (t_1, t_2) to the two events of sending message m1 and receiving message m2 at PCO.

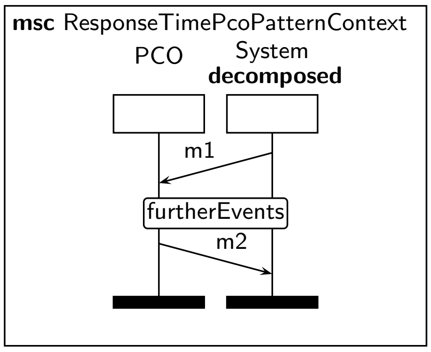


Related patterns: A variant of this pattern is *Response Time PCO*. Another delay pattern is *Latency*. The *Throughput One PCO* pattern may refer to this pattern.

Response Time PCO

Intent: Impose a response time requirement on the environment of a system.

Context: A system requests an answer from the environment.



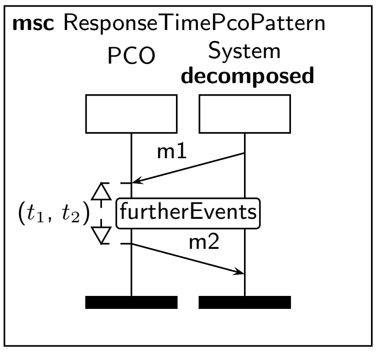
Problem: A response time real-time requirement shall be imposed on the environment of a system.

Response time is a delay requirement where the same interface is used for sending a message and receiving the corresponding answer. In contrast to the *Response Time* pattern, this pattern describes a requirement or assumption for the system environment or tester. This is necessary if a timely behaviour of the environment is needed by the system to fulfil some other requirements. The response time of the system's environment shall be t_1 and t_2 time units between receiving message m1 and sending message m2. The delay is usually introduced by the environment due to the behaviour referred to by the MSC reference furtherEvents.

Roles/Parameters:

PCO:	observing & replying interface
m1:	observed request from system
m2:	reply from environment
t_1, t_2:	lower and upper bound for response time real-time constraint

Solution: Add a relative time constraint $(t_1,\ t_2)$ to the two events of receiving message m1 and sending message m2 at PCO.



Related patterns: A variant of this pattern is *Response Time.* Another delay pattern is *Latency.* The *Throughput One PCO* pattern may refer to this pattern.

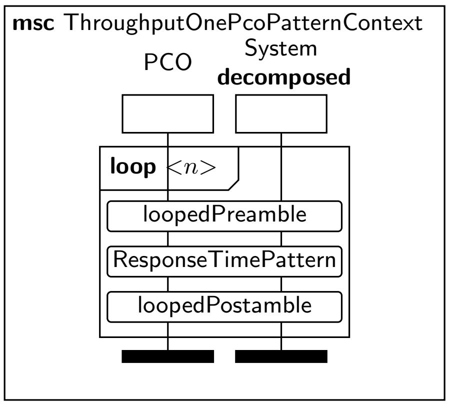
Throughput Patterns

In this section, two different patterns are provided: *Throughput One PCO* and *Throughput Two PCO*. Both patterns involve repeated behaviour, which can be expressed in MSC by using loop inline expressions. Thus, the patterns of this section make extensively use of the MSC loop construct.

Throughput One PCO

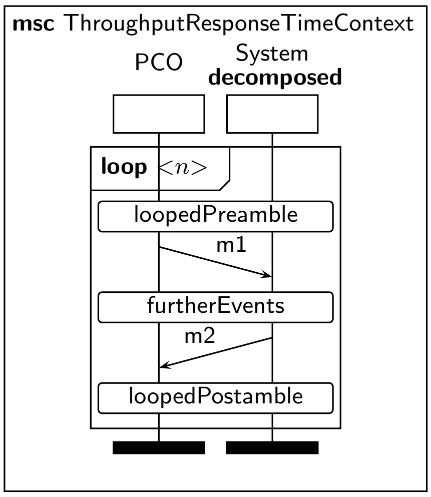
Intent: Impose a throughput real-time requirement on messages exchanged via one interface of a system.

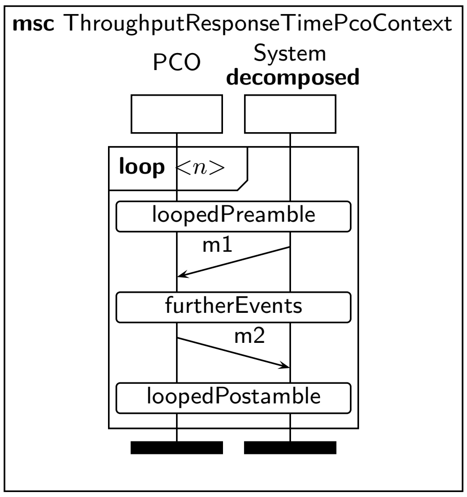
Context: A system exchanges repeatedly the same set of messages with the environment via one interface. The repetition can be subdivided into a preamble, into a message exchange based on either the *Response Time* or *Response Time PCO* pattern, and into a postamble.



The loop inline expression includes the references loopedPreamble, ResponseTimePattern, and loopedPostamble. Thus, additional behaviour, which precedes or follows the response pattern, may be contained in the MSC references loopedPreamble and loopedPostamble.

ResponseTimePattern refers to RTC-patterns *Response Time* or *Response Time PCO*. Though, the contained response time patterns defines usually just the functional behaviour, which is part of the throughput requirement. Thus, possible expansions of the ResponseTimePattern reference yield one of the following MSCs:





Problem: A throughput real-time requirement shall be imposed on messages repeatedly exchanged via one interface of a system.

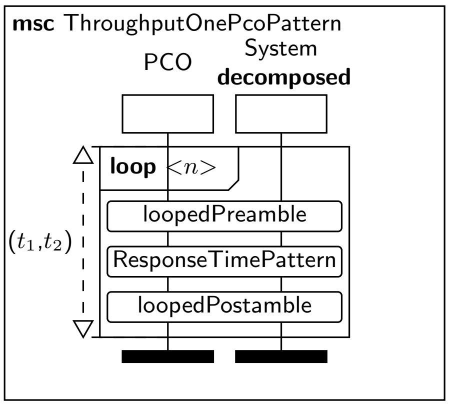
In contrast to periodic or delay-based real-time requirements, *throughput* requirements consider the system performance over a longer duration, not just for a single set of events. This means, *throughput* constrains the number of messages per time that a system has to deliver or to process repeatedly [IET90].

Roles/Parameters:

PCO:	observing & stimulating interface
n:	number of repetitions
t_1, t_2:	lower and upper bound for all n repetitions of the loop for which the throughput real-time requirment must hold

Thus, the actual throughput *TP* imposed by this pattern is within the interval $\left(\frac{n}{t_2}, \frac{n}{t_1}\right)$.

Solution: Add a relative time constraint (t_1, t_2) to the first and last event of the repetitive behaviour, i.e. the top and the bottom of the MSC loop inline expression.



The given throughput pattern constrains a throughput *TP* to be $\frac{n}{t_2} < TP < \frac{n}{t_1}$ events per time unit. Those “events” consist typically of a set of events, in particular such according to one of the response time patterns presented before.

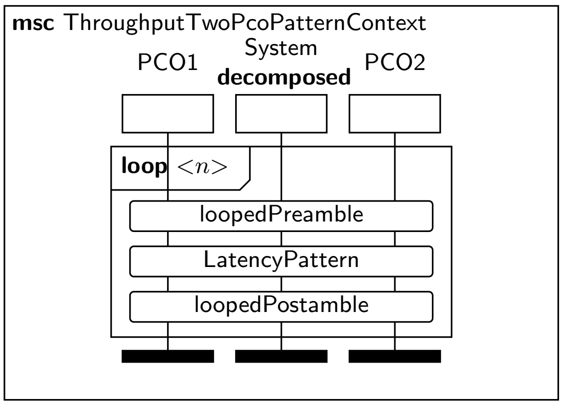
Note that even if a throughput requirement is fulfilled, this does not necessarily imply that all response time requirements are fulfilled for each of the loop’s iteration (e.g. due to bursty behaviour and buffers inside the system). Thus, when inserting a response time pattern into the throughput pattern, it has to be considered whether only the functional behaviour of a response time pattern is desired or also an additional real-time constraint. In the first case, the delay pattern has to be instantiated with the time interval $[0, \infty)$ which is equivalent to removing the real-time constraint from the response time pattern. In the latter case, an additional requirement for periodic events and their jitter is obtained.

Related patterns: A variant of this pattern is *Throughput Two PCO* which is suitable for a throughput which involves two interfaces. The response time patterns *Response Time* or *Response Time PCO* are referenced in this pattern. Patterns for periodic events have the same context, since they involve also repeated behaviour, however, they put a real-time constraint on each single occurrence of a set of events.

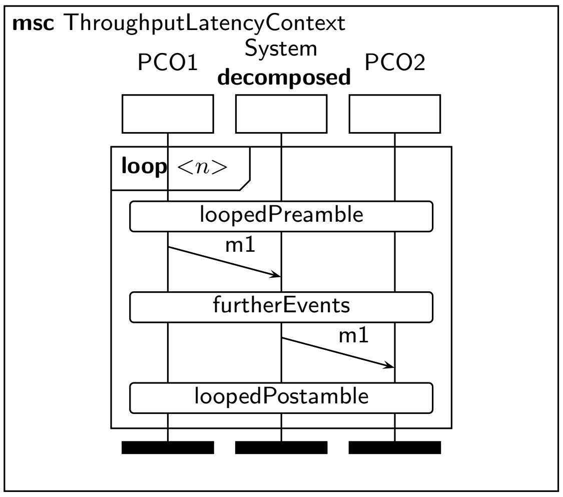
Throughput Two PCO

Intent: Impose a throughput real-time requirement on a system forwarding messages repeatedly from one interface to another.

Context: A system forwards repeatedly messages from one interface to another. The repetition can be subdivided into a preamble, into message forwarding based on the *Latency* pattern, and into a postamble.



The loop inline expression includes the references loopedPreamble, LatencyPattern, and loopedPostamble. Thus, additional behaviour, which precedes or follows the latency pattern, may be contained in the MSC references loopedPreamble and loopedPostamble. Usually, the contained latency pattern defines just the functional behaviour, which is part of the throughput requirement. Thus, possible expansions of the LatencyPattern reference yield the following MSC:



Problem: A throughput real-time requirement shall be imposed on a system forwarding messages repeatedly from one interface to another.

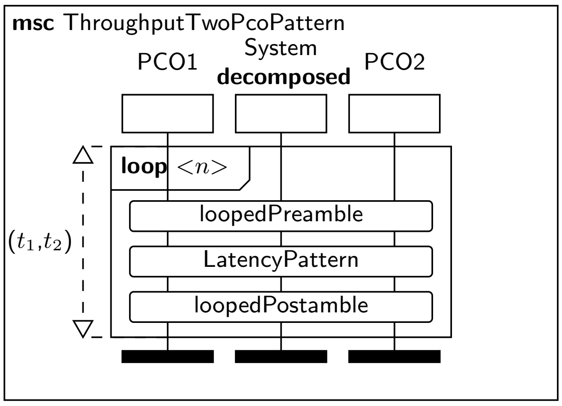
In contrast to periodic or delay-based real-time requirements, *throughput* requirements consider the system performance over a longer duration, not just for a single set of events. This means, *throughput* constrains the number of messages per time that a system has to deliver or to process repeatedly [IET90].

Roles/Parameters:

PCO1:	stimulating interface
PCO2:	observing interface
n:	number of repetitions
t_1, t_2:	lower and upper bound for all n repetitions of the loop for which the throughput real-time requirement must hold

Thus, the actual throughput *TP* imposed by this pattern is within the interval $\left(\frac{n}{t_2}, \frac{n}{t_1}\right)$.

Solution: Add a relative time constraint (t_1, t_2) to the first and last event of the repetitive behaviour, i.e. the top and the bottom of the MSC loop inline expression.



The given throughput pattern constrains a throughput *TP* to be $\frac{n}{t_2} < TP < \frac{n}{t_1}$ events per time unit. Those "events" consist typically of a set of events, in particular such according to the *Latency* pattern presented before.

Note that even if a throughput requirement is fulfilled, this does not necessarily imply that all latency requirements are fulfilled for each of the loop's iteration (e.g. due to bursty behaviour and buffers inside the system). Thus, when inserting a latency pattern into the throughput pattern, it has to be considered whether only the functional behaviour of the latency pattern is desired or also the additional real-time constraint. In the first case, the latency pattern has to be instantiated with the time interval $[0, \infty)$ which is equivalent to removing the real-time constraint from the response time pattern. In the latter case, a requirement for periodic events and their jitter is obtained.

Related patterns: A variant of this pattern is *Throughput One PCO* which is suitable for a throughput which involves just one interface. The *Latency* pattern is referenced in this pattern. Patterns for periodic events have the same context, since they involve also repeated behaviour, however, they put a real-time constraint on each single occurrence of a set of events.

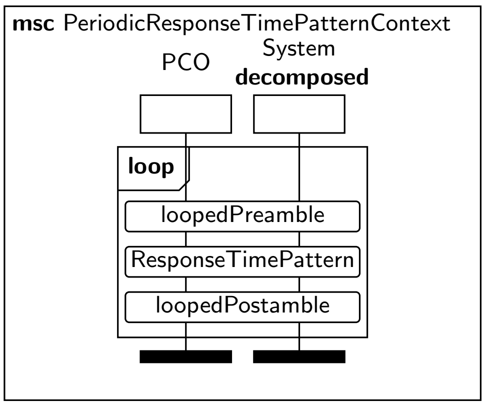
Patterns for Periodic Events

In contrast to throughput requirements, requirements for periodic events have to hold for each single execution of a periodic event. Like for the throughput requirement, iteration of events can be obtained by using MSC loop inline expressions—but for periodic requirements, the time constraint is contained inside the loop. Depending on the numbers of involved PCOs, several patterns are possible. In the following, the most common cases are listed.

Periodic Response Time and Jitter

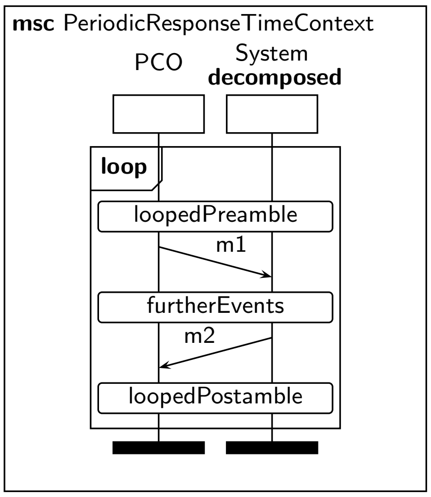
Intent: Impose a periodic response time (and thus jitter) real-time requirement on messages exchanged via one interface of a system.

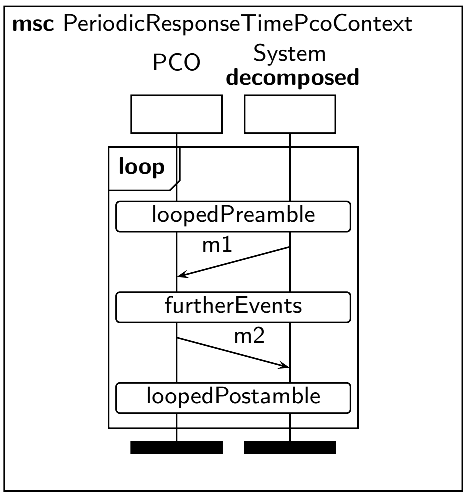
Context: A system exchanges repeatedly the same set of messages with the environment via one interface. The repetition can be subdivided into a preamble, into a message exchange based on either the *Response Time* or *Response Time PCO* pattern, and a postamble.



The loop inline expression includes the references loopedPreamble, ResponseTimePattern, and loopedPostamble. Thus, additional behaviour, which precedes or follows the response pattern, may be contained in the MSC references loopedPreamble and loopedPostamble. ResponseTimePattern refers to one of the RTC-patterns *Response Time* or *Response Time PCO*.

With respect to the context, just the functional behaviour of the contained response time patterns matters. Though, as shown in the solution below, the non-functional real-time requirement of the response time pattern is relevant for the periodic response time and jitter requirement. Thus, possible expansions of the ResponseTimePattern reference yield one of the following MSCs as context:





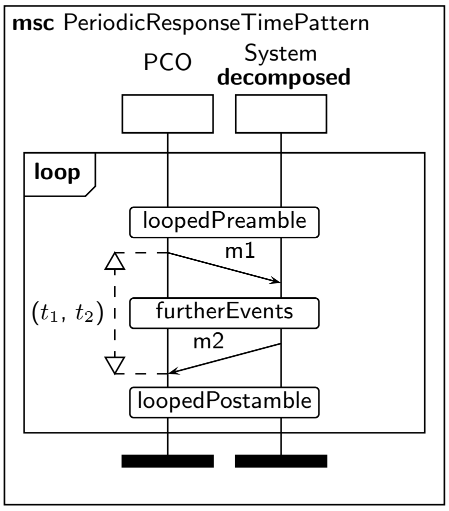
Problem: A periodic response time or jitter real-time requirement shall be imposed on messages repeatedly exchanged via one interface of a system.

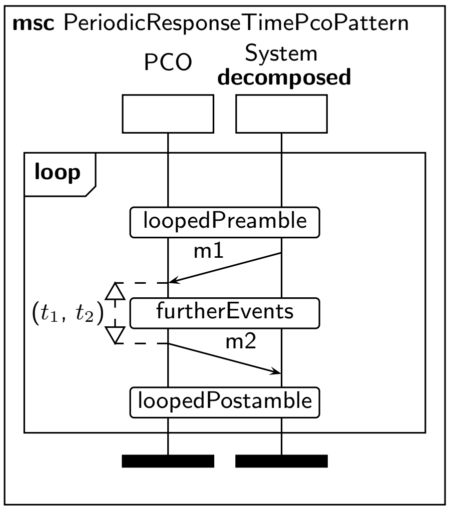
In contrast to throughput real-time requirements, *Periodic Response Time and Jitter* real-time requirements must hold for each repetition of the periodic behaviour.

Roles/Parameters:

PCO:	observing & stimulating interface
t_1, t_2:	lower and upper bound for response time and thus jitter

Solution: Add a relative time constraint (t_1, t_2) to the two events of sending message m1 and receiving message m2 at PCO (shown in MSC PeriodicResponseTimePattern) or respectively to the two events of receiving message m1 and sending message m2 at PCO (shown in MSC PeriodicResponseTimePcoPattern).





The above MSCs can also be interpreted as *response time jitter* specifications. Response time jitter describes the variation of the delay during repetition. Note that several interpretations of "jitter" exist [IET02]. Here, the following definition is used: $J_i = D_i - \overline{D}$, where $\overline{D}$ is the ideal (target) response time, D_i the actual response time of the i^{th} pair of events and thus J_i the jitter in the i^{th} repetition. Hence, a response time jitter requirement for the overall sequence of response times is expressed by the following inequation: $\forall i : J^- < J_i < J^+$, where J^- is the maximal allowed deviation below and J^+ the maximal allowed deviation above the target response time $\overline{D}$.

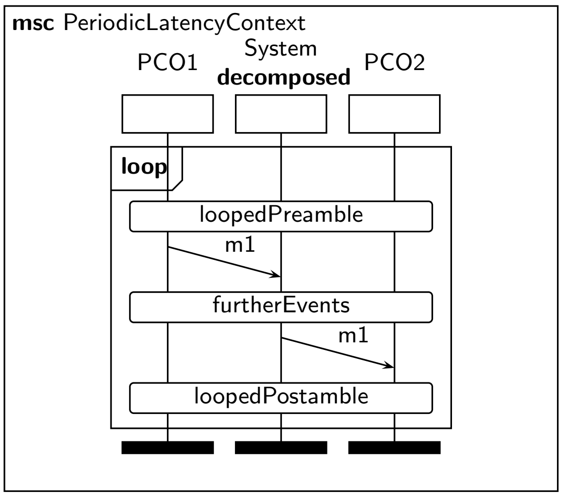
As a result, the above MSCs express a target response time $\overline{D}$, for which $t_1 < \overline{D} < t_2$ holds, and a response time jitter requirement with $J^- = t_1 - \overline{D}$ and $J^+ = t_2 - \overline{D}$. This means, the interval (t_1, t_2) could alternatively be written as $(\overline{D} + J^-, \overline{D} + J^+)$.

Related patterns: A variant of this pattern is *Periodic Latency and Jitter* which is suitable for expressing a jitter of a periodic latency requirement. The response time patterns *Response Time* or *Response Time PCO* are referenced in this pattern. The pattern *Throughput One PCO* has the same context, since it involves also repeated behaviour.

Periodic Latency and Jitter

Intent: Impose a periodic latency (and thus jitter) real-time requirement on a system forwarding messages from one interface to another.

Context: A system forwards repeatedly the same message from one interface to another. The repetition can be subdivided into a preamble, into the actual message forwarding, and a postamble.



The loop inline expression includes the references loopedPreamble and loopedPostamble. Thus, additional behaviour, which precedes or follows the message forwarding, may be contained in the MSC references loopedPreamble and loopedPostamble.

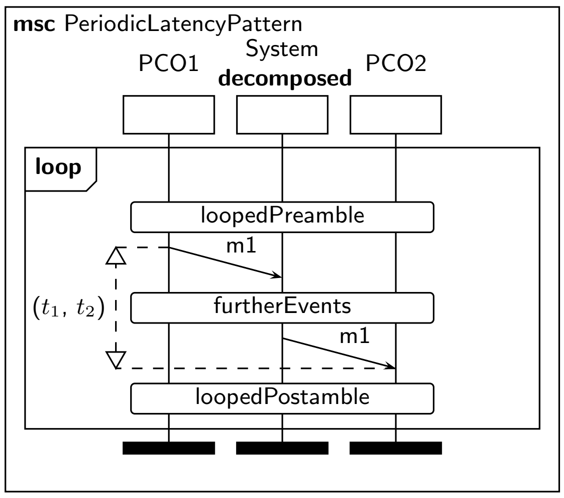
Problem: A periodic latency or jitter real-time requirement shall be imposed on a system forwarding messages repeatedly from one interface to another.

In contrast to throughput real-time requirements, *Periodic Latency and Jitter* real-time requirements must hold for each repetition of the periodic behaviour.

Roles/Parameters:

PCO1:	stimulating interface
PCO2:	observing interface
m1:	stimulus which is forwarded
t_1, t_2:	lower and upper bound for latency and thus jitter

Solution: Add a relative time constraint (t_1, t_2) to the two events of sending message m1 at PCO1 and receiving message m1 at PCO2.



The above MSC can also be interpreted as *latency jitter* specification. Latency jitter describes the variation of the latency during repetition. Note that several interpretations of "jitter" exist [IET02]. Here, the following definition is used: $J_i = D_i - \overline{D}$, where $\overline{D}$ is the ideal (target) latency, D_i the actual latency of the i^{th} pair of events and thus J_i the jitter in the i^{th} repetition. Hence, a latency jitter requirement for the overall sequence of delays is expressed by the following inequation: $\forall i : J^- < J_i < J^+$, where J^- is the maximal allowed deviation below and J^+ the maximal allowed deviation above the target latency $\overline{D}$.

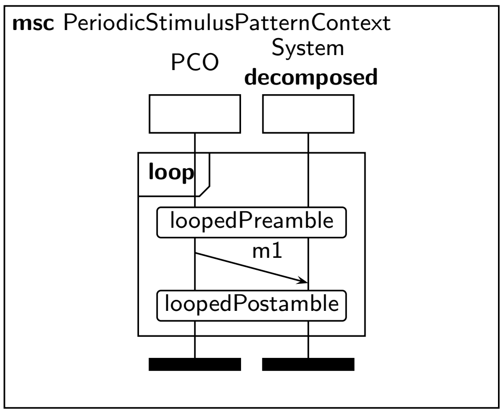
As a result, the above MSC expresses a target latency $\overline{D}$, for which $t_1 < \overline{D} < t_2$ holds, and a latency jitter requirement with $J^- = t_1 - \overline{D}$ and $J^+ = t_2 - \overline{D}$. This means, the interval (t_1, t_2) could alternatively be written as $(\overline{D} + J^-, \overline{D} + J^+)$.

Related patterns: A variant of this pattern is *Periodic Response Time and Jitter* which is suitable for expressing a jitter of a periodic response time requirement. The *Latency* pattern is the essence of this pattern. The pattern *Throughput Two PCO* has the same context, since it involves also repeated behaviour.

Periodic Stimulus and Jitter

Intent: Impose a frequency with an allowed jitter on a periodic stimulus sent to one interface of a system.

Context: A system is repeatedly stimulated by the same message via one interface. The repetition can be subdivided into a preamble, the stimulus, and a postamble.



The loop inline expression includes the references loopedPreamble and loopedPostamble which may contain additional behaviour, which precedes or follows the stimulating sending of message m1.

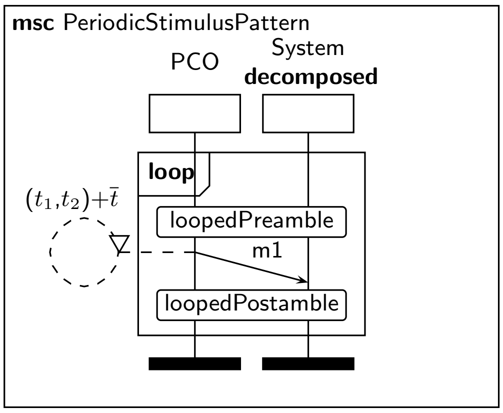
Problem: A frequency with an allowed jitter shall be imposed on messages repeatedly stimulating a system via one interface.

Roles/Parameters:

PCO:	stimulating interface
m1:	stimulus
$\bar{t}$:	period of the mean target frequency
t_1, t_2:	lower and upper bound for period of actual frequency

Alternatively, (t_1,t_2) can be written as $(\bar{t}+J^-,\bar{t}+J^+)$, where J^- is the maximum deviation below the target period and J^+ is the maximum deviation above.

Solution: Add a periodic time constraint $(t_1,t_2)+\bar{t}$ to the event of sending the stimulating message m1.



This MSC specifies a periodic sending of message m1 to the system. The requested periodicity $\bar{t}$ is specified as an additional parameter of the time interval.

Since standard MSC does not allow to attach time constraints to a pair of events which spans over adjacent repetitions of a loop, the provided solution uses an MSC extension which has been suggested in [Neu00].

Likewise to the *Periodic Response Time and Jitter* pattern, this pattern specifies also a jitter for the periodicity (and thus of the frequency). Periodicity jitter describes the variation of the periodicity during repetition. Note that several interpretations of "jitter" exist [IET02]. Here, the following definition is used: $J_i = T_i - \bar{t}$, where $\bar{t}$ is the ideal (target) period, T_i the actual period between the i^{th} and $(i+1)^{\text{th}}$ iteration and thus J_i the jitter in the i^{th} iteration.

Hence, a periodicity jitter requirement for the all iterations is expressed by the following inequation: $\forall i : J^- < J_i < J^+$, where J^- is the maximal allowed deviation below and J^+ the maximal allowed deviation above the target period $\bar{t}$.

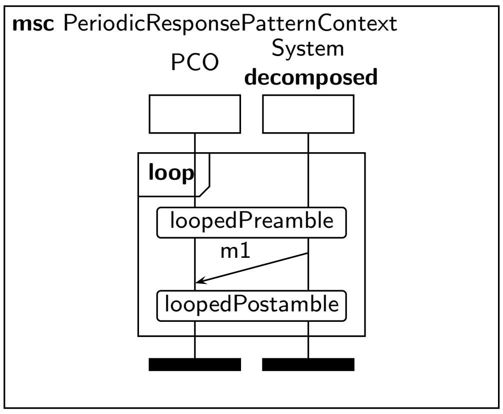
As a result, the given MSC expresses a target period $\bar{t}$, for which $t_1 < \bar{t} < t_2$ holds, and a periodicity jitter requirement with $J^- = t_1 - \bar{t}$ and $J^+ = t_2 - \bar{t}$. This means, the interval (t_1, t_2) could alternatively be written as $(\bar{t} + J^-, \bar{t} + J^+)$.

Related patterns: A variant of this pattern is *Periodic Response and Jitter* which is suitable for expressing a jitter of a periodic response requirement.

Periodic Response and Jitter

Intent: Impose a frequency with an allowed jitter on a periodic response of a system which is observed at one interface.

Context: A system sends repeatedly the same message via one interface. The repetition can be subdivided into a preamble, the actual response, and a postamble.



The loop inline expression includes the references loopedPreamble and loopedPostamble which may contain additional behaviour, which precedes or follows the reception of message m1.

Problem: A frequency with an allowed jitter shall be imposed on messages sent repeatedly by a system via one interface.

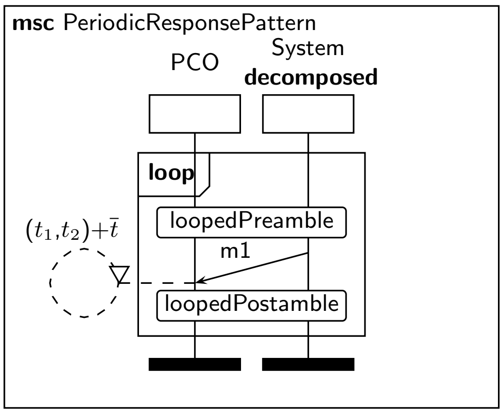
Roles/Parameters:

PCO:	observing interface
m1:	response of system
$\bar{t}$:	period of the mean target frequency
t_1, t_2:	lower and upper bound for period of actual frequency

Alternatively, (t_1,t_2) can be written as $(\bar{t}+J^-,\bar{t}+J^+)$, where J^- is the maximum deviation below the target period and J^+ is the maximum deviation above.

Solution: Add a periodic time constraint $(t_1,t_2)+\bar{t}$ to the event of receiving the response m1.

This is depicted in the MSC below. It specifies a periodic reception of message m1 from the system. The requested periodicity $\bar{t}$ is specified as an additional parameter of the time interval.



Since standard MSC does not allow to attach time constraints to a pair of events which spans over adjacent repetitions of a loop, the provided solution uses an MSC extension which has been suggested in [Neu00].

Like all patterns for periodic events in this section, this pattern specifies also a jitter for the periodicity (and thus of the frequency). Periodicity jitter describes the variation of the periodicity during repetition. Note that several interpretations of "jitter" exist [IET02]. Here, the following definition is used: $J_i = T_i - \bar{t}$, where $\bar{t}$ is the ideal (target) period, T_i the actual period between the i^{th} and $(i+1)^{\text{th}}$ iteration and thus J_i the jitter in the i^{th} iteration.

Hence, a periodicity jitter requirement for the all iterations is expressed by the following inequation: $\forall i : J^- < J_i < J^+$, where J^- is the maximal allowed deviation below and J^+ the maximal allowed deviation above the target period $\bar{t}$.

As a result, the given MSC expresses a target period $\bar{t}$, for which $t_1 < \bar{t} < t_2$ holds, and a periodicity jitter requirement with $J^- = t_1 - \bar{t}$ and $J^+ = t_2 - \bar{t}$. This means, the interval (t_1, t_2) could alternatively be written as $(\bar{t} + J^-, \bar{t} + J^+)$.

Related patterns: A variant of this pattern is *Periodic Stimulus and Jitter* which is suitable for expressing a jitter of a periodic stimulus requirement.

5.2.3 Pattern Instantiation

So far, the RTC-patterns themselves have been presented. In order to use them, the patterns need to be instantiated. In contrast to, e.g., design patterns which need to be instantiated informally by translating an object-oriented design into source code of an implementation language, the formal semantics of MSC allows to formalise the instantiation of MSC-based patterns. For this, the RTC-patterns have to be parameterised with respect to instance names, messages names, time intervals, loop boundaries, and references. While this is possible for most of its elements, the current MSC standard [ITU99b] does not allow to pass reference names and interval types as MSC parameters. Since it is not possible to parameterise all required elements of an MSC, an informal instantiation approach is suggested in this thesis. Moreover, this avoids cluttering up the MSCs of an RTC-pattern with formal parameter declarations.

Therefore, RTC-pattern instantiation is usually performed by copying the MSC given in the solution part of the pattern and pasting it into the MSC, which provides the context. Finally, the copied pattern can be modified according to the actual context. The modifications which are necessary are simple textual replacements of instance names, messages names, time intervals, and loop boundaries for loop inline expressions. References may either be expanded or just renamed.

The RTC-patterns which were presented use open time intervals in the MSCs for specifying real-time constraints. Since MSC allows an arbitrary combination of open and closed time interval boundaries, it is also valid for all of the presented RTC-patterns to adapt the interval boundaries as suitable. Thus, where given in the pattern template, corresponding inequations need to be modified as well by changing a $<$ to $\leq$.

As the *Related patterns* entries of the presented RTC-patterns suggest, RTC-patterns are closely linked to each other, i.e. they form a system of patterns (a pattern language). However, the *Related patterns* section describes the relationship of patterns just in an informal manner. A formal description of the pattern relationship is given by the MSC references, which refer to other RTC-patterns. These references determine how RTC-patterns can be composed. In general, composition of patterns is possible if the patterns do not overlap or as long as a pattern is fully contained in another pattern.

5.3 Application to Testing

In this section, it shall be demonstrated how RTC-patterns can be used for *Timed*TTCN-3 test development. First, it is described how to accompany RTC-patterns with *Timed*TTCN-3 code. Then, the application of this approach is demonstrated by providing an example for the Inres protocol.

5.3.1 Applying RTC-Patterns to *Timed*TTCN-3

The intention of RTC-patterns is to utilise them for real-time test specification and evaluation. This can be achieved by providing for each of these patterns suitable code fragments of a test language to generate and evaluate time stamps. Hence, tests can be developed either constructively from scratch by composing MSC test purposes from RTC-patterns or analytically by scanning existing test purposes for RTC-patterns. The corresponding test case can be easily derived due to the relationship between *Timed*TTCN-3 code and the RTC-patterns contained in a test purpose MSC. Figure 5.11 depicts the two possible usages of RTC-patterns. In either case, predefined evaluation functions and matching code skeletons for time stamp generation are provided by the RTC-pattern. Thus, it is guaranteed, that time stamp generation and evaluation match. Both usages of RTC-patterns may be supported by tools. In particular, a test generation tool may be guided by RTC-patterns to create and evaluate times stamps in the right way.

To gain full benefit of RTC-patterns for real-time testing, they have to be accompanied with *Timed*TTCN-3 code as an additional item of the pattern solution part. In this thesis, this is not shown for all of the presented RTC-patterns. Instead, just examples for the *Latency* and *Throughput* RTC-patterns are presented to give an idea how this approach looks like for *Timed*TTCN-3.

The application of RTC-patterns is presented for both real-time property evaluation approaches of *Timed*TTCN-3. One example demonstrates the online evaluation of a latency requirement, another example the offline evaluation of a throughput requirement. The examples are appropriate for a local test architecture. Though, as shown in the previous chapters, it is no problem to create and evaluate time stamps in the same manner for a distributed test architecture.

Figure 5.12 shows the *Timed*TTCN-3 code fragment which is to be provided as an additional item of the solution part of the *Latency* RTC-pattern. This code fragment provides a solution for testing a latency real-time requirement using online evaluation. The relevant events which constitute the functional

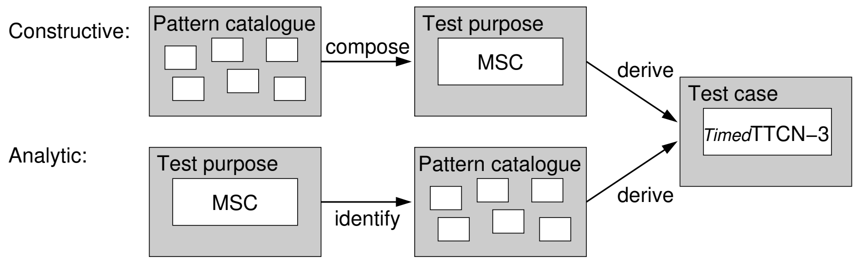


Figure 5.11: Possible Usages of RTC-patterns

```
 1  import from RTCPevaluationLibrary {
 2    function evalLatencyOnline
 3  }
 4   ...
 5  testcase latencyOnlinePattern() runs on ... {
 6    var float timeA, timeB;
 7     ...
 8    PCO1.send(m1);
 9    timeA:=self.now;
10    furtherEvents ();
11    PCO2.receive(m1);
12    timeB:=self.now;
13    setverdict(evalLatencyOnline(timeA, timeB, t1, t2));
14     ...
15  }
```

Figure 5.12: *Timed*TTCN-3 Skeleton for Online Evaluated Latency Pattern

behaviour of the *Latency* RTC-pattern are the two events of sending message m1 at PCO1 and receiving message m1 at PCO2 (cf. Section 5.2.2). Thus, the code fragment in Figure 5.12 provides the corresponding *Timed*TTCN-3 **send** and **receive** statements in lines 8 and 11. The reference to further events can be found as a function call in Line 10.

For measuring the latency, time stamps for these events need to be generated. Therefore, the *Timed*TTCN-3 code fragment contains calls of the **self.now** operation to obtain the current value of the clock in Line 9 just after sending m1 to the SUT and in Line 12 just after m1 is received. Since online evaluation is used, the time stamps are assigned to ordinary *Timed*TTCN-3 **float** variables which are defined in Line 6 of Figure 5.12. The online evaluation function evalLatencyOnline for assessing the generated time stamps timeA and timeB with respect to the latency real-time requirement is called in Line 13. The actual parameters are the generated time stamps and the upper and lower latency bound. Since the code fragment accompanies the *Latency* RTC-pattern, the same abstract names for the latency bounds (t1, t2) as in the corresponding MSC are used. However, for instantiating the pattern code fragment, they have to be replaced as well as, e.g., the message and port names.

The definition of the evaluation function evalLatencyOnline is not provided in Figure 5.12. Instead, it is imported from the *Timed*TTCN-3 module RTCPevaluationLibrary as shown in lines 1–3. In addition to the *Timed*TTCN-3 code fragments, a predefined library of evaluation functions and time stamp type definitions is provided together with the patterns.

An excerpt from this library is shown in Figure 5.13. The definition of the evalLatencyOnline evaluation function can be found in lines 7–16. Its implementation is exactly the same as in Figure 3.8 of Chapter 3.

```
module RTCPevaluationLibrary {
  type record ThroughputTimestampType {
    float logtime,
    charstring id
  }

  function evalLatencyOnline(float timeA, float timeB,
                       float lowerbound, float upperbound) return verdicttype {
    var float latency:=timeB−timeA;
    if ((latency<upperbound) and (lowerbound<latency)) {
      return pass; // Non−functional pass
    }
    else {
      return conf; // Non−functional fail
    }
  }

  function evalThroughputOffline(charstring loopEntryId, charstring loopExitId,
                                 float lowerThroughput, float upperThroughput,
                                 integer n, logfile timelog) return verdicttype {
    var float timeDiff;
    var ThroughputTimestampType stampA, stampB;
    if (timelog.first(ThroughputTimestampType:{?,−},
                     ThroughputTimestampType:{?, loopEntryId})==true) {
      stampA:=timelog.retrieve; // Get current time stamp entry
      if (timelog.next(ThroughputTimestampType:{?, loopExitId})==true) {
        stampB:=timelog.retrieve; // Get current time stamp entry
      }
      else {
        return fail; // Error while retrieving log
      }
    }
    else {
      return fail; // Error while retrieving log
    }
    timeDiff:=stampB.logTime−stampA.logTime;
    if ((lowerThroughput<int2float(n)/timeDiff)
        and (int2float(n)/timeDiff<upperThroughput)) {
      return pass; // Non−functional pass
    }
    else {
      return conf; // Non−functional fail
    }
  }
} // End of module RTCPevaluationLibrary
```

Figure 5.13: *Timed*TTCN-3 Library with Evaluation Functions for Patterns

The *Timed*TTCN-3 code fragment which is to be provided together with *Throughput Two PCO* RTC-pattern (cf. Section 5.2.2) is shown in Figure 5.14. This time, the code fragment demonstrates how to apply the offline evaluation approach to the throughput pattern.

```
 1  import from RTCPevaluationLibrary {
 2    type ThroughputTimestampType;
 3    function evalThroughputOffline
 4  }
 5   ...
 6  testcase throughputOfflinePattern(integer n) runs on ... {
 7    var integer i;
 8     ...
 9    log(ThroughputTimestampType:{self.now, "loopBegin"});
10    for (i:=0; i<n; i:=i+1) {
11      loopedPreamble();
12      LatencyPattern();
13      loopedPostamble();
14    }
15    log(ThroughputTimestampType:{self.now, "loopEnd"});
16     ...
17  }
18
19  control {
20    var testrun myTestrun;
21    var logfile myLog;
22    var verdicttype myVerdict;
23    myTestrun:=execute(throughputOfflinePattern(n));
24    myVerdict:=myTestrun.getverdict;
25    if (myVerdict==pass) {
26      myLog:=myTestrun.getlog;
27      myVerdict:=evalThroughputOffline("loopBegin", "loopEnd",
28                    int2float (n)/upperbound, int2float(n)/lowerbound, n, myLog);
29      myTestrun.setverdict(myVerdict);
30    }
31  }
```

Figure 5.14: Skeleton for Offline Evaluated Throughput Pattern

The events on which the throughput requirement is imposed are executed in a loop. Thus, the *Timed*TTCN-3 code fragment contains a **for** loop in lines 10–14. Since for assessing a throughput requirement just the number of iterations and the overall duration is of interest, only the time stamps for the points in time immediately before and after the execution of the loop are created and stored in a log file (lines 9 and 15). The predefined ThroughputTimestampType record type definition used as time stamp type is imported from the RTCPevaluationLibrary module (Line 2).

Since offline evaluation is used in the example, the *Timed*TTCN-3 code associated to the *Throughput Two PCO* RTC-pattern provides additionally a code snippet for the module control part (lines 19–31 of Figure 5.14). The structure of the control part is the same as in the examples presented in the previous chapters, i.e. first, the test case throughputOffline is executed (Line 23) and afterwards, the offline evaluation function evalThroughputOffline is called (lines 27 and 28). The parameters of the evaluation function are the identifiers of the log file entries, the upper and lower throughput bounds[3], the number of iterations, and the log file generated by the test case. Likewise to the latency evaluation function, the evalThroughputOffline evaluation function is also imported from the RTCPevaluationLibrary module as shown in Line 3 of Figure 5.14.

The definition of the evalThroughputOffline evaluation function is given in lines 18–44 of Figure 5.13. The function has six parameters: the labels of the entry and exit time stamps surrounding the loop (loopEntryId, loopExitId), the lower and upper throughput bounds (lowerThroughput, upperThroughput), the number of iterations (n), and the log file to evaluate (timelog). Lines 23–27 navigate to the relevant time stamps in the log file and retrieve the entries: The operation **first** (lines 23–24) sorts and restricts the log file entries and moves a cursor to the first matching entry in the log file. A "**?**" indicates the field used as sorting key. The second parameter of the **first** operation is used to move the cursor to the entry which relates to the loopEntryId. The log file entry which matches is extracted by the **retrieve** operation (Line 25). The operation **next** (Line 26) advances the cursor to the subsequent time stamp with a label identified by loopExitId. The calculation of the actual throughput value is performed in lines 36–43 based on the arithmetic expression for throughput presented in Section 5.2.2. Depending on the evaluation, the function returns a **pass** verdict if the real-time requirement is met, or a **conf** verdict if the requirement is violated. A **fail** verdict is returned if the desired time stamps were not found in the log file.

5.3.2 An Inres-based Example

The *Timed*TTCN-3 code which accompanies the RTC-patterns shall now be utilised in an example. The aim is to create a *Timed*TTCN-3 real-time test case from an MSC test purpose by using RTC-patterns.

Figure 5.15 depicts an MSC real-time test purpose for testing an Initiator implementation of the Inres protocol with respect to some real-time requirements. The SUT can be accessed via the PCOs ISAP and MSAP. The functional behaviour contained in the test purpose is the transfer data 100 times. For doing this, a connection needs to be established. After the test, the connection has to be released. The real-time requirements of the test purpose are to test:

1. a latency constraint imposed on the messages IDATreq and MDATind,

2. a throughput constraint on the events contained in the loop construct.

[3]The throughput bounds are calculated from the number of iterations and the interval bounds as explained in the *Throughput Two PCO* RTC-pattern in Section 5.2.2.

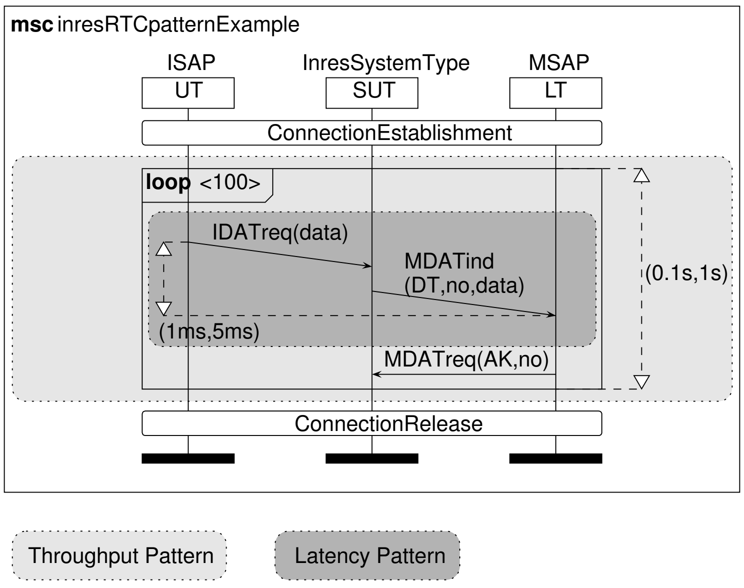


Figure 5.15: RTC-patterns in the Inres Test Purpose MSC

In this example, the latency between sending message IDISreq and receiving message MDATind shall be evaluated during the test execution (i.e. online), whereas the throughput of the loop construct shall be evaluated after the test execution (i.e. offline). The MSC diagram does not define which evaluation mechanism is desired since the MSC language does not provide the possibility to express this. Such information is considered as additional directives for test generation.

When scanning through the given MSC diagram, the RTC-patterns *Latency* and *Throughput Two PCO* can be recognised. The shaded areas in Figure 5.15 show, where both patterns are located in the diagram. The associated *Timed*TTCN-3 code fragments and the predefined evaluation module provided in figures 5.12, 5.13, and 5.14 of the previous section can be used to create a real-time test case which assesses the given test purpose.

Figure 5.16 shows the *Timed*TTCN-3 module which can be generated from the MSC diagram in Figure 5.15 by applying RTC-patterns. In lines 2–5, all required type and function definitions that are provided by the RTCPevaluationLibrary module are imported. The second import statement in Line 6 has to be added to provide access to all the Inres specific type definitions which are defined in the module inresDefinitions. These application specific definitions cannot be generated using RTC-patterns, but have to be manually specified as shown in Section 2.5.1.

The code fragments associated to the RTC-patterns which have been identified in the test purpose MSC can be found in the test case inresRTCpatternExample (lines 8–29). For example, the *Timed*TTCN-3 code associated to the

```
 1 module inresRTCpatternExampleModule {
 2   import from RTCPevaluationLibrary {
 3     type ThroughputTimestampType;
 4     function evalThroughputOffline, evalLatencyOnline
 5   }
 6   import from inresDefinitions all;
 7
 8   testcase inresRTCpatternExample(integer n) runs on InresSystemType {
 9     var integer i; // From Throughput pattern
10     var float timeA, timeB; // From Latency pattern
11     ConnectionEstablishment();
12     // Throughput pattern scheme begin
13     log(ThroughputTimestampType:{self.now, ”loopBegin”});
14     for (i:=0; i<n; i:=i+1) {
15       // Latency pattern scheme begin
16       ISAP.send(IDATreq:{data});
17       timeA:=self.now;
18       MSAP.receive(MDATind:{DT,number,data});
19       timeB:=self.now;
20       setverdict(evalLatencyOnline(timeA, timeB, 0.001, 0.005));
21       // Latency pattern scheme end
22       MSAP.send(MDATreq:{AK,number});
23     }
24     log(ThroughputTimestampType:{self.now, ”loopEnd”});
25     // Throughput pattern scheme end
26     ConnectionRelease();
27     setverdict(pass);
28     stop;
29   }
30
31   control {
32     var testrun myTestrun;
33     var logfile myLog;
34     var verdicttype myVerdict;
35     myTestrun:=execute(inresRTexample(100));
36     myVerdict:=myTestrun.getverdict;
37     if (myVerdict==pass) {
38       myLog:=myTestrun.getlog;
39       myVerdict:=evalThroughputOffline(”loopBegin”, ”loopEnd”,
40                     int2float (100)/1.0, int2float (100)/0.1, 100, myLog);
41       myTestrun.setverdict(myVerdict);
42     }
43   }
44 } // End of module inresRTCpatternExampleModule
```

Figure 5.16: Test Case Generated from Figure 5.15 and RTC-patterns

Throughput Two PCO pattern contributed to lines 9, 13–14, and 23–24 of Figure 5.16. In comparison to the code fragment in Figure 5.14, the function call loopedPreamble has been removed since it is empty, loopedPostamble has been replaced by MSAP.**send**(MDATreq:AK,number). Furthermore, parts of the **import** block (lines 2–5) and the whole control part in lines 31–43 are created from the *Timed*TTCN-3 code provided in Figure 5.14. For instantiation of the pattern code fragments, the abstract names have been replaced, e.g. n by 100, upperbound by 1.0, and lowerbound by 0.1.

The *Timed*TTCN-3 code fragments in Figure 5.12 which are provided together with the *Latency* pattern can be found in lines 10 and 16–20 of Figure 5.16. For instantiation of the code fragments, the abstract name PCO1 has been replaced by ISAP, PCO2 by MSAP, m1 by IDATreq:{data}, and MDATind:{DT,number,data}, respectively. The reference to furtherEvents has been dropped because it is empty. The interval bounds t1 and t2 have been instantiated using the actual values 0.001 and 0.005.

This example demonstrated that is possible to create a real-time test case from the *Timed*TTCN-3 code fragments accompanying the RTC-patterns which have been identified in a test purpose MSC. Since *Timed*TTCN-3 code for both, generation and evaluation of time stamps, is provided in combination, it is guaranteed that both fit together. Hence, RTC-patterns do not only speed up test development but reduce also the risk of erroneous test cases.

5.4 Summary

In this chapter, test patterns have been treated. After giving an introduction into existing patterns at large and test patterns in particular, a pattern classification scheme which is suitable for test patterns has been developed and assessed. Additionally, the benefit of a unified pattern notation based on a template has been discussed.

As the main contribution of this chapter, *Real-time Communication patterns* (RTC-patterns) have been introduced. These patterns support the specification of delay, throughput, and periodic real-time requirements for distributed systems. Thus, RTC-patterns improve the requirements definition and the specification phase of real-time system development. Moreover, these patterns can be applied to real-time test specification. Hence, RTC-patterns can be used as part of an integrated development methodology, which eases test development.

The suggested pattern-based real-time test development approach provides an unambiguous way of generating time stamps and evaluating them using corresponding evaluation functions. This is achieved by providing predefined *Timed*TTCN-3 code for both of these activities as part of an RTC-pattern. Thus, it is guaranteed that generation and evaluation of time stamps fit together.

Related Work

A similar pattern-based approach, which applies for passive testing, can be found in [HBUP03]. It is based on [UHPB03] which has been briefly discussed in Section 3.8. The approach is to obtain a trace of a distributed system under test by monitoring. Then, it can be checked, whether the trace fulfils temporal properties or not. The specification of temporal properties is composed from property patterns which are described in [DAC99]. However, the used property patterns allow only to describe temporal relations [Pnu77], i.e. "event b occurs *after* event a", but not hard real-time properties. Moreover, this approach supports just passive testing. In contrast to the presented RTC-patterns, it does hence not deal with the problem of harmonising test behaviour and test evaluation.

Even though a new class of test patterns has been provided in this chapter, further work on test patterns is possible. As the introduced classification of known test patterns illustrates, a lot of areas in the test pattern space are not covered, yet. For example, for functional system test patterns, a promising work has been started in the ETSI work item *Patterns in Test Development* (PTD) [ETS04].

Chapter 6

Conclusion

In this thesis, languages, tools, and patterns for the specification of distributed real-time tests have been presented. As a test specification language, *Timed*TTCN-3 has been introduced. To ease the development of *Timed*TTCN-3 real-time tests, computer aided test generation has been discussed and a pattern-based approach for the specification of real-time requirements and a harmonised subsequent test generation has been developed.

The proposed real-time test language *Timed*TTCN-3 is based on the *Testing and Test Control Notation version 3* (TTCN-3), a standardised language for the specification of distributed black-box tests. TTCN-3 allows to describe pure functional tests, only. Thus, *Timed*TTCN-3 adds some real-time extensions. *Timed*TTCN-3 introduces the concept of absolute time into TTCN-3, provides a means to specify clock-synchronised test components, extends the TTCN-3 logging mechanism, supports online and offline evaluation of tests, and adds a new test verdict to the existing TTCN-3 test verdicts.

For automatic test case generation, an approach has been presented which allows to derive *Timed*TTCN-3 real-time test cases from real-time test purposes. For the formalisation of test purposes, *Message Sequence Chart*s (MSCs) are used. It has been shown, how to derive real-time test cases for local and for distributed test architectures. The underlying transformation rules have been implemented by a tool which is able to generate *Timed*TTCN-3 test cases for local test architectures.

In addition to an automatic test generation approach, the benefit of using patterns for test development in general has been discussed. A survey on existing test patterns has been provided and a suitable test pattern classification scheme has been developed. For harmonising the specification and testing of real-time requirements, *Real-time Communication pattern*s (RTC-patterns) have been introduced. RTC-patterns provide solutions for using MSCs as building blocks for expressing real-time requirements in, e.g., test purposes. By associating *Timed*TTCN-3 code to each RTC-pattern, matching real-time test cases and test evaluation functions can be easily obtained for each RTC-pattern contained in a real-time test purpose.

6.1 Related Work

Approaches related to TTCN-based real-time test specification, generation of TTCN test cases from MSC test purposes and pattern-based property specification have already been mentioned at the end of each of the preceding three chapters. In the following, just two further test approaches which involve time shall be discussed.

JUnitPerf [Cla04] is an extension of the Java-based unit test framework *JUnit* [GMB04]. *JUnitPerf* provides a *TimedTest* decorator to impose upper bounds on the execution time of existing functional JUnit test cases. Additionally, a *LoadTest* decorator may be used for executing a test case simultaneously several times. In comparison to *Timed*TTCN-3, JUnitPerf has several drawbacks. Since it is intended for unit tests, it does not support distributed testing. Furthermore, the provided means for real-time testing are quite limited and, in fact, more comparable to elements of the TTCN-3 module control part: The TimedTest decorator behaves like the optional time supervision parameter of the TTCN-3 **execute** statement, i.e. it relates to the whole test case including the duration contributed by the pre- and postambles. The LoadTest decorator is comparable to a loop in a TTCN-3 module control part. Though, JUnitPerf supports concurrent calls of a test case. In TTCN-3, this cannot be achieved from within the module control part.

The *UML 2.0 Testing Profile* (U2TP) [OMG04b, DGNP04] is a profile for the version 2.0 of the *Unified Modeling Language* [OMG03a, OMG03b]. It allows the specification of abstract black-box tests using UML diagrams. The profile adds four concept packages to UML in order to cover the aspects *test architecture*, *test behaviour*, *test data*, and *time*. The test architecture concepts provide support for describing a test architecture using structural UML diagrams, the test behaviour concepts allow to specify test behaviour using any UML behavioural diagram. For real-time testing, the time concepts are of relevance.

The time concepts added to UML by U2TP are inspired by TTCN-3 and even *Timed*TTCN-3. For example, U2TP provides TTCN-3-like timers, which can be started, stopped, read, and they may trigger a timeout event. From *Timed*TTCN-3, the notion of timezones is adopted. Furthermore, the predefined UML 2.0 types *Duration* and *Time* may be used for storing relative and absolute time values. Like in *Timed*TTCN-3, the keyword *now* represents the current time. UML 2.0 sequence diagrams may, e.g., be used to specify test behaviour. Since UML 2.0 sequence diagrams are inspired by MSC, similar time constraints may be expressed.[1]

At the first glance, U2TP resembles the *Graphical Presentation Format for TTCN-3* (GFT). In fact, U2TP is much more abstract than GFT and TTCN-3, and thus, also more abstract than *Timed*TTCN-3. For example, for the implementation of U2TP tests, mappings to JUnit and TTCN-3 exist. However, both languages differ semantically, which shows that different interpretation of U2TP test suites are possible. The reason is that UML and U2TP have semantic variation points, which allows different test implementations for the same U2TP test specification. For example, the semantics of message reception at test components is not defined in detail, i.e. whether queues are used and, if yes, whether one queue per interface exists or just one queue for a test component which is shared by all its interfaces. Hence, in contrast to *Timed*TTCN-3, the implementation of real-time tests from U2TP test specifications is neither precise nor obvious. Another example are sequence diagrams which can be used in U2TP test behaviour specifications to impose time constraints on message exchanges. However, in U2TP, it cannot be specified how to actually evaluate the fulfilment of a real-time requirement. In fact, U2TP assumes to specify real-time requirements more in a test purpose-like style.

6.2 Outlook

Concerning *Timed*TTCN-3, a standardisation of the language is desirable. Thus, during the development of *Timed*TTCN-3, solutions have been chosen which reuse existing concepts of TTCN-3 and require only minimal changes to the existing TTCN-3 language. Corresponding change requests [Neu02, Dai03] have already been submitted to the *European Telecommunications Standards Institute* (ETSI) which is responsible for the maintenance of TTCN-3. Parts of the *Timed*TTCN-3 proposal have already influenced TTCN-3, e.g. the syntax for setting and getting a test verdict. Furthermore, it is under discussion to adopt local clocks into the standard as well as the introduction of more formalised log file format. In case of a standardisation of *Timed*TTCN-3, a formal operational semantics which refines the existing flow graph-based semantics of TTCN-3 is required.

Moreover, it would be worthwhile to assess the general applicability of *Timed*TTCN-3 to performance testing, i.e. testing of soft real-time requirements. Even though *Timed*TTCN-3 was developed with having hard real-time requirements in mind, first experiments demonstrated that at least

[1]In [DS04], a possible influence of U2TP's time concepts on the UML profile *Profile for Schedulability, Performance and Time Specification* [OMG03c] is investigated. Even though the latter profile does not relate to testing, its concepts might in turn be used in U2TP test specifications.

simple real-time properties, like mean inter-arrival times, can be evaluated using TimedTTCN-3.

The *UML 2.0 Testing Profile* was partly influenced by TimedTTCN-3. Though, not all concepts of TimedTTCN-3 are available in U2TP. Thus, it would be reasonable to investigate the necessary extensions of U2TP to allow a mapping between U2TP and TimedTTCN-3.

Regarding test patterns, a more general application of patterns in the test development process seems very promising as well as the adoption of other recent software development techniques also for test development. For example, refactoring [Fow00] might as well be applied to test suites, and for real-time test specification, aspect-oriented programming [KLM+97] might be considered as a means for non-intrusive instrumentation of test cases in order to generate time stamps.

The experiences from the development of the RTC-patterns are valuable for test patterns in general. First work on this topic has started in the *Patterns in Test Development* (PTD) ETSI work item. Hence, this thesis does not only provide a real-time test specification language and tools, but also methodological support for a broader area of test development.

Acronyms

ASN.1 *Abstract Syntax Notation One*

ATS *Abstract Test Suite*

CORBA *Common Object Request Broker Architecture*

CTMF *Conformance Testing Methodology and Framework*

ETS *Executable Test Suite*

ETSI *European Telecommunications Standards Institute*

FIFO *First In First Out*

GFT *Graphical Presentation Format for TTCN-3*

GPS *Global Positioning System*

HMSC *High-level Message Sequence Chart*

IDL *Interface Definition Language*

IEC *International Electrotechnical Commission*

IPv6 *Internet Protocol version 6*

ISO *International Organization for Standardization*

ITU *International Telecommunication Union*

IUT *Implementation Under Test*

LT *Lower Tester*

MSC *Message Sequence Chart*

MTBF *Mean Time Between Failure*

MTC *Main Test Component*

OSI *Open Systems Interconnection*

PCO *Point of Control and Observation*

PDU *Protocol Data Unit*

PIXIT *Protocol Implementation eXtra Information for Testing*

PTC *Parallel Test Component*

PTD *Patterns in Test Development*

QoS *Quality of Service*

RTC-pattern *Real-time Communication pattern*

SAP *Service Access Point*

SDL *Specification and Description Language*

SUT *System Under Test*

TC *Test Component*

TCI *TTCN-3 Control Interface*

TCL *Tool Command Language*

TFT *Tabular Presentation Format*

TRI *TTCN-3 Runtime Interface*

TTCN *Tree and Tabular Combined Notation*

TTCN-3 *Testing and Test Control Notation version 3*

UML *Unified Modeling Language*

UT *Upper Tester*

U2TP *UML 2.0 Testing Profile*

VoIP *Voice over IP*

XML *Extensible Markup Language*

Bibliography

[AH91] R. Alur and T.A. Henziger. Logics and Models of Real Time: A Survey. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory and Practice*, volume 600 of *Lecture Notes in Computer Science (LNCS)*, pages 74–106. Springer, June 1991.

[AIS$^+$77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[Ale79] C. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979.

[Anl98] M. Anlauf. Programming service tests with TTCN. In A. Petrenko and N. Yevtuschenko, editors, *Testing of Communicating Systems*, volume 11. Kluwer, 1998.

[ATM99a] ATM Forum Performance Testing Specification (AF-TEST-TM-0131.000). The ATM Forum Technical Committee, 1999.

[ATM99b] ATM Forum Traffic Management Specification Version 4.1 (AF-TM-0121.000). The ATM Forum Technical Committee, 1999.

[ATM00] ATM Forum UNI Signalling Performance Test Suite (AF-TEST-0158.000). The ATM Forum Technical Committee, 2000.

[Bal98] H. Balzert. *Lehrbuch der Software-Technik*, volume 2. Spektrum Akademischer Verlag, 1998.

[BBJ$^+$02] P. Baker, P. Bristow, C. Jervis, D. King, and B. Mitchell. Automatic Generation of Conformance Tests From Message Sequence Charts. In *Proceedings of the 3rd SAM (SDL and MSC) Workshop – Telecommunications and beyond: The Broader Applicability of SDL and MSC*, volume 2599 of *Lecture Notes in Computer Science (LNCS).* Springer, June 2002.

[Bei95] B. Beizer. *Black-Box Testing.* Wiley, 1995.

[Bin99] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley, 1996.

[BRS01] P. Baker, E. Rudolph, and I. Schieferdecker. Graphical Test Specification – The Graphical Format of TTCN-3. In R. Reed and J. Reed, editors, *SDL2001 – Meeting UML*, volume 2078 of *Lecture Notes in Computer Science (LNCS).* Springer, June 2001.

[Buc96] R.W. Buchanan. *The Art of Testing Network Systems.* Wiley, 1996.

[BW97] J. Bi and J. Wu. Application of a TTCN based conformance test environment on the Internet email protocol. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, volume 10. Chapman & Hall, 1997.

[Cla04] M. Clark. JUnitPerf. `http://www.clarkware.com/software/JUnitPerf.html`, 2004.

[Cli04] M. Clifton. Advanced Unit Test, Part V – Unit Test Patterns. `http://www.codeproject.com/gen/design/autp5.asp`, 2004.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[Cop92] J. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[DAC99] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99).* IEEE, May 1999. (The corresponding pattern catalog is available via `http://patterns.projects.cis.ksu.edu/`).

[Dai03] Z.R. Dai. TTCN-3 Change Request No. 232. Submitted to European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), June 2003.

[Dai05] Z.R. Dai. *(to appear).* PhD thesis, Berlin, 2005.

[Dan04] Danet TTCN Toolbox – TTCN-3. http://www.bss.danet.de/index_uk.asp?/solution/ttcn/ttcn_toolbox_ttcn-3_uk.htm, 2004.

[Dar88] I. F. Darwin. *Checking C Programs with lint.* O'Reilly, 1988.

[DaV04] Da Vinci Communications Terzo tools product information. http://www.davinci-communications.com/products_ttcn3.html, 2004.

[DeB95] D. DeBruler. Telecommunications Distributed Processing Patterns. http://www1.bell-labs.com/user/cope/Patterns/DistributedProcessing/DeBruler/index.html, 1995.

[DGN02] Z.R. Dai, J. Grabowski, and H. Neukirchen. TimedTTCN-3 – A Real-Time Extension for TTCN-3. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems*, volume 14, Berlin, March 2002. Kluwer.

[DGN03] Z.R. Dai, J. Grabowski, and H. Neukirchen. TimedTTCN-3 Based Graphical Real-Time Test Specification. In D. Hogrefe and A. Wiles, editors, *Testing of Communicating Systems*, volume 2644 of *Lecture Notes in Computer Science (LNCS).* Springer, May 2003.

[DGNP04] Z.R. Dai, J. Grabowski, H. Neukirchen, and H. Pals. From Design to Test with UML – Applied to a Roaming Algorithm for Bluetooth Devices. In R. Hierons, editor, *Testing of Communicating Systems*, volume 2978 of *Lecture Notes in Computer Science (LNCS).* Springer, May 2004.

[Dij70] E.W. Dijkstra. Notes on Structured Programming. Technical Report 70-WSK-03, Technological University Eindhoven, Department of Mathematics, April 1970.

[dMHB$^+$91] J. de Meer, V. Heymer, J. Burmeister, R. Hirr, and A. Rennoch. Distributed Testing. In J. Kroon, R.J. Heijink, and E. Brinksma, editors, *Protocol Test Systems IV.* North-Holland, 1991.

[DS04] Z.R. Dai and I. Schieferdecker. Time Concepts for UML 2.0 Based Testing. In *Workshop on the usage of the UML profile for Scheduling, Performance and Time (SIVOES 2004), hold in conjunction with the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, May 2004.

[DST04] S. Dibuz, T. Szabó, and Zsolt Torpis. BCMP Performance Test with TTCN-3 Mobile Node Emulator. In R. Hierons, editor, *Testing of Communicating Systems*, volume 2978 of *Lecture Notes in Computer Science (LNCS)*. Springer, May 2004.

[Ebn04] M. Ebner. *UML-based Test Specification for Communication Systems – A Methodology for the use of MSC and IDL in Testing*. PhD thesis, University of Göttingen (Germany), March 2004.

[ETS02a] ETSI European Standard (ES) 201 873-1 V2.2.1 (2002-08). The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), also published as ITU-T Recommendation Z.140, 2002.

[ETS02b] ETSI European Standard (ES) 201 873-2 V2.2.1 (2002-08). The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular Presentation Format (TFT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), also published as ITU-T Recommendation Z.141, 2002.

[ETS03a] ETSI European Standard (ES) 201 873-3 V2.2.2 (2003-04): The Tree and Tabular Combined Notation version 3; Part 3: Graphical Presentation Format for TTCN-3 (GFT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), also published as ITU-T Recommendation Z.142, 2003.

[ETS03b] ETSI European Standard (ES) 201 873-4 V2.2.1 (2003-02). The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), also published as ITU-T Rec. Z.140., 2003.

[ETS03c] ETSI European Standard (ES) 201 873-5 V1.1.1 (2003-02). The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), 2003.

[ETS03d] ETSI European Standard (ES) 201 873-6 V1.1.1 (2003-07). The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), 2003.

[ETS04] ETSI Draft Technical Report DTR/MTS-00091. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), 2004.

[EYL02] M. Ebner, A. Yin, and M. Li. Definition and Utilisation of OMG IDL to TTCN-3 Mappings. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems – Application to Internet Technologies and Services*, volume 14. Kluwer, March 2002.

[FLS04] K. Frühauf, J. Ludewig, and H. Sandmayr. *Software-Prüfung.* VdF Hochschulverlag, 2004.

[Fow00] M. Fowler. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 2000.

[FW90] D.P. Freedman and G.M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products.* Dorset House Publishing Company, 1990.

[Gec98] R. Gecse. Conformance testing methodology of Internet protocols – Internet application-layer protocol testing – the Hypertext Transfer Protocol. In A. Petrenko and N. Yevtuschenko, editors, *Testing of Communicating Systems*, volume 11. Kluwer, 1998.

[Gep01] B. Geppert. *The SDL Pattern Approach – A Reuse-Driven SDL Methodology for Designing Communication Software Systems.* PhD thesis, University of Kaiserslautern (Germany), July 2001.

[GG93] T. Gilb and D. Graham. *Software Inspection.* Addison-Wesley, 1993.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[GHN93] J. Grabowski, D. Hogrefe, and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. In O. Færgemand and A. Sarma, editors, *SDL'93 – Using Objects.* North-Holland, 1993.

[GHR$^+$03] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). *Computer Networks*, 42(3), June 2003.

[GKS00] R. Gecse, P. Krémer, and J. Szabó. HTTP Performance Evaluation with TTCN. In H. Ural, R.L. Probert, and G. von Bochmann, editors, *Testing of Communicating Systems*, volume 13. Kluwer, 2000.

[GKSH99] J. Grabowski, B. Koch, M. Schmitt, and D. Hogrefe. SDL and MSC Based Test Generation for Distributed Test Architectures. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 – The next Millenium.* Elsevier Science Publishers B.V., June 1999.

[GMB04] E. Gamma, E. Meade, and K. Beck. JUnit. `http://junit.sourceforge.net/`, 2004.

[GNS$^+$02] S. Graf, H. Neukirchen, R. Sinnott, W. Skelton, and L. Ritchie. Formalisation of the Timed Extensions. Interval Deliverable D13.2, June 2002.

[Gra02] J. Grabowski. *Specification Based Testing of Real-Time Distributed Systems.* Habilitation thesis, Universität zu Lübeck, 2002.

[GW98] J. Grabowski and T. Walter. Visualisation of TTCN test cases by MSCs. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC – SAM'98*, 1998.

[HBUP03] H. Hallal, S. Boroday, A. Ulrich, and A. Petrenko. An Automata-Based Approach to Property Testing in Event Traces. In D. Hogrefe and A. Wiles, editors, *Testing of Communicating Systems*, volume 2644 of *Lecture Notes in Computer Science (LNCS).* Springer, May 2003.

[Hil04] Patterns Library. `http://hillside.net/patterns/`, 2004.

[HKN01] D. Hogrefe, B. Koch, and H. Neukirchen. Some Implications of MSC, SDL and TTCN Time Extensions for Computer-aided Test Generation. In R. Reed and J. Reed, editors, *SDL2001 – Meeting UML*, volume 2078 of *Lecture Notes in Computer Science (LNCS).* Springer, June 2001.

[HMP91] T. Henzinger, Z. Manna, and A. Pnueli. Timed Transition Systems. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory and Practice*, volume 600 of *Lecture Notes in Computer Science (LNCS)*, pages 226–251. Springer, June 1991.

[Hog89] D. Hogrefe. *Estelle, Lotos und SDL.* Springer, 1989.

[Hog91] D. Hogrefe. OSI Formal Specification Case Study: The INRES Protocol and Service. Technical Report IAM-91-012, University of Berne, Institute for Informatics and Applied Mathematics, May 1991.

[IEE96] IEEE Standard 1003.1: Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]. Institute of Electrical and Electronics Engineers (IEEE), 1996.

[IET90] Request for Comments 1193: Client requirements for real-time communication services. Internet Engineering Task Force (IETF), 1990.

[IET91] Request for Comments 1242: Benchmarking Terminology for Network Interconnection Devices. Internet Engineering Task Force (IETF), July 1991.

[IET92] Request for Comments 1305: Network Time Protocol (Version 3) Specification, Implementation and Analysis. Internet Engineering Task Force (IETF), 1992.

[IET98] Request for Comments 2330: Framework for IP Performance Metrics. Internet Engineering Task Force (IETF), May 1998.

[IET99] Request for Comments 2679: A One-way Delay Metric for IPPM. Internet Engineering Task Force (IETF), September 1999.

[IET02] Request for Comments 3393: IP Packet Delay Variation Metric for IP Performance Metrics (IPPM). Internet Engineering Task Force (IETF), November 2002.

[Int02] Interval (Formal Design, Validation and Testing of Real-Time Telecommunications Systems). European Information Society Technologies Project IST-1999-11557. `http://www-interval.imag.fr/`, 2002.

[ISO97a] Information technology – Open Systems Interconnection – Basic Reference Model. ISO/IEC, 1989-1997. International ISO/IEC multipart standard No. 7498.

[ISO97b] Information Technology – Open Systems Interconnection – Conformance testing methodology and framework. ISO/IEC, 1994-1997. International ISO/IEC multipart standard No. 9646.

[ISO98] Programming language C++. ISO/IEC, 1998. International ISO/IEC standard No. 14882.

[ISO02] Information technology – Open Systems Interconnection – Network service definition. ISO/IEC, 2002. International ISO/IEC standard No. 8348.

[ITU99a] ITU-T Recommendation Z.100: Specification and Description Language (SDL). International Telecommunication Union (ITU-T), Geneve, 1999.

[ITU99b] ITU-T Recommendation Z.120: Message Sequence Charts (MSC). International Telecommunication Union (ITU-T), Geneve, 1999.

[ITU01] ITU-T Recommendation Z.120: Message Sequence Charts (MSC), Corrigendum 1. International Telecommunication Union (ITU-T), Geneve, 2001.

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley, 1991.

[KJG99] A. Kerbrat, T. Jéron, and R. Groz. Automated test generation from SDL specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 – The next Millenium.* Elsevier Science Publishers B.V., June 1999.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science (LNCS).* Springer, June 1997.

[Koc01] B. Koch. *Test-purpose-based Test Generation for Distributed Test Architectures.* PhD thesis, University of Lübeck (Germany), February 2001.

[Koy91] R. Koymans. (Real) Time: A Philosophical Perspective. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science (LNCS)*, pages 353–370. Springer, June 1991.

[LAK99] W. Lewandowski, J. Azoubib, and W. Klepczynski. GPS: A Primary Tool for Time Transfer. *Proceedings of the IEEE*, 87(1), January 1999.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21, 1978.

[Lam90] L. Lamport. Concurrent Reading and Writing of Clocks. *ACM Transactions on Computer Systems*, 8, 1990.

[lL90] G. le Lann. Critical issues for the development of distributed real-time computing systems. Technical Report 1274, Institut National de Recherche en Informatique et en Automatique (INRIA), Le Chesnay (France), August 1990.

[MFC01] T. Mackinnon, S. Freeman, and P. Craig. EndoTesting: Unit Testing with Mock Objects. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*, chapter 17. Addison-Wesley, 2001.

[Mye79] G. Myers. *The Art of Software Testing.* Wiley, 1979.

[NDG04] H. Neukirchen, Z.R. Dai, and J. Grabowski. Communication Patterns for Expressing Real-Time Requirements Using MSC and their Application to Testing. In R. Hierons, editor, *Testing of Communicating Systems*, volume 2978 of *Lecture Notes in Computer Science (LNCS).* Springer, May 2004.

[Neu00] H. Neukirchen. Corrections and extensions to Z.120, November 2000. Delayed Contribution No. 9 to ITU-T Study Group 10, Question 9.

[Neu02] H. Neukirchen. TTCN-3 Change Request No. 148. Submitted to European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), June 2002.

[Obj04] ObjectMentor. JUnit Test Patterns. `http://www.junit.org/news/article/patterns/index.htm`, 2004.

[OMG03a] UML 2.0 Infrastructure Final Adopted Specification (ptc/03-09-15). Object Management Group (OMG), September 2003.

[OMG03b] UML 2.0 Superstructure (ptc/03-08-02). Object Management Group (OMG), August 2003.

[OMG03c] UML Profile for Schedulability, Performance, and Time Specification, Version 1.0 (formal/03-09-01). Object Management Group (OMG), September 2003.

[OMG04a] Common Object Request Broker Architecture: Core Specification, Version 3.0.3 (formal/04-03-12). Object Management Group (OMG), March 2004.

[OMG04b] UML 2.0 Testing Profile Specification (ptc/04-04-02). Object Management Group (OMG), April 2004.

[Ope04] OpenTTCN Oy OpenTTCN Tester for TTCN-3 product information. `http://www.openttcn.com/Sections/Products/OpenTTCN3`, 2004.

[Ous04] J. Ousterhout. Tool Command Language (TCL). `http://tcl.sourceforge.net/`, 2004.

[Par91] H. Partsch. *Requirements Engineering*. Oldenbourg Verlag, 1991.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science (FOCS)*. IEEE Computer Society Press, 1977.

[PUT$^+$01] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L.G. Votta. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering*, 27(12):1134–1144, December 2001.

[RSB90] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-Tolerant Clock Synchronization in Distributed Systems. *IEEE Computer*, 23, 1990.

[Sch93] W. Schütz. *The Testability of Distributed Real-Time Systems*. Kluwer, 1993.

[Sch03] M. Schmitt. *Automatic Test Generation Based on Formal Specifications – Practical Procedures for Efficient State Space Exploration and Improved Representation of Test Cases*. PhD thesis, University of Göttingen (Germany), April 2003.

[SE04] R. Savoye and B. Elliston. DejaGnu. `http://www.gnu.org/software/dejagnu/`, 2004.

[SEG$^+$98] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink – Putting SDL-based test generation into practice. In A. Petrenko and N. Yevtuschenko, editors, *Testing of Communicating Systems*, volume 11. Kluwer, 1998.

[SPVG01] I. Schieferdecker, S. Pietsch, and T. Vassiliou-Gioles. Systematic Testing of Internet Protocols – First Experiences in Using TTCN-3 for SIP. In *Proceedings of the 5th IFIP Africom Conference on Communication Systems, Cape Town (South Africa)*, May 2001.

[SR90] J. Stankovic and K. Ramamritham. What is Predictability for Real-Time Systems? *Real-Time Systems*, 2(4):247–254, 1990. Kluwer.

[SR96] I. Schieferdecker and A. Rennoch. Formal Based Testing of ATM Signalling. In U. Herzog and H. Hermanns, editors, *Formale Beschreibungstechniken für verteilte Systeme*, Arbeitsberichte des Instituts für mathematische Maschinen und Datenverarbeitung (Informatik), Band 29, Nummer 9. Universität Erlangen-Nürnberg, May 1996.

[SS03] I. Schieferdecker and B. Stepien. Automated Testing of XML/-SOAP based Web Services. In *Proceedings of the 13th Fachkonferenz der Gesellschaft für Informatik (GI) Fachgruppe "Kommunikation in Verteilten Systemen" (KiVS), Leipzig (Germany)*, February 2003.

[SSR97] I. Schieferdecker, B. Stepien, and A. Rennoch. PerfTTCN, a TTCN Language Extension for Performace Testing. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, volume 10. Chapman & Hall, 1997.

[Sun04] Sun Microsystems. Java Technology. `http://java.sun.com/`, 2004.

[Sza02] J.Z. Szabó. Experiences of TTCN-3 Test Executor Development. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems – Application to Internet Technologies and Services*, volume 14. Kluwer, March 2002.

[Tel04] Telelogic Tau/Tester product information. `http://www.tautester.com/`, 2004.

[Tes04] Testing Technologies TT Tool Series product information. `http://www.testingtech.de/products/TTToolSeries.html`, 2004.

[UHPB03] A. Ulrich, H. Hallal, A. Petrenko, and S. Boroday. Verifying Trustworthiness Requirements in Distributed Systems with Formal Log-file Analysis. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*. IEEE, 2003.

[VGDS04] T. Vassiliou-Gioles, G. Din, and I. Schieferdecker. Execution of External Applications Using TTCN-3. In R. Hierons, editor, *Testing of Communicating Systems*, volume 2978 of *Lecture Notes in Computer Science (LNCS)*. Springer, May 2004.

[W3C04] Extensible Markup Language (XML) 1.0 (Third Edition). World Wide Web Consortium (W3C) Recommendation, February 2004.

[Wal01] E. Wallmüller. *Software-Qualitätssicherung in der Praxis*. Hanser, 2001.

[Wey86] E. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, 12(12), December 1986.

[Wey88] E. Weyuker. The Evaluation of Program-based Software Test Data Adequacy Criteria. *Communications of the ACM*, 31(6), June 1988.

[WG97] T. Walter and J. Grabowski. Real-Time TTCN for Testing Real-Time and Multimedia Systems. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, volume 10. Chapman & Hall, 1997.

[WG99] T. Walter and J. Grabowski. A Framework for the Specification of Test Cases for Real Time Distributed Systems. *Information and Software Technology*, 41:781–798, July 1999.

[Wik04] Cunningham & Cunningham WikiWikiWeb. Code Review Patterns. `http://c2.com/cgi/wiki?CodeReviewPatterns`, 2004.

[ZK02] T. Zheng and F. Khendek. An extension to MSC-2000 and its application. In *Proceedings of the 3rd SAM (SDL and MSC) Workshop – Telecommunications and beyond: The Broader Applicability of SDL and MSC*, volume 2599 of *Lecture Notes in Computer Science (LNCS)*. Springer, June 2002.

Curriculum Vitae

Helmut Wolfram Neukirchen

Persönliche Daten

Geburt	1971 in Krefeld
Staatsangehörigkeit	deutsch

Wissenschaftlicher Werdegang

1978-1982	Bischöfliche Maria-Montessori-Grundschule Krefeld
1982-1991	Bischöfliche Maria-Montessori-Gesamtschule Krefeld; Abschluss: Abitur
1992-1999	Studium der Informatik an der Rheinisch-Westfälischen Technischen Hochschule Aachen mit Vertiefungsgebiet Softwarekonstruktion; Abschluss: Diplom-Informatiker
2000-2003	Wissenschaftlicher Mitarbeiter am Institut für Telematik der Universität zu Lübeck
seit 2003	Wissenschaftlicher Mitarbeiter am Institut für Informatik der Georg-August-Universität zu Göttingen

---