

SIIM Technical Report

Communication Patterns for Expressing Real-Time Requirements

Using MSC and their
Application to Testing
by

HELMUT NEUKIRCHEN,
ZHEN RU DAI,
JENS GRABOWSKI

**Schriftenreihe der Institute für
Informatik/Mathematik**

Serie A

March 28, 2003



Universität zu Lübeck
Technisch-Naturwissenschaftliche Fakultät

Email: neukirch@informatik.mu-luebeck.de

Phone: +49-451-500-3721

Fax: +49-451-500-3722

Communication Patterns for Expressing Real-Time Requirements Using MSC and their Application to Testing

Helmut Neukirchen, Zhen Ru Dai, Jens Grabowski

March 28, 2003

Abstract

This paper presents real-time communication patterns (RTC-patterns) for capturing real-time requirements. RTC-patterns are formalized by using Message Sequence Charts (MSCs). The application of RTC-patterns to testing is shown by an example.

1 Introduction

The motivation for the work presented in this paper comes from our research on test specification and test generation for testing real-time requirements of communication systems. Especially, we investigate graphical specification methods that can be used in all phases of an integrated system development methodology and that allow an automated generation and implementation of test cases.

We use the *Testing and Test Control Notation* (TTCN-3) [5] as test implementation language and developed the associated real-time extension *TIMEDTTCN-3* [3] to support the test of real-time requirements. For graphical test specification, we apply the *Message Sequence Chart* (MSC¹) language [13]. The MSC-based specification of real-time test cases and generation of *TIMEDTTCN-3* code from MSC test specifications is explained in [4].

Even though it is possible to generate *TIMEDTTCN-3* code automatically for each MSC test description, we would like to facilitate and harmonize the use of *TIMEDTTCN-3* by providing a common set of test evaluation functions. This would make test results more comparable and avoid misinterpretations due to the use of different or erroneous evaluation functions. The key issue of this approach is the identification of commonly applicable evaluation functions for *TIMEDTTCN-3* test cases. 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, 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, we look for a mechanism to identify evaluation functions for the most common cases.

¹The term *MSC* is used for a diagram written in the MSC language and for the language itself.

Our idea is to use *real-time communication patterns* (RTC-patterns) for expressing real-time requirements and to provide evaluation functions for these patterns only. By using RTC-patterns during test design or by scanning test specifications for RTC-patterns, it is possible to use predefined evaluation functions in *TIMEDTTCN-3* test descriptions.

The idea of patterns is not new. *Software patterns* as described in [6, 2] focus on structural aspects of software design. Conventional software patterns are independent of an implementation language and described in a rather informal manner. Different from software patterns, *SDL patterns* [7] are tailored to the development of SDL [12] systems. SDL patterns can be characterized as formalized software patterns. They 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.

RTC-patterns are used to describe real-time requirements in form of time relations among communication operations at the interfaces of a communication system. We use MSC for the pattern description. The formality of MSC allows formalizing at least some parts of the pattern instantiation. Even though the application domain of testing communication systems motivates our work on RTC-patterns, we believe that such patterns are of general interest for system development. Therefore, we present RTC-patterns independent of the testing domain (Section 2) and explain afterwards their application to testing (Section 3).

2 MSC and Patterns

This section gives a short introduction into the subset of the MSC language, which is used in this paper, and presents MSC patterns for capturing real-time requirements.

2.1 MSC

Basically, an MSC describes the flow of *messages* between the *instances* of a communication system. For example, the MSC Referenced (Figure 1c) includes three instances, i.e., *System₁*, *System₂* and *PCO*, and specifies that message *m3* is sent from *System₁* to *System₂*.

The MSC language supports abstraction from and refinement of behavior by *decomposed instances* and *references*.

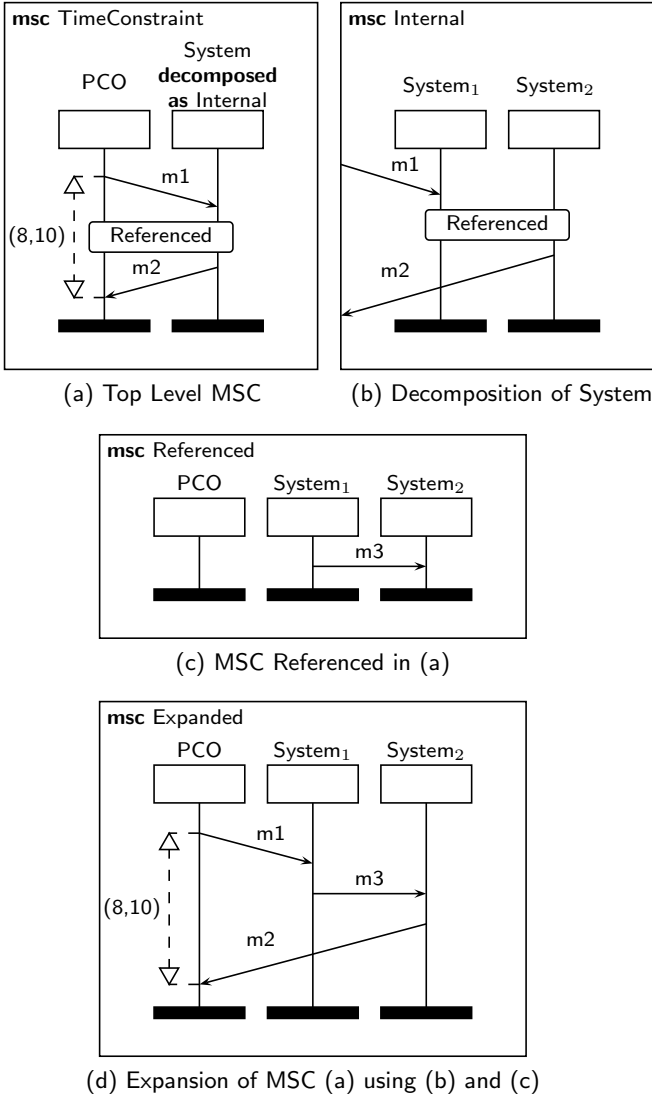


Figure 1: Used MSC Constructs

The decomposition mechanism allows to refine the behavior of an instance. This is shown in Figure 1a and 1b. The keywords **decomposed as** followed by the name *Internal* in the header of instance *System* (Figure 1a) indicates that *System* is an abstraction of the behavior specified by MSC *Internal* (Figure 1b). The MSCs in Figure 1a and 1b 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 behavior of the referenced MSC replaces the reference. By applying the rules for decomposed instances and references, the MSC *TimeConstraint* can be expanded to the MSC shown in Figure 1d.

For the specification of complex communication behavior in a compact manner within one diagram, MSC provides *inline expressions*. In this paper, we only use **loop** inline expressions to specify the repeated occurrence of events. Figure 5 presents an example, the behavior of the reference symbols `loopedPreamble`, `ResponseTimePattern` and `loopedPostamble` is repeated n times.

MSC allows to attach time annotations to events like sending or receiving a message. In this paper we make use of relative *time constraints* which limit the duration between two events. A time constraint is shown in Figure 1a: the time difference between sending `m1` and receiving `m2` at instance PCO is restricted to be between 8 and 10 seconds. The value of a time constraint is specified using intervals. The interval boundaries may be open, by using parenthesis, or closed, by using square bracket. An omitted lower bound is treated as *zero*, an omitted upper bound as *infinite*.

Time constraints can also be attached to the beginning and end of an inline expression (Figures 5 and 6). In this case, the constraint refers to the first or last event respectively which occurs inside the inline expression.

In addition to such relative time constraints, Figure 8 contains a time constraint for a cyclic event (sending message `m1` every \bar{t} seconds) inside a loop inline expression. The definition of such periodic events is not supported in the MSC standard. Therefore, we use an extension proposed in [14].

2.2 RTC-Patterns and MSC

In the following, MSCs are used to present RTC-patterns for the most common hard real-time requirements [1, 9, 10].² Since real-time requirements are always related to some functional behavior on which they are imposed, it is not possible to provide patterns for pure real-time requirements. Therefore, the RTC-patterns contain communication events on which the real-time requirements are imposed.

In order to ease specification and testing of real-time communication systems, it was our 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, we assume that the system for which the requirements are specified has appropriate interfaces called *points of control and observation (PCOs)*.

In our RTC-patterns, we represent each PCO as one MSC instance. The system is described by a single decomposed instance with the name *System*. We abstract from the internal structure of the system by omitting the reference to an MSC that refines the system behavior. Hence we obtain a black-box view of the system.

The most common real-time requirements are related to *delay*, *throughput*, *periodic events* and *jitter* respectively. Basically, those requirements describe time relations between one sending and one receiving event, or the repeated occurrence of one sending and one receiving 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 same real-time requirement in different system configurations. In this paper, we provide RTC-patterns for systems with one or two PCOs only.

²Note, that MSC is not well suited for expressing requirements involving statistical properties like soft real-time requirements or loss distributions.

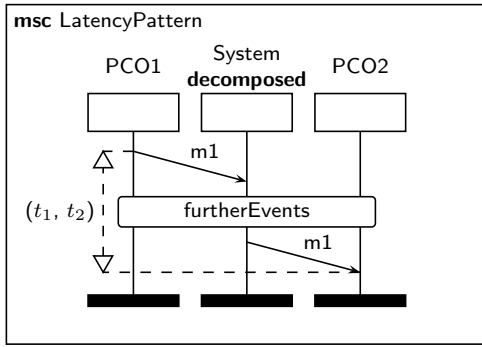


Figure 2: Latency Pattern

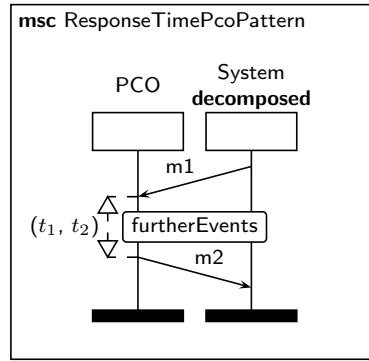


Figure 4: Response Time for the PCO Pattern

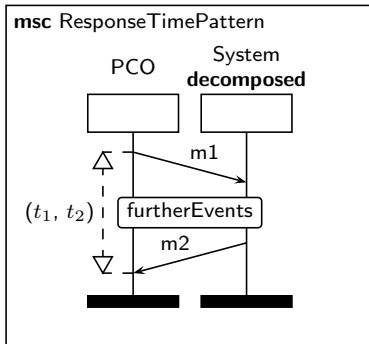


Figure 3: Response Time Pattern

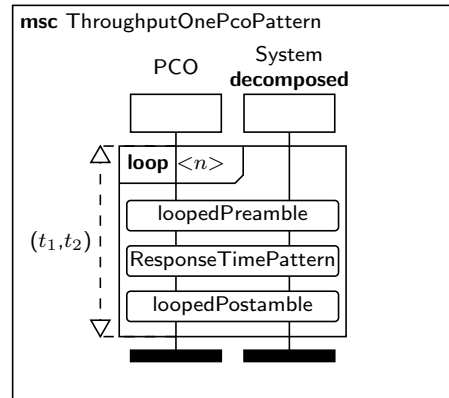


Figure 5: Throughput Pattern with One PCO

2.2.1 Delays: Latency/Response Time

The term *delay* is often used as an umbrella term for both *latency* and *response time* [9], since both only differ in the number of PCOs which are involved in the requirement. Hence, patterns for both types of real-time requirements are given.

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 [10]. The RTC-pattern for the latency requirement is given by the MSC `LatencyPattern` in Figure 2. The allowed latency between sending message `m1` via PCO1 and receiving it at PCO2 should 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`³, and additional computations inside the system (indicated by the `decomposed` keyword in the heading of the `System` instance).

Response time is a delay requirement where the same PCO is used for sending a message and receiving the corresponding answer. The *response time* pattern is shown in Figure 3. In contrast to the latency pattern, the messages in the response time pattern usually differ significantly, e.g., request (message `m1`) and response (message `m2`) in a client-server

³Even though in this pattern the same message name is used 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.

system. The given MSC shows a pattern for a response-time of t_1 and t_2 time units between sending message `m1` and receiving message `m2`.

The response time requirement can also be turned in an requirement or assumption for the system environment or tester. This is necessary, if a timely behavior of the environment is needed by the system to fulfil some other requirements. This requirement can be specified using the *response time PCO* pattern given in Figure 4.

2.2.2 Throughput

While delay-based real-time requirements focus on a systems performance for a single set of events, *throughput* requirements consider a systems performance over a longer duration. This means, the number of messages per time that a system has to deliver or to process repeatedly is constrained [9]. In MSC, this can be expressed using loop inline expressions with time constraints.

The *throughput one PCO* pattern shown in Figure 5 captures a throughput requirement for communication which is observed at one PCO.

The loop inline expression includes the references `loopedPreamble`, `ResponseTimePattern` and `loopedPostamble`. `ResponseTimePattern` refers to RTC-patterns *response time* (Figure 3) or *response time PCO* (Figure 4). The response time patterns define the functional behavior, which is part of the

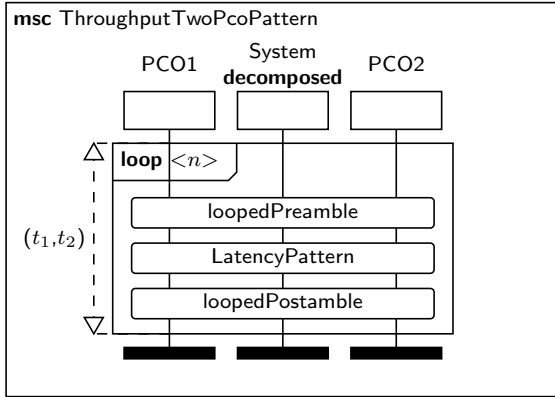


Figure 6: Throughput Pattern with Two PCOs

throughput requirement. Additional behavior, which precedes or follows the response pattern, may be contained in the MSC references `loopedPreamble` and `loopedPostamble`.

Even if a throughput requirement is fulfilled, this does not necessarily imply that all response requirements are fulfilled for each of the loop's iteration (e.g., due to bursty behavior and buffers inside the system). Thus, when inserting a response patterns into the throughput pattern, it has to be considered whether only the functional behavior of a response 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 pattern. The latter case leads to requirements for periodic events and their jitter (see next section).

The given throughput pattern constraints a throughput TP to be $\frac{n}{t_2} < TP < \frac{n}{t_1}$ events per time unit. Note that those "events" typically consist of a set of events, in particular to such according to one of the delay patterns presented before.

For specifying a throughput requirement, which is observed at two PCOs, the *throughput two PCO* pattern shown in Figure 6 is appropriate. This RTC-pattern re-uses the latency pattern (Figure 2) for describing the functional behavior, which is part of the throughput requirement.

2.2.3 Periodic Events and Jitter

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 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 this paper, we can only present some selected cases.

The first class of periodic requirements can be obtained, if delay patterns are put inside the loop. As an example, Figure 7 shows the expansion of a *repeated response time* pattern. The expressed real-time requirement is that the response time needs to hold every iteration of the loop.

Such MSCs can also be interpreted *delay jitter* specifications. Delay jitter describes the variation of the delay during

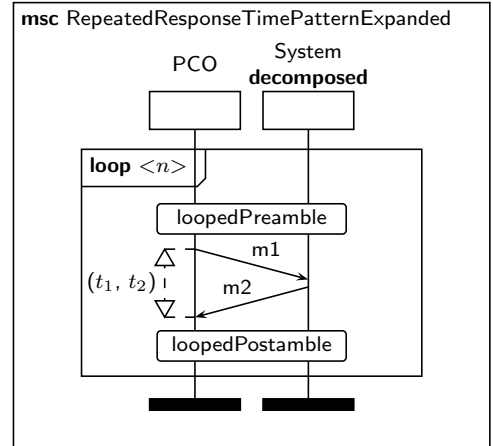


Figure 7: Expansion of a looped Response Time Pattern

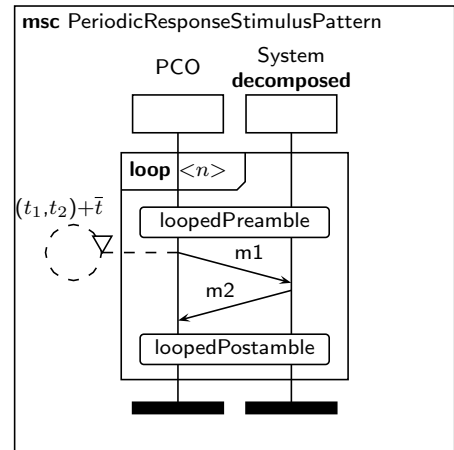


Figure 8: Periodic Response Stimulus Pattern

repetition. Note, that several interpretations of "jitter" exist [11]. Here, we use the following definition: $J_i = D_i - \bar{D}$, where \bar{D} is the ideal (target) delay, D_i the actual delay of the i^{th} pair of events and thus J_i the jitter in the i^{th} repetition. Hence, a delay 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 delay \bar{D} .

The RTC-pattern in Figure 7 expresses a target delay \bar{D} for which $t_1 < \bar{D} < t_2$ holds and a delay jitter requirement with $J^- = t_1 - \bar{D}$ and $J^+ = t_2 - \bar{D}$. I.e., the interval (t_1, t_2) could alternatively be written as $(\bar{D} + J^-, \bar{D} + J^+)$.

While time constraints for delays can be easily expressed using MSC, it is not possible to express the periodicity of cyclic events, i.e., a frequency. The reason is, that standard MSC does not allow to attach time constraints to a pair of events which spans over adjacent repetitions of a loop. Thus MSC extensions for either plain MSC [14] and high-level MSC [15] have been suggested. The extension for plain MSC is shown in Figure 8. The semantics of this extension can be obtained by unrolling that loop as shown in Figure 9.

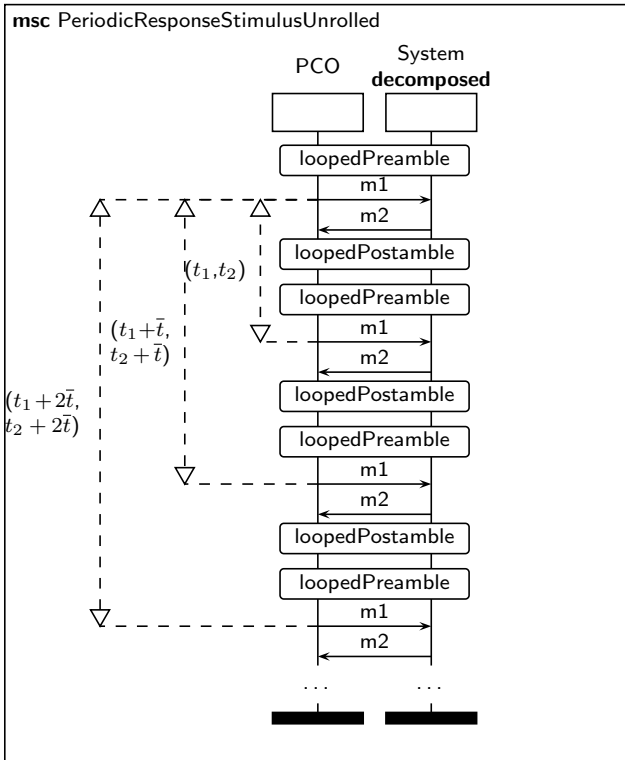


Figure 9: Pattern of Figure 8 with Unrolled Loop

The *periodic response stimulus* pattern in Figure 8 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. Likewise to delay jitter, a *jitter for the periodicity* or *frequency jitter* respectively is also specified by this pattern via $t1$ and $t2$, i.e. periodicity jitter requirement with $J^- = t1 - \bar{t}$ and $J^+ = t2 - \bar{t}$.

Further patterns can be obtained if two PCOs are used or the periodicity constraint is attached to another event, e.g., if the frequency of message $m2$ should be constrained.

3 Application to Testing

In the previous section, it was shown how MSC RTC-patterns can be applied for specifying real-time requirements. In this section, we demonstrate how the RTC-patterns can be used for test development.

3.1 Applying RTC-Patterns to *TIMEDTTCN-3*

TIMEDTTCN-3 [3] is a real-time extension for *TTCN-3*. It introduces the concept of absolute time, extends the *TTCN-3* logging mechanism, supports online and offline evaluation of tests and adds the new test verdict **conf** to the existing *TTCN-3* test verdicts.

This section does not introduce *TIMEDTTCN-3* language in detail. The presented *TIMEDTTCN-3* code is understand-

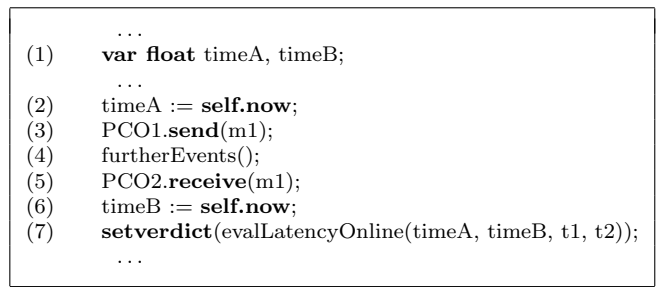


Figure 10: Code for Online Latency Evaluation

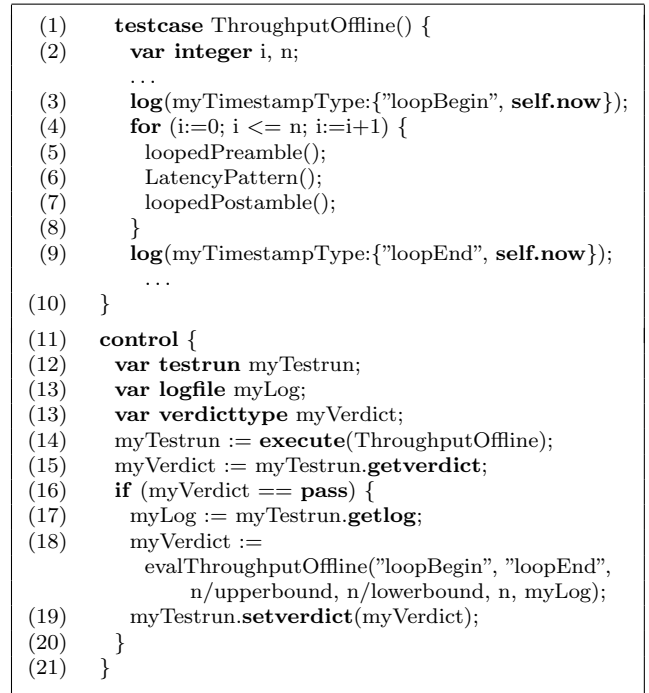


Figure 11: Code for Offline Throughput Evaluation

able for readers with some basic knowledge of common programming languages like, e.g., C or C++. Further details about *TTCN-3* and *TIMEDTTCN-3* can be found in [3] and [5].

TIMEDTTCN-3 distinguishes between two different evaluation mechanisms for real-time requirements. On the one hand, *online evaluation* refers to the evaluation of a real-time requirement during the test run. On the other hand, *offline evaluation* describes the evaluation of a real-time requirement after the test run. We explain both by presenting the online evaluation of a latency requirement and by describing the offline evaluation of a throughput requirement.

Figure 10 shows the *TIMEDTTCN-3* code fragment, which can be generated automatically for the latency RTC-pattern. The time critical events for measuring the latency of two events are the sending of message $m1$ and receiving of message $m1$ (Figure 2). Thus, before $m1$ is sent to the SUT (line 2 of Figure 10) and after $m1$ is received (line 6), the points in time are measured and stored in the variables `timeA` and `timeB`. The online evaluation func-

tion for latency is called in line 7 with the parameters of the measured time values, i.e., `timeA` and `timeB`, and the time-bounds stored in `t1` and `t2`.

The definition of the function `evalLatencyOnline` can be found in the lines 6-15 of Figure 12. Figure 12 is an excerpt of the library module `EvaluationFunctionModule`, which embodies all functions for real-time evaluations.

In Figure 10, function `evalLatencyOnline` is called in line 7 within a `setverdict` operation. Depending on the time measurement, the function returns a `pass` verdict, if the real-time requirement is met, or a `conf` verdict if the requirement is not met. The `setverdict` operation sets the verdict of the tester to the result of `evalLatencyOnline`.

The lines 1-10 in Figure 11 depict the code fragment for a test case developed with the *throughput two PCO* pattern (Figure 6) that uses the offline evaluation mechanism for the throughput requirement. The time-critical events for throughput are executed in a loop, which we consider as a black-box. Thus, only the time points immediately before and after the execution of the loop construct are of interest. In line 3 and line 9 of Figure 11, the points in time before and after the loop are logged in a logfile. Each entry of the logfile contains the name of the event and the time value, which is gained by the `self.now` statement.

In order to perform the offline evaluation, test case `ThroughputOffline` is invoked in the control part of the `TIMEDTTCN-3` module (line 14 of Figure 11). Afterwards, the verdict of the functional behavior is obtained (line 15). If the functional verdict is a `pass` verdict, the real-time requirement will be evaluated. For that, the logfile is retrieved (line 17) and the evaluation function `evalThroughputOffline` is triggered (line 18). The parameters of the function are the identifiers of the logfile entries, the upper and lower throughput bounds⁴, the number of iterations and the logfile generated by the test case.

The definition of function `evalThroughputOffline` can again be found in the library module `EvaluationFunctionModule` (lines 16-40 of Figure 12). The function has six parameters: the identifiers of the entry and exit points of the loop (`loopEntry`, `loopExit`), the lower and upper throughput bounds (`lowerThroughput`, `upperThroughput`), the number of iterations (`n`) and the logfile (`timelog`). Lines 19-33 are defined to select and get the entries with the time-critical events. The `first` operation sorts the logfile entries by their types and moves the focus to the first matching entry in the logfile. A `”?"` indicates the field that is used as a sorting key. The second parameter of the `first` operation is used to move the focus to the entry which owns the `loopEntry` (line 20). The logfile entry which matches to the request, is issued by the `retrieve` operation (line 21). The `next` operation gives out the following entry which owns the `loopExit` (line 23). The calculation of the throughput value is defined in lines 34-40. 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 not met.

⁴The throughput bounds are calculated from the number of iterations and the interval bounds.

```

(1)  module EvaluationFunctionModule() {
(2)      type record ThroughputTimestampType {
(3)          float logTime,
(4)          charstring id
(5)      };
(6)      function evalLatencyOnline(float sendSigTime,
(7)          float receiveSigTime, float lowerbound,
(8)          float upperbound) return verdicttype {
(9)          var float timeDiff;
(10)         timeDiff := receiveSigTime - sendSigTime;
(11)         if ((lowerbound <= timeDiff) and
(12)             (timeDiff <= upperbound)) {
(13)             return pass; // non-functional pass
(14)         }
(15)         else {
(16)             return conf; // non-functional fail
(17)         }
(18)     }
(19)     function evalThroughputOffline(charstring loopEntry,
(20)         charstring loopExit, float lowerThroughput,
(21)         float upperThroughput, integer n,
(22)         logfile timelog) return verdicttype {
(23)         var TimestampType stampA, stampB;
(24)         var float timeDiff;
(25)         if (timelog.first(TimestampType:{?,-},
(26)             TimestampType:{?, loopEntry}) == true) {
(27)             stampA := timelog.retrieve;
(28)             // Get current timestamp entry
(29)             if (timelog.next(TimestampType:
(30)                 {?, loopExit}) == true) {
(31)                 stampB := timelog.retrieve;
(32)                 // Get current timestamp entry
(33)             }
(34)             else {
(35)                 return fail; // Error while retrieving log
(36)             }
(37)         }
(38)         else {
(39)             return fail; // Error while retrieving log
(40)         }
(41)         timeDiff := stampB.logTime - stampA.logTime;
(42)         if ((lowerThroughput < n/timeDiff) and
(43)             (n/timeDiff < upperThroughput)) {
(44)             return pass; // non-functional pass
(45)         }
(46)         else {
(47)             return conf; // non-functional fail
(48)         }
(49)     }
(50) }

```

Figure 12: Module with Evaluation Functions

In Figure 11, the offline evaluation function is called in line 18. In line 19, the result of the function is used to set the final verdict of the test case.

3.2 The Inres Example

Figure 13 shows an MSC test purpose for testing the Initiator implementation of the Inres protocol [8] with real-time requirements. The Inres system can be accessed by using the PCOs ISAP and MSAP. The functional requirement of

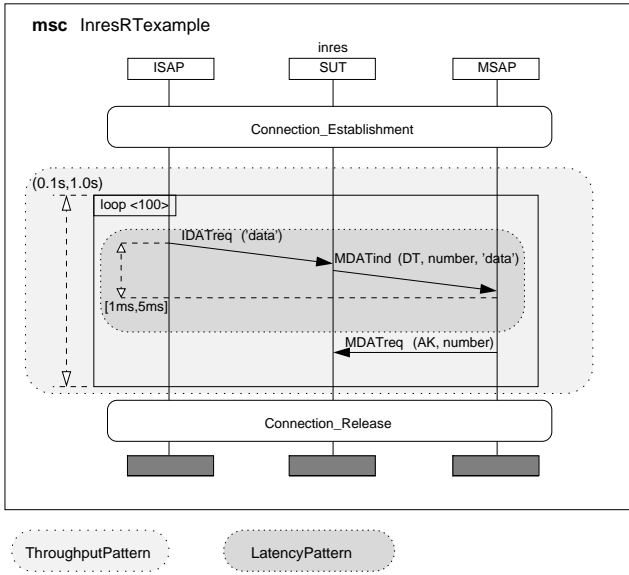


Figure 13: Test Purpose for the Inres Example

the test purpose is to test 100 data transfers. 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 between the signals IDATreq and MDATind, and
2. a throughput constraint of the loop construct.

When scanning through the given MSC diagram, the RTC-pattern for latency and throughput with two PCOs (Section 2.2) can be recognized. The shaded areas in Figure 13 show where both patterns are settled in the diagram.

For this example, we decided to evaluate the latency between IDISreq and MDATind during the test execution (i.e., online) and the throughput of the loop construct after the test execution (i.e., offline). The MSC diagram does not define which method of the evaluations is requested since the MSC language does not provide the possibility to express those kind of requirements. We consider such information as directives for a code generation algorithm.

In the previous section, we have introduced the *TIMEDTTCN-3* code fragments and evaluation functions for online latency and offline throughput requirements (Figures 10, 11 and 12). Now we shall utilize them in our example.

Figure 14 shows the *TIMEDTTCN-3* code for the Inres example, which is generated automatically from the MSC diagram in Figure 13. The module *InresRTexample_module* (Figure 14) imports all evaluation functions and types from the library *EvaluationFunctionModule* (line 2). It embodies only one test case called *InresRTexample* (lines 3-28) and the module control part (lines 29-42).

Test case *InresRTexample* starts with a connection establishment (line 7). After connection establishment, points in time for the throughput measurement are surveyed. Analogical to the throughput code fragment, the time value and the

```

(1)  module InresRTexample_module() {
(2)  import all from EvaluationFunctionModule;
(3)  testcase InresRTexample() runs on inres {
(4)    var integer i;
(5)    var float timeBegin, timeEnd;
(6)    var verdicttype myVerdict;
(7)    Connection_Establishment();
(8)    // throughput pattern scheme begin
(9)    log(ThroughputTimestampType:{self.now, "loopBegin"});
(10)   for (i:=0; i<100; i:=i+1) {
(11)     // latency pattern scheme begin
(12)     timeBegin := self.now;
(13)     ISAP.send(IDATreq:{"data"});
(14)     MSAP.receive(MDATind:{DT, number, "data"});
(15)     timeEnd := self.now;
(16)     myVerdict := evalLatencyOnline(timeBegin,
                                     timeEnd, 0.001, 0.005);
(17)     setverdict(myVerdict);
(18)     // online evaluation of latency.
(19)     // latency pattern scheme end;
(20)     MSAP.send(MDATreq:{AK, number});
(21)   }
(22)   log(ThroughputTimestampType:{self.now, "loopEnd"});
(23)   // offline evaluation of throughput in control part;
(24)   // throughput pattern scheme end.
(25)   Connection_Release();
(26)   setverdict(pass);
(27)   stop;
(28) }

(29) control {
(30)   var testrun myTestrun;
(31)   var logfile myLog;
(32)   var integer i;
(33)   var verdicttype myVerdict;
(34)   myTestrun := execute(InresRTexample);
(35)   myVerdict := myTestrun.getverdict;
(36)   if (myVerdict == pass) {
(37)     myLog := myTestrun.getlog;
(38)     myVerdict:=
(39)       evalThroughputOffline("loopBegin", "loopEnd",
(40)                             100/1.0, 100/0.1, 100, myLog);
(41)   }
(42) }

(43) }

```

Figure 14: Test Case Generated from Figure 13

event names are stored into the logfile before the loop construct is activated and after it is terminated (line 9 and line 22 in Figure 14). These logged informations are requested and accessed after the test run for the offline evaluation of the throughput requirement.

The online evaluation of latency between the signals IDATind and MDATreq is executed within the test case in lines 12-16. According to the code fragment, the time before and after the time-critical events IDATreq and MDATreq are stored in the variables *timeBegin* and *timeEnd*. The evaluation is performed during the test run and the verdict of the test case is set in line 17.

In the control part (lines 29-42), the throughput function is invoked. After the test case *InresRTexample* has been successfully executed regarding its functional behavior (lines 34-

36), the logfile is extracted (line 37) and the offline evaluation function `evalThroughputOffline` is called (line 38). The final verdict is set with respect to the outcome of the evaluation function (line 40).

4 Summary and Outlook

In this paper, MSC-based RTC-patterns for the specification of delay, throughput and periodic real-time requirements for communication systems have been presented. We demonstrated, how test development is eased, since pre-defined *TIMEDTTCN-3* evaluation functions can be associated to each RTC-pattern.

RTC-patterns may also improve the requirements definition and the specification phase of an integrated development methodology for real-time communication systems. For this, the formalisation of instantiation and composition of MSC-based RTC-patterns has to be studied. A formalization is possible due to the formality of MSC. Further investigations on the required MSC extensions, tool support and the usability of such an approach is necessary. Such investigations will be the focus of our future work.

Furthermore, we will implement support for RTC-patterns in our tool, which translates MSC test descriptions into *TIMEDTTCN-3* test cases. This includes also the provision of a library of generic evaluation functions for the RTC-patterns.

References

- [1] ATM Forum Performance Testing Specification (AF-TEST-TM-0131.000). The ATM Forum Technical Committee, 1999.
- [2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.
- [3] 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.
- [4] 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 15 of *Lecture Notes in Computer Science (LNCS)*. Springer, to appear in May 2003.
- [5] ETSI European Standard (ES) 201 873-1 (2002). 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 Rec. Z.140.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [7] B. Geppert. *The SDL Pattern Approach - A Reuse-Driven SDL Methodology for Designing Communication Software Systems*. PhD thesis, University of Kaiserslautern (Germany), July 2001.
- [8] D. Hogrefe. Report on the Validation of the Inres System. Technical Report IAM-95-007, Universität Bern, November 1995.
- [9] Request for Comments 1193: Client requirements for real-time communication services. Internet Engineering Task Force (IETF), 1990.
- [10] Request for Comments 1242: Benchmarking Terminology for Network Interconnection Devices. Internet Engineering Task Force (IETF), July 1991.
- [11] Request for Comments 3393: IP Packet Delay Variation Metric for IP Performance Metrics (IPPM). Internet Engineering Task Force (IETF), November 2002.
- [12] ITU-T Rec. Z.100 (1999). Specification and Description Language (SDL). International Telecommunication Union (ITU-T), Geneva.
- [13] ITU-T Rec. Z.120 (1999). Message Sequence Chart (MSC). International Telecommunication Union (ITU-T), Geneva.
- [14] H. Neukirchen. Corrections and extensions to Z.120, November 2000. Delayed Contribution No. 9 to ITU-T Study Group 10, Question 9.
- [15] T. Zheng and F. Khendek. An extension to MSC-2000 and its application. In *Proceedings of the 3rd SAM (SDL and MSC) Workshop*, 2002.