Expressing and Analyzing Timing Constraints in Message Sequence Chart Specifications

Hanêne Ben-Abdallah and **Stefan Leue**
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada

Technical Report 97-04

© Hanêne Ben-Abdallah and Stefan Leue, 1997
hanene|sleue@swen.uwaterloo.ca

April 1997

Abstract

Message Sequence Charts (MSCs) are increasingly supported in software engineering tools and methodologies for communication systems. The last Z.120 standard extends MSCs with operators to organize them in a compositional, hierarchical fashion to describe systems with non-trivial sizes. When dealing with timing constraints, the standard is still evolving along with several proposals. This paper first reviews proposed extensions of MSCs to describe timing constraints. Secondly, the paper describes an analysis technique for timing consistency in iterating and branching MSC specifications. The analysis extends efficient current techniques for timing analysis of MSCs with no loops or branchings. Finally, we use an example to illustrate our analysis technique.

Contents

1 Introduction **1**

2 Timing Constraints in Basic MSCs **2**

2.1 Syntax 2

2.2 Timing Analysis Based on Timers and Delay Intervals 5

2.3 Summary and Open Issues 5

3 Interpreting Timing Constraints in MSC Specifications **6**

3.1 Interpreting Iterations 6

3.2 Interpreting Branchings 8

4 Timed MSC Specifications **9**

4.1 Syntax 9

4.2 Semantics 10

5 Timing Analysis of MSC Specifications **11**

5.1 Timing Consistency of bMSCs 11

5.2 Timing Consistency of hMSCs 12

5.3 Process Divergence in Timed MSC Specifications 14

6 ATM Example **15**

7 Conclusion **18**

A Proofs **20**

List of Figures

1 MSC specification example: basic MSCs (left) and high-level MSC (right) 1
2 Timing constraints expressed through a Z.120 timer 3
3 (a) Event-associated Timing constraints; (b) Trace-associated Timing constraints . . 4
4 Timing constraints expressed through a Z.120 timer and delay intervals 4
5 Timing constraints in an iterating MSC specification 7
6 Timed MSC Specification 8
7 Timed basic MSC (left) and the corresponding temporal graph (right) 12
8 High-level MSC for the ATM example 15
9 Basic MSCs in the ATM example 16
10 Paths inside the temporal graph of $M_1 \bullet M_2 \bullet M_1 \bullet M_2$ 21

1 Introduction

Various flavours of Message Sequence Charts (MSCs) have been used in software engineering of telecommunications systems as well as object-oriented analysis and design notations. The graphical constructs in MSCs provide for an intuitive description of the interactions, protocols and services between a system's components. MSCs are used to document system requirements that guide the system design [17], describe test cases and scenarios [13, 7, 8], express system properties that are verified against SDL specifications [1], visualize sample behavior of a simulated system specification [17, 1], and to express legacy specifications in an intermediate representation that helps in software maintenance and reengineering [10].

Recently, the intuitive, graphical notation of MSCs increased their popularity within the software engineering community and motivated a standardization effort. The MSC standard as defined by the ITU-T in Recommendation Z.120 [12] introduces two basic concepts: *basic MSCs* (bMSCs) and *High-Level MSCs* (hMSCs). A bMSC consists of a set of processes that run in parallel and exchange messages in a one-to-one, asynchronous fashion. An hMSC graphically combines references to basic MSCs to describe parallel, sequential, iterating, and non-deterministic execution of basic MSCs. In addition, an hMSC can describe a system in a hierarchical fashion by combining hMSCs within an hMSC. We call the combination of a set of bMSCs and an hMSC describing their composition an *MSC specification*. MSC specifications describe potentially iterating and branching system behaviours. Figure 1 shows an example of MSC specification that describes a simple con-

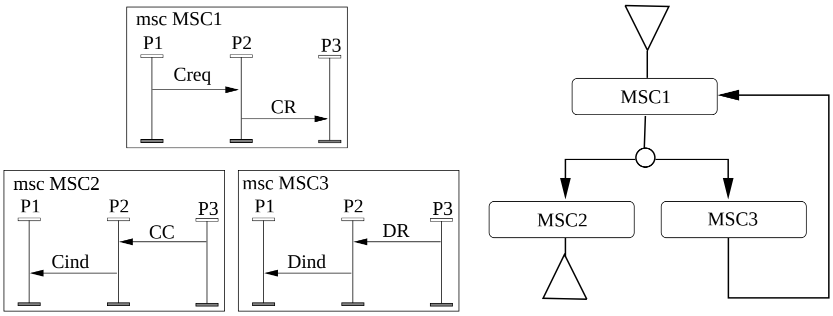


Figure 1: MSC specification example: basic MSCs (left) and high-level MSC (right)

nection establishment protocol in a telecommunication system. The basic MSC `MSC1` describes a connection request, the basic MSC `MSC2` describes the successful establishment of the connection, and the basic MSC `MSC3` describes an unsuccessful establishment of the connection. Process `P1` is a service provider, process `P2` is a local and process `P3` is a remote protocol machine. Within the hMSC, the iterating branch describes a repeated request to establish the connection. The non-iterating branch describes a successful connection establishment.

The semantics of an MSC essentially consists of sequences (or traces) of messages that are sent and received among the concurrent processes in the MSC [14]. The order of communication events (i.e. message sent or received) in a trace is deduced from the visual partial order determined by the flow of control within each process in the MSC along with a causal dependency between the events of sending and receiving a message.

To facilitate the specification of real-time systems, a few extensions to MSCs have been proposed to express timing constraints: timers [12], interval delays [2, 15] and timing markers [7, 6]. The proposed extensions evolved independently and differ in terms of their expressiveness and support for analysis. Further, all proposed analysis of MSCs with timing constraints have been so far limited to basic MSCs.

In an effort to help consolidate the best of the proposed timing extensions possibly within the standard, in this paper we first review the various proposed syntactic annotations of basic MSCs with timing constraints. For each of the proposed timing extensions, we highlight the syntactic features, expressiveness and limitations, and we discuss ambiguities that must be addressed when bMSCs are composed within an hMSC.

Another motivation for this paper is to extend the timing consistency analysis for bMSCs to deal with iterating and branching MSC specifications. The analysis technique we present has been implemented within our prototype toolset for requirements engineering based on MSCs [3]. In addition, in this paper we extend our syntactic analysis of the process divergence problem in MSC specifications [5, 4] in the presence of timing constraints. We use the example of an automatic teller machine system to illustrate our chosen timing extension and the presented timing analysis.

2 Timing Constraints in Basic MSCs

We focused on three issues when reviewing current work that deals with timing constraints in MSCs: how they extend MSCs to express timing constraints; how they interpret timed MSCs; and what kind of timing analysis they offer.

2.1 Syntax

There are essentially four classes of syntactic constructs to express timing constraints in MSCs and MSC reminiscient notations: 1) *timers* [12, 2], 2) *delay intervals* [2, 15], 3) *drawing rules* [7, 6], and 4) other *timing markers* [7, 6].

Timers. Recommendation Z.120 provides *timers* to express timing constraints in a basic MSC. Within a single process ('*instance*' in Z.120 terminology), a timer can be *set* to a value, *reset* to zero, and observed for *timeout*. A timer cannot be shared among concurrent processes in an MSC. Figure 2 shows an example of a basic MSC with timing constraints expressed through two timers. In this example, process `P3` first sets the timer `T3.1` say to five time units, it then sends message `e`, receives message `b`, and sends message `c` before `T3.1` times out. In other words, process `P3` must exchange its messages within at most five time units relative to the timer setting event. Process `P1` first sets the timer `T1.1` say to three time units, receives message `a`, then resets its timer. Since process `P1` does not see a timeout event, the *implicit* assumption here is that the timer does not expire before it is reset. As the example illustrates, a timer can be used to express either a *minimal* delay between two consecutive events in one process, or a *maximal* delay between two or more consecutive events in one process.

The process algebraic, standard semantics [11] treats a timer's setting, resetting, and timeout as special events. This semantics uses an untimed model, and therefore has no notion of time. In addition, it offers no timing analysis. The partial order semantics defined by Alur et al [2] treats

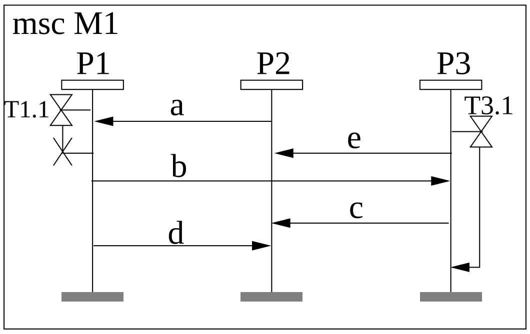


Figure 2: Timing constraints expressed through a Z.120 timer

timer events as "regular" events and provides *timing consistency* analysis. This work incorporates Z.120 timers with delay intervals; we therefore discuss its results shortly.

Delay Intervals. Besides the Z.120 timers, *delay intervals* have been proposed to express timing constraints in a basic MSC. Depending on how delay intervals annotate a bMSC, they express three types of timing constraints:

1. *Event-associated* timing constraints [15], which are denoted as an interval that is associated with an event in the basic MSC.

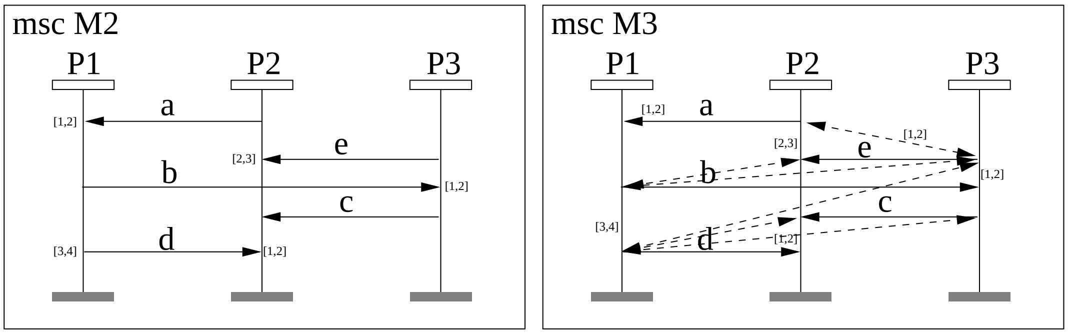
2. *Message delivery* delays [2, 15], which are expressed as a time interval over a message arrow.

3. *Processor's speed* constraints [2, 15], which are expressed as time intervals between two consecutive events in a process.

An event-associated timing constraint is a *global* constraint on the timed occurrence of an event: the event must occur within the specified minimal and maximal time delays with respect to *any* previous event, whenever it occurs in a trace. Figure 3 (a) shows sample event-associated timing constraints.

In the message delivery and processor's speed constraints, a delay interval is delimited with respect to the occurrence of the two consecutively, visually ordered events it constrains. Figure 4 augments the timing constraints in Figure 2 with message delays (i.e., intervals on message arrows) and processor's speed constraints (i.e., intervals on vertical lines). In this version, message `b` takes between two and three time units from the time it is sent by process `P1` to the time it is received by process `P3`. In addition, process `P3` requires that message `b` be received between one and two time units from the time it sends message `e`.

In [15], the author generalizes the message delivery and processor's delay intervals (called *trace-associated* timing constraints) by using a semantic notion of *consecutive* events: two events are consecutive if they can be executed one after the other. In addition, this work extends the use of trace-associated timing constraints to express timing constraints between events that are not related. For this, the syntax of bMSCs is extended with *precedence* edges that connect unrelated events. The user can then annotate the extended bMSC with timing constraints to impose on unrelated events. Figure 3 (b) shows sample trace-associated timing constraints where the precedence



(a) (b)

Figure 3: (a) Event-associated Timing constraints; (b) Trace-associated Timing constraints

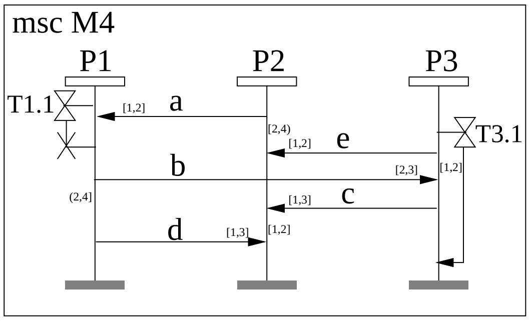


Figure 4: Timing constraints expressed through a Z.120 timer and delay intervals

edges are drawn with dashed-line, bidirected edges. As this example illustrates, while precedence edges allow the expression of more timing constraints, they may result in a cumbersome graph.

Drawing rules and timing markers. Sequence diagrams within the Unified Modeling Language [6] extend the Z.120 MSCs with additional information, e.g., focus of control to show the time when a process has a thread of control. Timing constraints are represented in a sequence diagram in two ways: the drawing rules of message arrows and *timing markers*. A horizontal message arrow indicates the *simultaneous* occurrence of the send and receive events of the message. A downward slanted message arrow, on the other hand, indicates a required delay between the send and receive events of the message. In addition, within each object outgoing message arrows can be drawn at a single point to indicate the simultaneous sending of a message. (Incoming message arrows are not allowed to meet at the same point within an object.)

To describe more quantitative timing constraints, timing markers are attached to a sequence diagram. Timing markers are boolean expressions placed in braces and attached to the diagram [6]. The boolean expressions can constrain particular events or the whole diagram. However, since neither the precise syntax of timing markers nor their formal semantics is defined, we cannot completely assess their expressiveness. In addition, no formal analysis of timing constraints has been proposed within the Unified Modeling Language.

2.2 Timing Analysis Based on Timers and Delay Intervals

Timing analysis consists of validating a timing assignment and verifying timing consistency. A timing assignment is essentially a time-stamp function that associates with the MSC events occurrence times with respect to a global clock. A timing assignment is valid if it respects the timing constraints in the MSC. A bMSC is timing consistent if there is at least one valid timing assignment that allows the MSC to have a behavior where the events occur according to the specified timing constraints.

For bMSCs extended with timers and timing delays, using the temporal constraint network techniques in [9], timing analysis reduces to computing all-pairs-shortest-paths in a labeled directed graph [15, 2]. In the worst case, this can be computed in $O(n^3)$ time where n is the number of events in the bMSC. We will discuss timing consistency analysis with this technique in detail in Section 5.

The MSC analyzer tool by Bell Labs [2] offers in addition to the above timing analysis for bMSCs, timed analysis based on a semantics that accounts for the queueing strategies in a bMSC:

1. *Visual conflict*, which occurs when two events e and f are visually such that e occurs before f, but every timing assignment makes f occur before e. This analysis is reduced to finding a path from e to f with a negative cost.

2. *Timing conflict*, which occurs when two events are *inferred* to occur in one order, but there is a timing assignment that violates the inferred timing delay between the two events. This analysis is reduced to finding the cost of the shortest path between the two events in question.

To analyze MSC specifications within this tool, the user would have to select the various bMSCs that compose one sequential path in the hMSC and analyze each path separately. However, in the presence of loops in the hMSC, this tool offers no hints on how many times the user is supposed to unfold a loop to conclude timing consistency of the loop. Further, analysis based on path selection should resolve certain issues about the usage of timer events and the interpretation of the timing constraints in the MSC specification.

2.3 Summary and Open Issues

- Z.120 suggests the use of timers added to basic MSCs in order to express either maximum delay constraints for a set of events, or minimum and maximum delays for consecutive events withing one process. Processor speed timing constraints as suggested in [2, 15] provide the same expressiveness for consecutive events. Unlike timers, time intervals cannot express delays between more than two consecutive events.

- Several timing properties are expressed in terms of delays between the sending and receiving a message. Since messages are sent and received by different, concurrent processes, message delivery delays can not be expressed via timers. Annotating message arrows with delay intervals as suggested in [2, 15] is an intuitive extension to the Z.120 basic MSCs. Timing interval-based analysis of basic MSCs has been proven practical in [2].

- To express timing constraints between events in one bMSC that are neither within the same process nor related by message arrwos, [15] suggests the use of time-annotated precedence edges. Timing analysis is feasible, but may involve potentially exponential model construction.
- To express more general timing constraints, e.g., to relate events within different basic MSCs, the current notation must be further extended. This can be done either by directly annotating the events within an MSC specification (e.g., through timer markers [7, 6]), or by augmenting an MSC specification with temporal predicates that describe the timing requirements. To provide an expressive notation, the first alternative could sacrifice the simple, graphical systax of bMSCs. On the other hand, the second alternative can result in a gap within a specification and would often require model-based analysis to examine the consistency of the timing constraints.

When basic MSCs are composed within an hMSC, serveral issues pertinent to timers must be addressed: 1) syntactic well-formedness of hMSCs when timer events are split across basic MSCs; 2) how to interprete timers inside iterations; and 3) how to determine timing consistency in the presence of branchings. We address these issues in the next sections.

3 Interpreting Timing Constraints in MSC Specifications

An MSC specification connects basic MSCs to describe sequential, possibly iterating and non-deterministic behavior. In the presence of timing constraints, iterations and non-determinism require a special attention for one essential reason: timing constraints can be spread across sequentially connected basic MSCs. We next illustrate how the Z.120 standard syntax [11] is ill-defined when timers are used in hMSCs, and outline possible choices of interpreting timing constraints in hMSCs. We assume that timing constraints are expressed through timers as suggested by the Z.120 standard syntax [11]. However, the arguments we present also hold when in addition delay intervals are used.

3.1 Interpreting Iterations

Current analyses of iterations in an hMSC rely on unfolding loops a finite number of times and analyze the resulting basic MSC. As we show in [5], in the case of untimed behavior, this technique misses anomalous behavior such as process divergence. In the case of timed behavior, this technique raises several questions about: 1) interpreting multiple occurrences of the set event of the same timer, 2) resolving the correspondence between several timers' set and timeout events, and overall 3) the syntactic well-formedness of basic MSCs with timers.

Consider the MSC specification of Figure 5 where the timer `T1.1` is set in the basic MSC `M1` and its timeout is detected in the basic MSC `M3`. As control iterates through the basic MSCs `M1` and `M2`, it is unclear whether the system generates a new instance of the timer, or uses the same

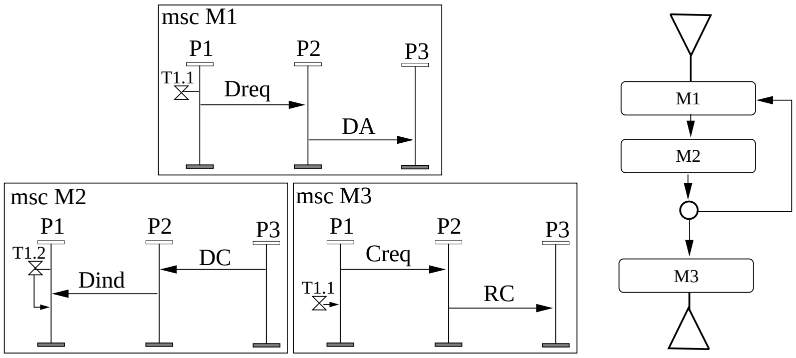


Figure 5: Timing constraints in an iterating MSC specification

timer during all iterations. Both interpretations can be justified by the common interpretation of a loop in an hMSC through a finite unfolding to a basic MSC.

More specifically, consider the following execution scenario of the MSC specification of Figure 5: `M1`, `M2`, `M1`, `M2`, and then `M3`. (Using loop unfolding, this sequence of basic MSCs represents a syntactically legal basic MSC.) When a loop is interpreted through a finite unfolding operation, it seems that a *new* timer is generated each time `M1` is executed. Therefore, we need to resolve the association of the timeout event in `M3` with the timer setting events. The choice affects the possible behavior of the MSC specification, i.e., its timing consistency analysis. Let us consider a uniform treatment of timers and events: The semantics in [14] intreprets events such that multiple sends of a message are deactivated by one receive of the message, based on an argument showing that MSC specifications are finite-state devices[1]. In the case of timers, this translates to associating the timeout event with *any* of the two timer setting events. A more reasonable choice is to associate it with the first timer since it would time out first. For `T1.1` set to 5 and `T1.2` set to 3, this intrepretation makes the above execution scenario timing inconsitent.

A second alternative is to associate the timeout event with the last timer set. This alternative coincides with the second interpretation where the same timer is reset then reused during all iterations. In this case, the same execution scenario for the example of Figure 5 is timing consistent in the sense that the last timer set does not expire before sending the event `Crq`; however, such an analysis is misleading when the overall behavior is considered: the first time the message `Dreq` was sent and sending the event `Crq` are separated by at least 6 time units and thus one timeout event, i.e., deadline was obviously missed.

The above ambiguity in interpreting timers within loops results from the ill-defined syntax of hMSCs when timers are involved. The Z.120 syntax of an hMSC assumes the well-formedness of the bMSCs used in the hMSC. The Z.120 syntax of bMSCs only restricts the usage of timers such that a reset or timeout event may occur only after a timer is set [11]; that is, this syntax neither forces the reset or timeout event to occur after a timer set event, nor does it restrict multiple occurrences of a timer set event prior to its reset or timeout event. As the above example illustrates, this relaxed syntax of bMSCs can lead to ambiguities when a loop in an hMSC contains bMSCs where one timer is set but neither its timeout nor reset event occurs in the loop.

[1]The standard Z.120 semantics [12] does not deal with iterations.

In a broader context, the above example raises a fundmental question about what an MSC specification means: does it describe *all* behaviors of a system, or does it describe a set of *sample* behaviors of a system? In the first case, the standard syntax must be further restricted to disallow the above example. In the second case, the above example should be allowed and interpreted according to the second alternative; that is, timers may expire without explicitly being modeled in the MSC specification. However, this interpretation may create practical difficulties since timers' expirations are usually implemented as interrupts and thus can not be ignored in some occasions and handled at other times.

3.2 Interpreting Branchings

An MSC specification can compose bMSCs in a non-deterministic fashion. This is described through nodes in its hMSC that have multiple successor nodes. Non-determinism in an MSC specification raises the following question: How can we determine whether the timing constraints in the MSC specification are satisfiable? This can be achieved in two ways:

1. *Local semantics:* select one path at a time and analyze its timing requirements, independently of other paths that may branch out of the selected path. This interpretation of timing constraints allows the derivation of several timing assignments, one for each path in the hMSC. In other words, any particular basic MSC that is shared by different paths may have different timed behavior depending on both the *past* and *future* behavior of the system. This approach has the advantage of producing possibly looser timing constraints. Current approaches which rely on interpreting bMSCs [2] adopt a local semantics for timing constraints.

2. *Global semantics:* all paths must be analyzed simultaneously. This analysis technique assumes that any timing assignment for the hMSC must be valid along all shared portions of all paths in the hMSC. In this approach, each basic MSC will have the same timed behavior independently of the execution path on which it resides, hence independently of the future behavior of the system. This approach produces tighter timing constraints. However, it may conclude the timing inconsistency of an hMSC when one path can in fact be timing consistent.

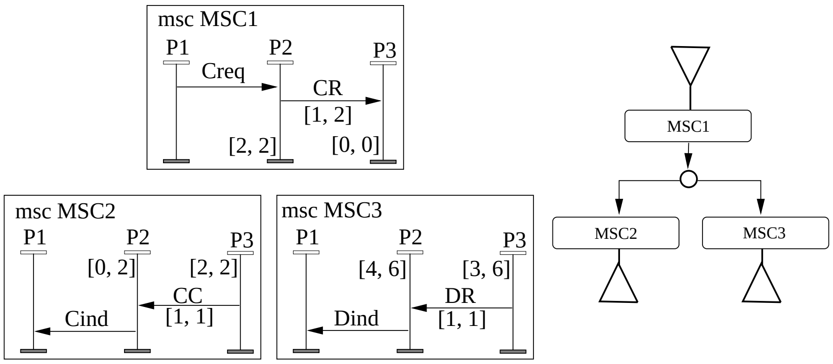


Figure 6: Timed MSC Specification

Example. To illustrate the differences between the two approaches, consider the timed, non-iterating variant of the MSC specification in Figure 1 shown in Figure 6. Table 1 shows two timed execution traces derived from the two possible paths in the hMSC taken independently. Thus, the MSC specification in Figure 1 is locally timing consistent. However, there is no timing assignment for the common prefix of execution traces I and II, i.e. <`!CR`, `?CR`>, that allows both traces to be continued in a timing consistent fashion: timing consistency of path I requires that `?CR` not to happen later than at time 1, whereas timing consistency of path II requires that `?CR` not to happen before time 2, relative to the time of occurrence of `!CR`. The MSC specification in Figure 6 is therefore not globally timing consistent.

Execution #	`!CR`	`?CR`	`!CC`	`?CC`	`!DR`	`?DR`
I	0	1	3	4	-	-
II	0	2	-	-	5	6

Table 1: Consistent timed executions

4 Timed MSC Specifications

Before we present our timing consistency analysis for MSC specification, we next define the syntax we use to describe timing constraints.

4.1 Syntax

In this paper, we use a subset of the Z.120 language for untimed bMSCs that comprises message exchanges only. To express timing constraints we use the Z.120 timer events together with (non-standard) timing delay intervals. A timer event can be: setting a timer, resetting a timer or a timeout. We assume that a timer can be set to a positive integer value, and rest to zero. In compliance with the Z.120 standard [11], a reset and timeout event must be preceeded by a timer setting event. In addition, a timer is private to a process and thus its events can be used by a single process only. Further, to distinguish between timers, we assume that each timer has a unique identifier associated with it.

A timing delay interval is a label over either a message arrow, or a control flow segment, i.e., a portion in a process's line that is delimited by two consecutive events. (We use the generic term *event* to denote one of the following types of events: the start of a process, the end of a process, sending a message, receiving a message, or a timer event.) A delay interval labelling a message arrow denotes the relative minimal and maximum delays between the events of sending the message and receiving it. A delay interval labelling a control flow segment denotes the relative minimal and maximal delays between the events delimiting the control flow segment. Delay intervals can be of the form $[a, b]$, $[a, b)$, or $(a, b]$ where $a \in \boldsymbol{N}$ and $b \in \boldsymbol{N} \cup \{\infty\}$. For an example of a timed bMSC according to this syntax see Figure 4.

An *MSC specification* is a structure $S = (B, V, suc, ref)$ where

- B is a finite set of bMSCs;

- $V = \top \cup I \cup \bot$ is a finite set of nodes partitioned into the three sets of: singleton-set of *start* node, *intermediate* nodes, and *end* nodes, respectively;
- $suc \subseteq (\top \cup I) \times V$ the relation which reflects the connectivity of the hMSC of S such that all nodes in V are reachable from the start node; and
- $ref : I \longmapsto B$ a function that maps each intermediate node to a bMSC in B[2].

A *path* in an MSC specification $S = (B, V, suc, ref)$ is a sequence of intermediate nodes (i.e., bMSCs), $b_1, b_2, \cdots$, such that $(b_i, b_{i+1}) \in suc$ for $i \geq 1$. A path is *simple* if all its nodes are distinct. A *loop* in S is a path $b_1, b_2, \cdots, b_n$ with $(b_n, b_1) \in suc$, and a loop is called *simple* if all its nodes are distinct.

In compliance with the Z.120 we allow timer events to be split across bMSCs in an hMSC. The Z.120 restriction that each timeout and timer resetting event must be preceeded by a timer setting event in bMSCs is extended to paths in the MSC specifications, i.e. in every path timeout and timer resetting events must be preceded by a timer setting. However, to avoid the ambiguities described in the previous section, we require that every simple loop in an MSC specification has *matched* timer events:

1. every timer setting event in the loop must be followed by either a timeout or reset event; and
2. every timeout event must be preceeded by a timer setting event in the loop.

The first restriction disallows the example in Figure 5. Note that this restriction is for loops only; that is, it does not force the use of a timeout or reset event in non-looping paths. As we see in the next section, our timing analysis will ensure that the absence of these events does not mean the specification is incomplete. The second restriction disallows the possibility that time ellapes in the loop making the timeout event obsolete.

4.2 Semantics

In accordance with earlier work [14], we interpret an MSC specification as the set of all execution traces that are consistent with the (visual) partial order of events specified in the bMSCs composing the MSC specification. That is, a *trace* σ of a bMSC M is a finite sequence of communication (i.e., send and receive message) and timer events in M such that the events in σ are ordered according to the visual ordering of events in M. To account for time, a *timed trace* of a bMSC M is a trace σ of M together with a *timing assignment* $T : E_\sigma \mapsto \boldsymbol{R}^+$ that assigns to each event e in σ a time-stamp $T(e)$ from the set of positive real numbers such that the timing constraints in M are met. Since events are partially ordered in M, and since there can be several possible timing assignments for a given trace of M, the behavior of M is the set of all its possible timed traces.

[2] We assume that an MSC specification contains one level of nesting; however, the definitions and results presented in this paper can be easily extended to deal with MSC specifications where nodes refer to *other* hMSCs.

5 Timing Analysis of MSC Specifications

In this section, we first augment the timing analysis for bMSCs presented in [15, 2] to handle the possibility that a timer is set in a bMSC but no reset nor timeout event follows the timer setting in the bMSC. We then extend it to to analyze MSC specifications with branchings and iterations.

5.1 Timing Consistency of bMSCs

To determine the timing consistency of a basic MSC, we adopt an approach similar to the ones presented in [15, 2]. First the bMSC is translated into a directed, labeled graph that we call *temporal constraint graph*. The vertices and edges in the graph reflect the control flow and message exchanges in the bMSC. The edge labels represent the timing constraints in the bMSC. Once a temporal constraint graph is constructed, to verify that the bMSC is timing consistent, we just check that the temporal constraint graph has no cycles with a negative cost [9, 15, 2].

Since it is unclear whether the informal translation presented in [2] handles the possibility that a timer is set but no reset nor timeout event is explicitly included in the bMSC, we next present the translation we assume in our analysis of timing consistency of MSC specifications.

From a bMSC to a temporal constraint graph. An edge in the temporal constraint graph is labeled with either the lower or upper bound of the delay interval imposed on the corresponding "edge" in the bMSC. To extract the bounds of a delay interval I with bounds $a, b \in \mathbb{N} \cup \{\infty\}$ $(a \leq b)$, we use the functions $\mathcal{L}(I)$ and $\mathcal{U}(I)$ defined as follows:

I	$[a,b]$	$(a,b]$	$[a,b)$	(a,b)
$\mathcal{L}(I)$	a	a^-	a	a^-
$\mathcal{U}(I)$	b	b	$\begin{cases} b^- & \text{if } b \neq \infty \\ \infty & \text{otherwise} \end{cases}$	$\begin{cases} b^- & \text{if } b \neq \infty \\ \infty & \text{otherwise} \end{cases}$

It is straightforward to represent a bMSC a directed, labeled graph (c.f. basic Message Flow Graph [14]). A node in the graph represents one of the following events: the start of a process, the end of a process sending a message, receiving a message, or setting, resetting or timeout a timer. An edge in the graph can have one of three types that represent the dependencies between events: 1) a "signal" edge (x, y) represents sending a message from one process to another; 2) a "next event" edge (x, y) represents the control flow within a process where event x appears before event y on the vertical line of the process; and 3) a "temporal" edge (x, y) connects a timer setting event x with a timer reset or timeout event y. The label of an edge is the timing delay and, for a signal edge, the message type. (For details, the reader is referred to [14] where basic MSCs without timing constraints are represented as Message Flow Graphs. This translation is easily augmented with the temporal edges to represent timing constraints.)

Given a bMSC M, its temporal graph $\mathcal{T}_g(M)$ is obtained as follows:

- each node in M is represented by a node in $\mathcal{T}_g(M)$;
- each next event edge, each signal edge and each temporal edge (e, e') with timing label I in M is represented by two labeled edges in $\mathcal{T}_g(M)$: 1) edge (e, e') with label $-\mathcal{L}(I)$; and 2) edge (e', e) with label $\mathcal{U}(I)$.

- for each process P_i in M, for each of its set timer event e_i with value t and no matching reset or timeout event, the following two edges are added in $\mathcal{T}_g(M)$: 1) (e_i, e'_i) with label $-t$; and 2) (e'_i, e) with label t, where the node e'_i is the node that corresponds to the bottom node of process P_i in M.

The above translation could differ from previous translations in the last step. As mentioned earlier, this additional step allows us to cover the lose Z.120 syntax [11] which does not force a set timer to have a reset or timeout event in the same bMSC. The analysis of the temporal constraint graph as constructed above ensures that those timers that were not explicitly reset or timeout, in fact, did not expire. Hence, the analysis results of the extended temporal constraint graph are coherent with the implicit assumption that a missing timeout/reset event in a process is interpreted as the set timer not having expired prior to the process's end of execution. Figure 7 illustrates the translation of a timed basic MSC into a temporal graph.

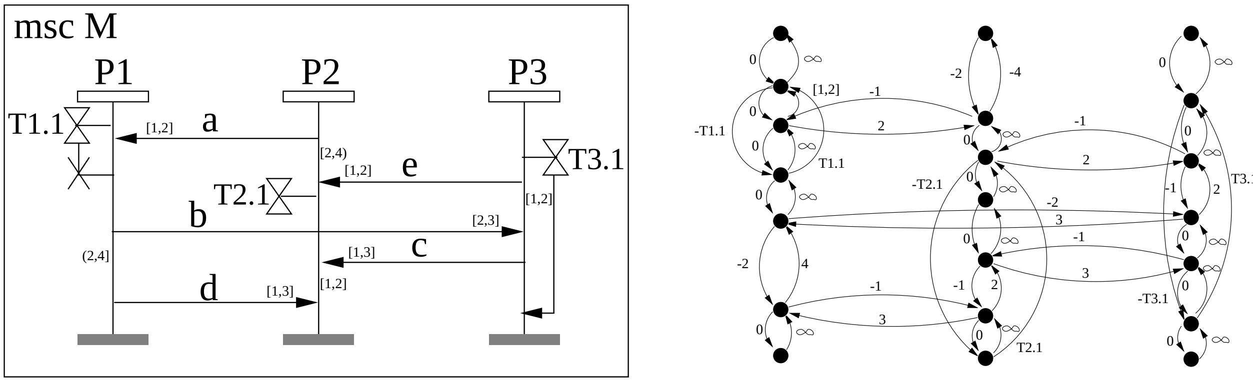


Figure 7: Timed basic MSC (left) and the corresponding temporal graph (right)

A bMSC is timing consistent if and only if its temporal constraint graph has no cycles with a negative cost [9, 15, 2][3]. Detecting cycles in the temporal constraint graph can be done through the Floyd-Warshall's algorithm which computes all-pairs-shortest paths in the graph [16] in a worst case time of $O(n^3)$ where n is the number of events in the graph. The timing consistency of the graph in Figure 7 depends on the choice of values for the timers denoted.

5.2 Timing Consistency of hMSCs

Our notion of timing consistency of an MSC specification relies on a local interpretation of the timing constraints.

Definition 5.1 An MSC specification S is *timing consistent* if *every* path that starts from the start node in S is timing consistent. S is *partially* timing consistent if *some* of its paths are timing consistent. S is *timing inconsistent* if *none* of its paths is timing consistent.

The above definition of timing consistency is impractical since in the presence of iterations in the MSC specification, the number of paths is infinite. We next present a syntactic approach to determine the consistency of an MSC specification based on a finite subset of its paths.

[3]Addition over the natural numbers $\boldsymbol{N}$ is extended over $\boldsymbol{N} \cup \boldsymbol{N}^- \cup \{\infty, -\infty\}$ in a straigfoward way.

In the sequel, we adopt the following notation to ease readability: given two bMSCS, M_1 and M_2, we denote the MSC specification that consists of the sequential composition of M_1 followed by M_2 as $M_1 \bullet M_2$.

Lemma 5.1 Given two basic MSCs M_1 and M_2, if $M_1 \bullet M_2$ is timing consistent then M_1 and M_2 are also timing consistent.

Proof: See Appendix A ∎

Lemma 5.2 If the MSC specification $M_1 \bullet M_2 \bullet M_1$ is timing consistent with $M_1 \bullet M_2$ having matched timer events, then the MSC specification $M_1 \bullet M_2 \bullet M_1 \bullet M_2$ is also timing consistent.

Proof: See Appendix A ∎

The above Lemma allows us to deduce the timing consistency of a loop from the timing consistency of an augmented version of the simple path the loop contains, without unfolding the loop. Thus, to decide the timing consistency of an MSC specification, we can focus on simple paths and augmented simple paths that represent loops in the specification. We next define these paths.

Definition 5.2 Let $S = (B, \top \cup I \cup \bot, suc, ref)$ be an MSC specification. A *sequential component* in S is a simple path $n_1, n_2, \cdots, n_k$ in S such that:

1. $(s, n_1) \in suc$ for the start node $s \in \top$; and
2. either $(n_k, e) \in suc$ for an end node $e \in \bot$, or there exists $n_j \in \{n_1, n_2, \cdots, n_k\}$ such that $(n_k, n_j) \in suc$.

Informally, a sequential component represents either a simple path from the start node to an end node, or a path that starts with a simple path and ends with a simple loop. We call the first type *finite* sequential components, and the second *infinite* sequential components.

Definition 5.3 Given an MSC specification S, we say that a path $n_1, n_2, \cdots, n_j, \cdots, n_k, n_j$ in S is a *closed* infinite sequential component if $n_1, n_2, \cdots, n_j, \cdots, n_k$ is an infinite sequential component in S.

Theorem 5.1 An MSC specification S is timing consistent if

1. each finite and each closed infinite sequential component in S is timing consistent; and
2. each infinite sequential component in S has matched timer events.

Proof: See Appendix A ∎

The condition on the infinite sequential components is stronger because the loop in the component allows time to progress an arbitrary amount, and thus possibly missing a timeout event. If a set timer is missing a reset or timeout event inside the loop, our analysis can not conclude from one iteration that the timer will never expire. In fact, if a timing consistent, infinite component has unmatched timer events, then the specification is partially timing consistent is the best we can predict.

Timing Consistency Algorithm

To examine the timing consistency of an MSC specification, we have implemented within our MSC tool [3] the following algorithm:

Input: an MSC specification S

Output: for each timing inconsistent sequential component in S, the events involved in a negative cost cycle

1. Find the finite and closed infinite sequential components in S
2. For each sequential component L:
3. construct the temporal constraint graph $\mathcal{T}_g(L)$
4. compute all-pairs-shortest paths in $\mathcal{T}_g(L)$
5. report all events involved in a negative cost cycle

End

Step 1 is carried out through a depth-first-search algorithm of the the hMSC of S. To construct the temporal constraint graph of a sequence of bMSCs, step 3 extends the bMSC to temporal graph translation of Section 5.1 in a straightforward way based on the following fact: the behavior of $M_1 \bullet M_2$ is equivalent to the behavior of the bMSCs obtained by glueing M_2 after M_1, with the timing delays at the end of a process in M_1 added to the timing delays of the same process at the beginning of M_2.

Step 4 uses the Floyd-Warshall's algorithm on a mtraix representation of the temporal constraint graph. In step 5 an event is in a cycle with a negative cost if its corresponding diagonal element in the all-pairs-shortest-path matrix is negative.

5.3 Process Divergence in Timed MSC Specifications

In the presence of loops, an MSC specification may suffer from process divergence: a system execution where a process sends messages an unbounded number of times ahead of the messages being received [5]. An MSC specification with a process divergence can be either unimplementable as it requires message queues with an infinite size, or it can be implementable with discrepencies, e.g., unexpected deadlocks since message queues are finite and messages must be dropped or overwritten.

In [5], we have syntactically characterized process divergence in untimed MSC specifications by examining the communication patterns of its processes. Informally, we proved that an (untimed) MSC specification suffers from process divergence if and only if it has a loop where at least one of its processes does not depend on, i.e., sends to but never receives directly or indirectly from another concurrent process in the loop.

In the presence of timing constraints through timers and/or delay intervals, our syntactic characterization of process divergence can be extended as follows.

Theorem 5.2 Given an MSC specification S that has untimed process divergence through the processes $P_1, \cdots, P_n$ in a loop L which can jointly race ahead of the remaining processes. S has timed process divergence iff either

1. the sum of all minimal delays in the processes $P_1, \cdots, P_n$ within L is equal to zero; or

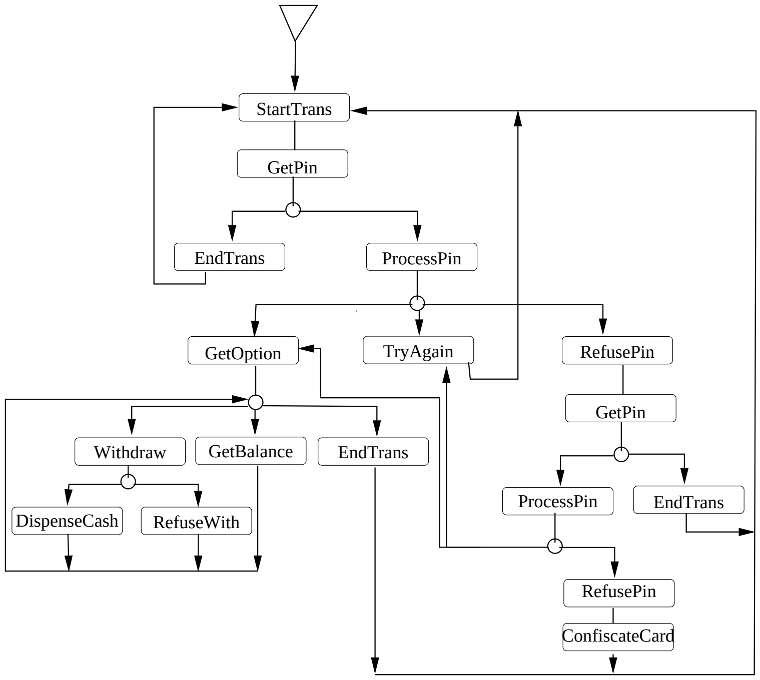


Figure 8: High-level MSC for the ATM example

2. the minimum of all maximum delays of the processes receiving messages from $P_1, \cdots, P_n$ including delays on the received messages is equal to ∞.

Proof: See Appendix A ▮

6 ATM Example

To illustrate our timing analysis, Figure 8 shows the hMSC and Figure 9 shows the referenced basic MSCs of an MSC specification describing an automatic teller machine (ATM) system. The ATM system consists of three components: potential customers that are represented by the `User` process, the ATM controller which is represented by the `ATM` process, and a host bank that is represented by the `Bank` process. Each one of the bMSCs represents a scenario or 'use-case' of the system, and the hMSC graph specifies a successor relationship between the scenarios.

Initially, the ATM controller waits to receive a message that signals a customer has inserted their bank card. Once this message is received, the system then behaves in two possible ways: either the ATM controller receives a request to cancel the transaction within $(0, 4)$ seconds (bMSC `EndTrans`), or the ATM receives the customer's pin number within $(5, 60)$ seconds (bMSC `ProcessPin`), relative to the time the message `Card` was received. Note that since these timing constraint relates two events in two basic MSCs, `GetPin` and either `EndTrans` in the first case or `ProcessPin` in the second case, we had to insert an artificial delay of $[0, 0]$ at the end of the `ATM` process in the bMSC `GetPin`.

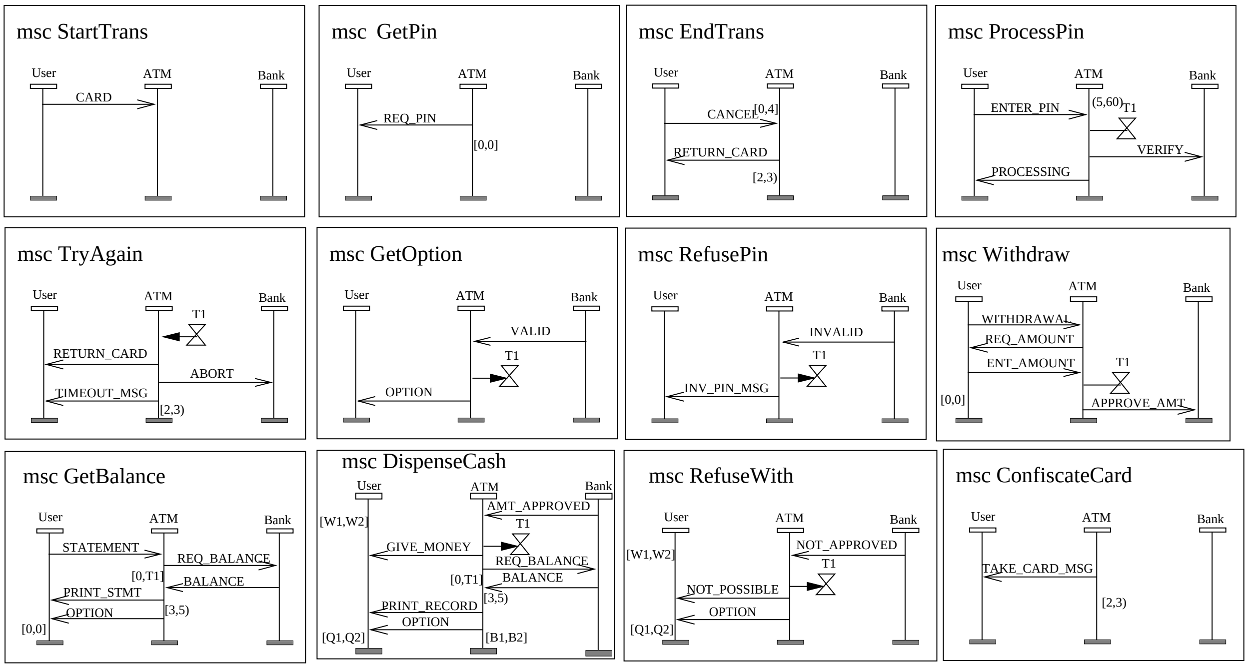


Figure 9: Basic MSCs in the ATM example

If the ATM receives a request to cancel the transaction (bMSC `EndTrans`), it returns the customer's card and takes between $[2,3)$ seconds to update its display after which the system returns to its initial state as described by the bMSC `StartTrans`. If the ATM receives the customer's pin number, it sends a request to the bank to validate the pin number, signals the customer to wait as it is processing the request, and then waits for a reply from the bank. For performance reasons, the ATM constrains communication with the bank to take no longer than T_1 seconds. In this case, the timing constraints are describe via the the timer T_1 in the bMSC `ProcessPin`. The next behavior of the system is described as follows:

- In the bMSC `TryAgain`, the ATM times-out on its wait for a validation reply from the bank. It therefore signals the customer to retry later, signals the bank to abort the validation request, returns the customer's card, and then takes between $[2,3)$ seconds to update its display. After this scenario, the system returns to its initial state.

- In the bMSC `RefusePin`, the ATM receives an invalid reply from the bank within the deadline. It resets the timer T_1, signals the customer that the entered pin is invalid, and tries to get a new pin. In this MSC specification, the customer tries to enter a valid pin at most twice, after which the ATM confiscates the customer's card– see the path from the bMSC `RefusePin` to `ConfiscateCard` in Figure 8.

- In the bMSC `GetOption`, the ATM receives a valid reply from the bank within the deadline. It therefore resets the timer T_1, and signals the user a list of options. From this point on, the customer can make one or more withdraws (bMSC `Withdraw`), request one or more time their account balance **bMSC GetBalance**), or end the transaction (bMSC **EndTrans**).

The customer expects a withdraw request to be processed (either successfully as in the bMSC **DispenseCash**, or unsuccessfully as in the bMSC **RefuseWith**) within $[W_1, W_2]$ seconds relative to the time they enter an amount. In addition, a customer takes $[Q_1, Q_2]$ seconds to decide whether to make another transaction while the ATM has their card. This is described by the combination of two delays: at the end the bMSCs **DispenseCash**, **RefuseWith** and **GetBalance**, the customer process delays the $[Q_1, Q_2]$ seconds; and at the beginning of the bMSCs **Withdraw** and **bMSC GetBalance**, the customer must not delay to start–the $[0, 0]$ before the first send message.

At the end of the bMSC **DispenseCash**, the ATM takes $[B_1, B_2]$ seconds to update its local records. In the bMSCs **DispenseCash** and **GetBalance**, the ATM takes $[3, 5)$ seconds to print a receipt after receiving the balance information from the bank.

In addition to the above timing constraints, we assume that each ATM-customer communication (and vice versa) has a delay of $[0, 2)$ seconds, and that each vertical line without a time delay has a default delay of $[0, \infty)$. For simplicity, these delays are not represented in Figure 9.

Timing Analysis

It is easy to check that our MSC specification of the ATM satisfies the syntactic conditions of Theorem 5.1. We used our analysis tool [3] to verify automatically the timing consistency of the specification for $T_1 = 10$ and various values of W_1, W_2, Q_1, Q_2, B_1, and B_2. Table 2 shows sample results. For the hMSC of Figure 8, the tool generated 43 infinite closed sequential components whose temporal graphs were then examined for cycles with a negative cost.

Table 2: Sample results of timing consistency analysis

Case #	(1)	(2)	(3)	(4)	(5)	(6)
$[W_1, W_2]$	$[0, \infty)$	$[0, 3]$	$[0, 4]$	$[0, 4]$	$[0, 4]$	$[0, 4]$
$[Q_1, Q_2]$	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	$[0, 2]$	$[0, 2]$	$[0, 1]$
$[B_1, B_2]$	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	$[5, 6]$	$[4, 6]$	$[4, 6]$
Consistent?	yes	no	yes	no	yes	no

For the case (1) in Table 2, the user does not impose any timing constraints on the system. This case in fact makes any value acceptable for the remaining variables. In the case (2), the user expects the ATM to process their withdraw request with $[0, 3]$ seconds; such a deadline leads to timing inconsistencies in all sequential components that contain the bMSC **Withdraw** followed by either **DispenseCash** or **RefuseWith**. In all of the timining inconsistent components, the tool detects the event send **ENT_AMOUNT** as being in a cycle with a negative cost of -1. The cost report gave us the hint to increase the upper-bound of the delay to 4 and run the verification again. As shown in the case (3), the specification becomes timing consistent. Note that for a value of $W_2 < 4$, the values of the remaining variables are irrelevant, since they affect paths that extend the problematic paths.

In the cases (4) and (5), we examine the affects of the book-keeping time $[B_1, B_2]$ the ATM requires after dispensing cash (bMSC **DispenseCash**). In this case, we only needed to vary the lower bound B_1 since all of the bMSCs composed after `DispenseCash` require a delay of $[0, \infty)$ before the first event that can be processes by the ATM; hence, for the ATM, the combined delays between the last event in `DispenseCash` and any next event is $[B_1, \infty)$. In the case (4), the timing inconsistent components all share the bMSC `DispenseCash` followed by either the bMSC `Withdraw` or `GetBalance`. Case (6) proves the dependency between the minimum book-keeping delay B_1 and the delays between consecutive customer requests while the ATM holds the customer's card, interval $[Q_1, Q_2]$. In the above cases, when a timing inconsistency is detected, the cost of the cycle and the involved events reported by the tool helped us to focus on which variables to adjust by which amount.

7 Conclusion

We have reviewed four proposed extensions of MSCs to express timing constraints and available analysis techniques for timing consistency of basic MSCs. The Z.120 standard timers together with delay intervals as suggested in [15, 2] can describe timing constraints for events within a process and events that are directly related, i.e., via the control edge or message arrow. From our experience, labeling all control edges and message arrows in a bMSC can however result in a cumbersome graph; a default assumption about the delays alleviates the problem. In addition, to express more general timing constraints, e.g., to relate events within different basic MSCs or processes, these extensions must be further augmented either by directly annotating the events within an MSC specification (e.g., through time markers [7, 6]), or by complementing an MSC specification with temporal predicates (e.g., boolean expressions [7, 6]) that describe the timing constraints. To provide an expressive notation, these alternatives could sacrifice the simple, graphical syntax of bMSCs and may result in more expensive analysis techniques.

Currently timing analysis is restricted to basic MSCs extended with timers and delay intervals. Following the analysis techniques of temporal constraint networks [9], timing consistency of a basic MSC is reduced to checking cycles with negative cost in a directed graph, which can be automated in $O(n^3)$ where n is the number of events in the basic MSC [15, 2]. To extend this analysis to MSC specifications with iterations and branchings, we highlighted syntactic issues that the Z.120 standard syntax must address. Based on specific syntactic recommendations, we then extended the analysis of timing consistency of bMSCs with timers and delays to the analysis of MSC specifications with branchings and iterations. To deal with branchings, we adopted a local interpretation of the timing constraints. To handle iterations, we showed that, under a reasonable assumption, a loop in the MSC specification can be analyzed by analyzing a simple extension of it, hence eliminating the need to unfold the loop to examine its timing consistency.

Acknowledgements. This work was supported by ObjecTime Limited and the Information Technology Research Centre (ITRC).

References

[1] B. Algayres, Y. Lejeune, F. Hugonment, and F. Hantz. The AVALON project: a validation environment for SDL/MSC descriptions. In O. Faergemand and A. Sarma, editors, *Proceedings*

of the 6th SDL Forum, SDL'93: Using Objects, October 1993.

[2] R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1055*, pages 35–48. Springer Verlag, 1996.

[3] H. Ben-Abdallah and S. Leue. Architecture of a requirements and design tool based on message sequence charts. Technical Report 96-13, Department of Electrical & Computer Engineering, University of Waterloo, October 1996. 18 p.

[4] H. Ben-Abdallah and S. Leue. Syntactic analysis of Message Sequence Chart specifications. Tech Report 96-12, Department of Electrical and Computer Engineering, University of Waterloo, November 1996.

[5] H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1217*, pages 259–274. Springer Verlag, 1997.

[6] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language for Object-Oriented Development (Version 0.91 Addendum).* RATIONAL Software Corporation, September 1996.

[7] G. Booch and J. Rumbaugh. *The Unified Method: User Guide Version 0.8.* RATIONAL Software Corporation, October 1995.

[8] R.J.A. Buhr and C.S. Casselman. *Use Case Maps for Object-Oriented Systems.* Prentice Hall, 1996.

[9] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[10] H. Ichikawa, M. Itoh, J. Kato, A. Takura, and M. Shibasaki. SDE: Incremental specification and development of communications software. *IEEE Transactions on Computers*, 40(4):553–561, Apr. 1991.

[11] ITU-T. Recommendation Z.120, Annex B: Algebraic Semantics of Message Sequence Charts. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, 1995.

[12] ITU-T. Recommendation Z.120. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, May 1996. Review Draft Version.

[13] I. Jacobson and et al. *Object-Oriented Software Engineering - A Use-case Driven Approach.* Addison-Wesley, 1992.

[14] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[15] N. Meng-Siew. Reasoning with timing constraints in Message Sequence Charts. Master's thesis, University of Stirling, Scotland, U.K., August 1993.

[16] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[17] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling.* John Wiley & Sons, Inc., 1994.

A Proofs

Proof of Lemma 5.1. We prove that if the sequential composition of two basic MSCs is timing consistent, then each basic MSC is also timing consistent. Let the MSC specification $S = M_1 \bullet M_2$ be timing consistent. We show that M_1 and M_2 are also timing consistent.

Recall that the semantics of the sequential composition of M_1 followed by M_2 is such that the behavior is the same as glueing the two basic MSCs into one basic MSC. Thus, the temporal graph of S, $\mathcal{T}_g(S)$ is obtained from $\mathcal{T}_g(M_1)$ and $\mathcal{T}_g(M_2)$ such that:

- all nodes except for the bottom nodes of $\mathcal{T}_g(M_1)$, and all nodes except for the top nodes of $\mathcal{T}_g(M_2)$ are in $\mathcal{T}_g(S)$;
- the bottom nodes of $\mathcal{T}_g(M_2)$ are the bottom nodes of $\mathcal{T}_g(S)$;
- the additional edges to match timer events in M_1 either have target nodes from $\mathcal{T}_g(M_2)$ (when the timers are matched in M_2), or now have target nodes as the the bottom nodes of $\mathcal{T}_g(M_2)$.

In other words, there is a corresponding from the nodes and edges of $\mathcal{T}_g(S)$ to the edges and nodes of $\mathcal{T}_g(M_1)$ and $\mathcal{T}_g(M_2)$. Now, assume that M_1 is not timing consistent. Then, $\mathcal{T}_g(M_1)$ has a cycle with a negative cost. This can result from two cases:

1. the edges in the cycle do not include the additional edges to match timers in M_1. This implies that the same cycle exists in $\mathcal{T}_g(S)$;
2. the edges in the cycle include the additional edges to match timers in M_1. Assume the cycle has the edge (n_{ik}, n_{il}) which has been added to match a time in process P_i of M_1. If the timer has not been matched in M_2, then then same edges are present in $\mathcal{T}_g(S)$ connected to the bottom nodes of $\mathcal{T}_g(S)$.

   If the timer has been matched in M_2, then the same edges are present in $\mathcal{T}_g(S)$ connected to some intermediate nodes in $\mathcal{T}_g(S)$ that correspond to nodes in $\mathcal{T}_g(M_2)$. Using graph isomorphism in both cases we can conclude that an isomorphic cycle is present in $\mathcal{T}_g(S)$;

Each of the above two cases implies that $\mathcal{T}_g(S)$ has a cycle with a negative cost; that is, S is timing inconsitent: a contradiction. The case M_2 is timing inconsistent is proven to lead to a contradiction in a similar way.

Proof of Lemma 5.2. Let the MSC specification $S = M_1 \bullet M_2 \bullet M_1$ be timing consistent and assume that $M_1 \bullet M_2$ has matched timer events. We show that the MSC specification $S' = M_1 \bullet M_2 \bullet M_1 \bullet M_2$ is also timing consistent. Assume that $M_1 \bullet M_2 \bullet M_1 \bullet M_2$ is not timing consistent. Thus, its temporal graph has a cycle with a negative cost. We distinguish four different cases how this negative cost cycle can be added, and show a contradiction to the assuption that S is timing consistent.

Case 1: The cycle involves only nodes that correspond to events in the second occurrence of M_2. This cycle will have an isomorphic cycle in the temporal graph portion that corresponds to the first occurrence of M_1. Using Lemma 5.1 and the timing consistency of S, this leads to a contradiction.

Case 2: The cycle involves only nodes that correspond to events in the second occurrence of M_1 and M_2. Using reasoning similar to Case 1, this leads to a contradiction.

Case 3: The cycle involves nodes that correspond to events from all of the basic MSCs. Let the cycle be

$$r, \cdots, x, v, \cdots, w, y, \cdots, s, \cdots, r$$

where "..." means possibly passing through other nodes and the specified nodes are defined as follows (see Figure 10):

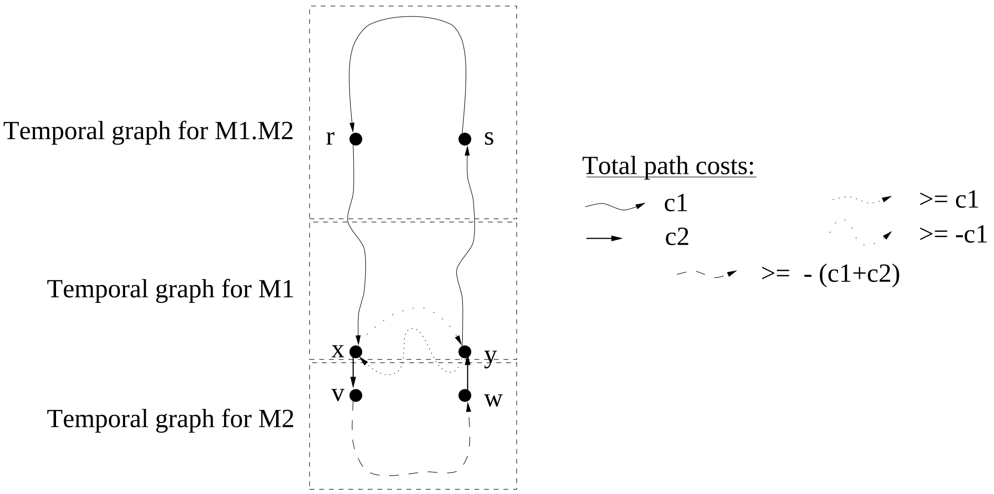


Figure 10: Paths inside the temporal graph of $M_1 \bullet M_2 \bullet M_1 \bullet M_2$

- r and s two nodes that correspond to events from the first occurrence of M_1 and/or M_2;
- x and y two nodes that correspond to the end events of the second occurrence of M_1;
- v and w two nodes that correspond to events of the second occurrence of M_1. In the corresponding temporal graphs they are immediate successors of x and y, respecitvely.

Now, let c_1 be the cost of the path (in the cycle) from y to x via the parts corresponding to M_1, M_2 and M_1. (This path is marked with a solid line in Figure 10.) Let c_2 be the accumulated cost of the two edges (in the cycle) from x to v and from w to y. (These segments are marked with thick line in Figure 10.) Thus, we have the path (in the cycle)

$$w, y, \cdots, s, \cdots, r, \cdots, x, v$$

which has a total cost of $c_1 + c_2$.

Since $M_1 \bullet M_2 \bullet M_1$ is timing consistent, then we have *every* path from x to y inside the temporal graph of $M_1 \bullet M_2 \bullet M_1$ has a cost greater than or equal to $-c_1$. (Otherwise, we get a cycle from r, x, y, s back to r with a negative cost.) This also implies that any path from y to x within the second occurrence of M_2 has a cost greater than or equal to c_1. (Otherwise, we get a cycle with a negative cost within the second occurrence of M_2 which, by graph isomorphism, makes $M_1 \bullet M_2 \bullet M_1$ timing inconsistent.)

Now, since $M_1 \bullet M_2$ is timing consistent (by Lemma 5.1), we can therefore use graph isomorphism to infer that the cost of *any* path from v to w within the second occurrence of M_2 must have a cost greater than or equal to $-(c_1 + c_2)$.

Thus the cost of the cycle from r back to r is the sum of the costs of the following paths: $y, \cdots, s, \cdots, r, \cdots, x$ followed by x, v followed by $v, \cdots, w$, and then w, y. This is greater than or equal to

$$(c_1 + c_2) \; + \; -(c_1 + c_2) = 0$$

which is a contradiction. ∎

Proof of Theorem 5.1. Let $S = (B, \top \cup I \cup \bot, suc, ref)$ be a timing inconsistent MSC specification that satisfies conditions 1 and 2 of the Theorem. By Definition 5.1, this implies that there is a path $n_1, n_2, \cdots, n_k$ in S that starts from the start node and is timing inconsistent.

Case 1: The timing inconsistent path is simple. It is straightforward to prove that there is a sequential component in S that extends the path and which is timing inconsistent. This contradicts the assumption from condition 1 of the Theorem.

Case 2: The timing inconsistent path is not simple, i.e., it is obtained through loops in S. Without loss of generality, assume that the path is obtained through one loop. That is, the timing inconstent path is of the form:

$$n_1, n_2, \cdots, (n_i, n_{i+1}, \cdots, n_j)^w, n_{j+1}, \cdots, n_k$$

for an integer constant $w \geq 2$, $(n_j, n_i) \in suc$ and distinct n's. This can be the result of the following three cases.

Case 2.1: The path $n_1, n_2, \cdots, (n_i, n_{i+1}, \cdots, n_j)^w$ is timing inconsistent for $w \geq 2$ and distinct n's. We use induction on w to generalize Lemma 5.2 and a reasoning similar to the proof of Lemma 5.2 to conclude that the simple path $n_1, n_2, \cdots, (n_i, n_{i+1}, \cdots, n_j)$ is timing inconsistent, which contradicts the Theorem hypothesis that $n_1, n_2, \cdots, (n_i, n_{i+1}, \cdots, n_j)n_i$ is timing consistent.

Case 2.2: The path $n_1, n_2, \cdots, (n_i, n_{i+1}, \cdots, n_j)^w$ is timing consistent for a constant $w \geq 2$, but the path $n_1, n_2, \cdots, (n_i, n_{i+1}, \cdots, n_j)^w n_{i+1}, \cdots, n_k$ is timing inconsistent. Thus, there is a cycle in the corresponding temporal graph with a negative cost. This can be the result of two cases:

Case 2.2.a: The cycle involves only nodes that correspond to events in the wth occurrence of $n_i, n_{i+1}, \cdots, n_j$ and $n_{i+1}, \cdots, n_k$. Using graph isomorphism (and the assumption that the timer events are matched in the loop), this implies that an isomorphic cycle exists in the temporal graph of the simple path $n_1, n_2, \cdots, (n_i, n_{i+1}, \cdots, n_j)n_{i+1}, \cdots, n_k$; this reduces to Case 1 where we also get a contradiction.

Case 2.2.b: The cycle involves nodes that correspond to events from more than just the wth occurrence of $n_i, n_{i+1}, \cdots, n_j$ and $n_{i+1}, \cdots, n_k$. In this case, we use a reasoning similar to the proof of Lemma 5.2, to show that the cycle actually must have a positive cost. (In this case, for Figure 10, the nodes r and s are anywhere in the portion of the graph that corresponds to $n_1, n_2, \cdots, (n_i, n_{i+1}, \cdots, n_j)^{w-1}$; the nodes x and y are the last nodes from the wth occurrence of $n_i, n_{i+1}, \cdots, n_j$; and the nodes v and w are the first nodes from $n_{j+1}, \cdots, n_k$.)

The above case analysis can be carried out by induction on the number of loops in the path to prove that in each case the path is in fact timing consistent. ∎

Proof of Theorem 5.2. Let S be an MSC specification with untimed process divergence through the processes $P_1, \cdots, P_n$ in a loop L which can jointly race ahead of the remaining processes.

If part: For the case 1., it is clear that the behavior of S via the loop L is *zeno*; that is, all events within the processes $P_1, \cdots, P_n$ can be executed infinitely many times within zero time units. In particular, those events sent from $P_1, \cdots, P_n$ to other processes can be executed an unbounded number of times in zero time units. Also, in the untimed model these events can be sent an unbounded number of times ahead of their corresponding receive events, a behavior that is also acceptable in the timed model. Thus, S also has process divergence in the timed model.
For the case 2., we have the timed behavior of the processes receiving messages from $P_1, \cdots, P_n$ can be delayed an ∞ time units. Thus, the untimed behavior where the processes $P_1, \cdots, P_n$ send an unbounded number of times messages without being received can be time-stampped to a timed behavioral for S, without violating any timing constraints in the processes which receive events from $P_1, \cdots, P_n$. This establishes that S has a process divergence in the timed model.

Only part: Assume that in the untimed model, the processes $P_1, \cdots, P_n$ within a loop L in S are involved in a process divergence. Thus, there is an infinite (untimed) execution of S such that these processes send messages an unbounded number of times ahead of the reception of these messages; let this sequence be $ss_1s_1s_1\cdots$.

Now, assume that neither case 1 nor 2 of the Theorem holds. Thus, let the sum of of the minimal delays within $P_1, \cdots, P_n$ be $min \neq 0$ and let the minimal of the maxmimum delays within the processes receiving messages from $P_1, \cdots, P_n$ including messages received from these latter be $max < \infty$.

For the execution $ss_1s_1s_1\cdots$ to be acceptable in the timed model, we must have a timing assignment that satisfies the timing constraints within: 1) the processes $P_1, \cdots, P_n$, and 2) those processes which receive events from $P_1, \cdots, P_n$. Assume there is such a timing assignment. Thus, the timing assignment for the divergent behavior $ss_1s_1s_1\cdots$ must be such that the duration of the infinite string $s_1s_1s_1\cdots$ is strictly less than $max < \infty$. Since the string is infinite, this implies that the duration of $s_1s_1s_1\cdots$ is zero, which contradicts the assumption that $min \neq 0$. ∎