6
Timing Constraints in Message Sequence Chart Specifications

Hanêne Ben-Abdallah and Stefan Leue
University of Waterloo
Waterloo, ON N2L 3G1, Canada, telephone +1 (519) 888 4567,
email [hanene|sleue]@swen.uwaterloo.ca

Abstract

When dealing with timing constraints, the Z.120 standard of Message Sequence Charts (MSCs) 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 nor branchings. Finally, the paper extends our syntactic analysis of process divergence to MSCs with timing constraints.

Keywords

Message Sequence Charts, timing constraints, timing consistency analysis

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, e.g. (Selic, Gullekson & Ward 1994, Algayres, Lejeune, Hugonment & Hantz 1993, Jacobson & et al. 1992, Ichikawa, Itoh, Kato, Takura & Shibasaki 1991). The increasing popularity of MSCs recently motivated a standardization effort that produced the ITU-T Recommendation Z.120 (ITU-T 1996). The Z.120 standard defines two main 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,

Formal Description Techniques and Protocol Specification, Testing and Verification
T. Mizuno, N. Shiratori, T. Higashino & A. Togashi (Eds.) © 1997 IFIP. Published by Chapman & Hall

asynchronous fashion. An hMSC 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.

To facilitate the specification of real-time systems, a few extensions to MSCs have been proposed to express timing constraints: timers (ITU-T 1996), interval delays (Alur, Holzmann & Peled 1996, Meng-Siew 1993) and timing markers (Booch, Jacobson & Rumbaugh 1996). The proposed extensions evolved independently and differ in terms of their expressiveness and support for formal analysis. Further, all proposed analysis of MSCs with timing constraints have been so far limited to basic MSCs (Alur et al. 1996, Meng-Siew 1993).

In an effort to help consolidate 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 (Ben-Abdallah & Leue 1996). In addition, in this paper we extend our syntactic analysis of the process divergence problem in MSC specifications (Ben-Abdallah & Leue 1997*b*) in the presence of timing constraints. We use the example of an automatic teller machine system to illustrate our selected timing extension and the presented timing analysis.

2 TIMING CONSTRAINTS IN BASIC MSCS

There are essentially four classes of syntactic constructs to express timing constraints in MSCs and MSC reminiscent notations: timers (ITU-T 1996, Alur et al. 1996), delay intervals (Alur et al. 1996, Meng-Siew 1993), drawing rules (Booch et al. 1996), and other timing markers (Booch et al. 1996).

Timers. Recommendation Z.120 provides timers to express timing constraints in a basic MSC. Within a single process, a timer can be *set* to a value, *reset* to zero, and observed for *timeout.* A timer cannot be shared among concurrent processes in a basic MSC. Figure 1 (a) shows an example of a basic MSC with timing constraints expressed through two timers. In this example, for instance if the timer `T3.1` is set to 5, then `P3` must exchange its messages within at most five time units relative to the timer setting event. Process `P1`, on the other hand, first sets the timer `T1.1` say to three time units, receives message `a`, then resets its timer. Since process `P1` does not explicitly use a timeout event for the timer `T1.1`, the *implicit* assumption here is that the timer T1.1 does not expire before it is reset. As this example illustrates, a timer can be used to express a *maximal* delay between two or more consecutive events in one process. In addition, a timer can be also used to express a *minimal* delay between two consecutive events in one process.

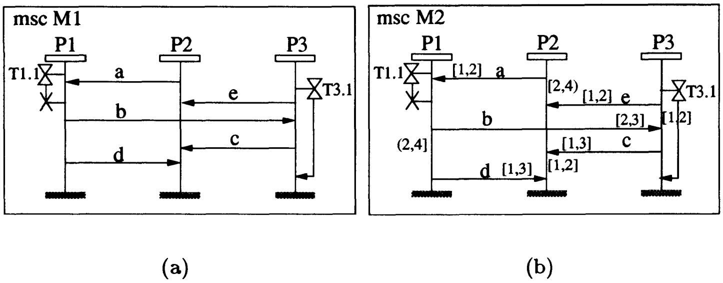


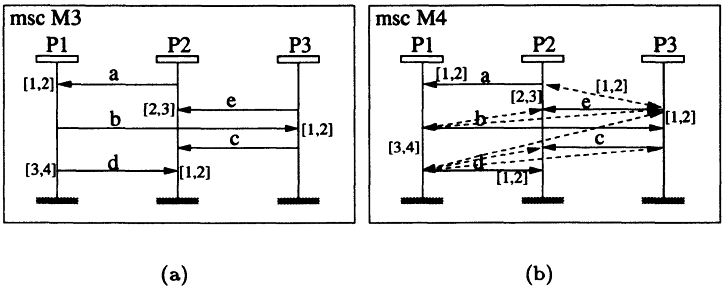
Figure 1 Timing constraints expressed through: (a) Z.120 timers; (b) Z.120 timers and delay intervals

Delay Intervals. Depending on how delay intervals annotate a bMSC, they express three types of timing constraints: 1) *event-associated* timing constraints (Meng-Siew 1993) which are denoted as an interval that is associated with an event in the basic MSC; 2) *message delivery* delays (Alur et al. 1996, Meng-Siew 1993) which are expressed as a time interval over a message arrow; and 3) *Processor's speed* constraints (Alur et al. 1996, Meng-Siew 1993) 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 an execution trace of the bMSC. Figure 2 (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 1 augments the timing constraints in Figure 1 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 (Meng-Siew 1993), the author generalizes the message delivery and processor's delay intervals (called *trace-associated* timing constraints) by using



(a) (b)

Figure 2 (a) Event-associated and (b) trace-associated timing constraints

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 2 (b) shows sample trace-associated timing constraints where the precedence 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 (Booch et al. 1996) 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 (Booch et al. 1996). 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.

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, one can use the temporal constraint network techniques in (Dechter, Meiri & Pearl 1991) to reduce timing analysis to computing all-pairs-shortest-paths in a labeled directed graph. 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 4.

The MSC analyzer tool (Alur et al. 1996) offers in addition to the above timing analysis for bMSCs, timed analysis based on a semantics that accounts for the queuing strategies in a bMSC. 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 as we discuss in the next section.

3 INTERPRETING TIMING CONSTRAINTS IN MSC SPECIFICATIONS

The hMSC in an MSC specification connects basic MSCs to describe sequential, possibly iterating and non-deterministic behavior. The presence of timing constraints as described in the previous section in MSC specifications requires attention for one essential reason: timing constraints can be spread across sequentially connected basic MSCs. We next illustrate how the Z.120 standard syntax (ITU-T 1995) is ill-defined when timers are used in hMSCs, and outline possible choices of interpreting timing consistency in MSC specifications with branchings.

Interpreting Iterations

Current analyses of iterations in an hMSC rely on unfolding loops a finite number of times and analyzing resulting basic MSCs (Alur et al. 1996). 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 3) the syntactic well-formedness of MSC specifications with timers.

Consider the MSC specification of Figure 3 (a) where the timer `T1.1` is

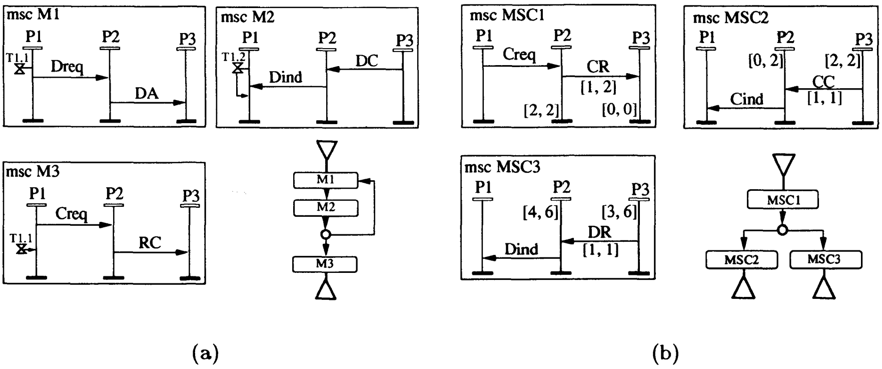


Figure 3 (a) Timers in a loop; (b) Timing constraints and branching

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 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: M1, M2, M1, M2, and then M3. Using loop unfolding, this sequence of basic MSCs represents a syntactically legal basic MSC. It then seems that a *new* timer is generated each time M1 is executed. Therefore, we need to resolve the association of the one timeout event in M3 with the two timer setting events. The choice obviously affects the timing consistency analysis.

Let us try to derive the semantics of repeatedly invoked timers from that of repeatedly sent messages. Ladkin and Leue (Ladkin & Leue 1995) use a finite state argument to interpret multiple sends of a message as deactivated by one receive of the message. (At the time of writing Z.120 does not deal with the semantics of iterating system behavior.) In the case of timers, this interpretation translates into 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 interpretation makes the above execution scenario timing inconsistent. Another alternative is to associate the timeout event with the last timer setting event. This coincides with the interpretation where the same timer is reset then reused during all iterations. In this case, the same execution scenario for the example of Figure 3 (a) is timing consistent in the sense that the last timer set does not expire before sending the event Creq; 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 (ITU-T 1995); 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.

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.

Interpreting Branchings

When an MSC specification contains branchings, we can determine whether its timing constraints are satisfiable in two ways:

- *Local semantics:* select one path at a time and analyze its timing constraints, independently of other paths that may branch out of the selected path (Alur et al. 1996). 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.
- *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 than the local semantics.

To illustrate the differences between the two approaches, consider the timed MSC specification show in Figure 3 (b) and which describes a simple connection establishment. The example is locally timing consistent. However, it is not globally timing consistent since there is no timing assignment for the common prefix of its two paths and that allows both paths to be simultaneously timing consistent. Timing consistency of the path leading to `MSC2` requires that receive `CR` occurs at most 1 time unit after it is sent, whereas timing consistency of the path leading to `MSC 3` requires that receive `CR` not to occur before 2 time units after it is sent.

4 TIMING ANALYSIS OF MSC SPECIFICATIONS

In this section, we first define the syntax of timed MSC specification we will use. Second, we augment the timing analysis for bMSCs presented in (Meng-Siew 1993, Alur et al. 1996) 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 analyze MSC specifications with branchings and iterations. For the proofs of the lemmas and theorems in this and the following Sections see (Ben-Abdallah & Leue 1997*a*).

Timed MSC Specifications

In remainder of this paper, we assume that the untimed bMSCs contain only message exchanges drawn in accordance to Z.120. To express timing constraints we use the Z.120 timer events together with (non-standard) timing delay intervals. A timer can be set to a positive integer value, and rest to zero. In compliance with the Z.120 standard (ITU-T 1995), a reset and timeout event must be preceded 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 labeling a message arrow denotes the relative minimal and maximal delays between the events of sending the message and receiving it. A delay interval labeling 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 $b \in \boldsymbol{N} \cup \{\infty\}$.

An *MSC specification* is a structure $S = (B, V, suc, ref)$ where: B is a finite set of bMSCs; V is a finite set of nodes partitioned into the singleton-set of *start* node, the set of *intermediate* nodes, and the set of *end* nodes; *suc* is 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* is a function that maps each intermediate node to a bMSC in B^*.

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 standard we allow timer events to be split

*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.

across bMSCs in an hMSC. The Z.120 standard restriction that each timeout and timer resetting event must be preceded 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 earlier, 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 preceded by a timer setting event in the loop. The first restriction disallows the example in Figure 3 (a). 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.

Timing Consistency of bMSCs

To determine the timing consistency of a basic MSC, we adopt an approach similar to the ones presented in (Meng-Siew 1993, Alur et al. 1996). 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 (Dechter et al. 1991, Meng-Siew 1993, Alur et al. 1996).

Since it is unclear whether the informal translation presented in (Alur et al. 1996) 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 $a, b \in \boldsymbol{N} \cup \{\infty\}$ $(a \leq b)$, we use the functions $\mathcal{L}(I)$ and $\mathcal{U}(I)$ defined as follows: $\mathcal{L}([a,b]) = \mathcal{L}([a,b)) = a$, $\mathcal{L}((a,b]) = \mathcal{L}((a,b)) = a^-$;

It is straightforward to represent a bMSC as a directed, labeled graph. 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 in addition. (For details, the reader is referred to (Ladkin & Leue 1995) 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 $\mathcal{T}_g(M)$: edge (e, e') with label $-\mathcal{L}(I)$, and 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 $\mathcal{T}_g(M)$: edge (e_i, e'_i) with label $-t$, and edge (e'_i, e) with label t, where the node e'_i is the node that corresponds to the end 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 (ITU-T 1995) 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.

A bMSC is timing consistent if and only if its temporal constraint graph has no cycles with a negative cost (Dechter et al. 1991, Meng-Siew 1993, Alur et al. 1996)*. Detecting cycles in the temporal constraint graph can be done through the Floyd-Warshall's algorithm (Papadimitriou & Steiglitz 1982) which computes all-pairs-shortest paths in the graph in a worst case time of $O(n^3)$ where n is the number of events in the graph.

Timing Consistency of hMSCs

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

Definition 4.1 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

*Addition over the natural numbers $\boldsymbol{N}$ is extended over $\boldsymbol{N} \cup \boldsymbol{N}^- \cup \{\infty, -\infty\}$ in a straightforward way.

next present a syntactic approach to determine the consistency of an MSC specification based on a finite subset of its paths. 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 4.1 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.

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 that 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 4.2 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: $(s, n_1) \in suc$ for the start node $s \in \top$; and 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 is 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. In addition, we say that a path $n_1, n_2, \cdots, n_j, \cdots, n_k, n_j$ in an MSC specification 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 4.1 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.

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 (Ben-Abdallah & Leue 1996) the algorithm shown in Table 1. 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 4 in a straightforward way based on the

Table 1 Timing consistency analysis 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
```

following fact: the behavior of $M_1 \bullet M_2$ is equivalent to the behavior of the bMSCs obtained by gluing 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 matrix 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.

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 (Ben-Abdallah & Leue 1997*b*). 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 discrepancies, e.g., unexpected deadlocks since message queues are finite and messages must be dropped or over-written.

In (Ben-Abdallah & Leue 1997*b*), 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 4.2 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 2) the minimum of all maximal delays of the processes receiving messages from $P_1, \cdots, P_n$ including delays on the received messages is equal to ∞.

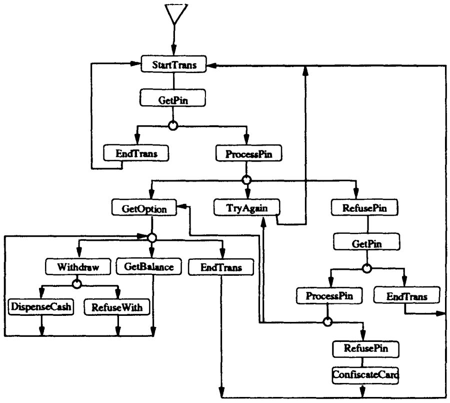


Figure 4 High-level MSC for the ATM example

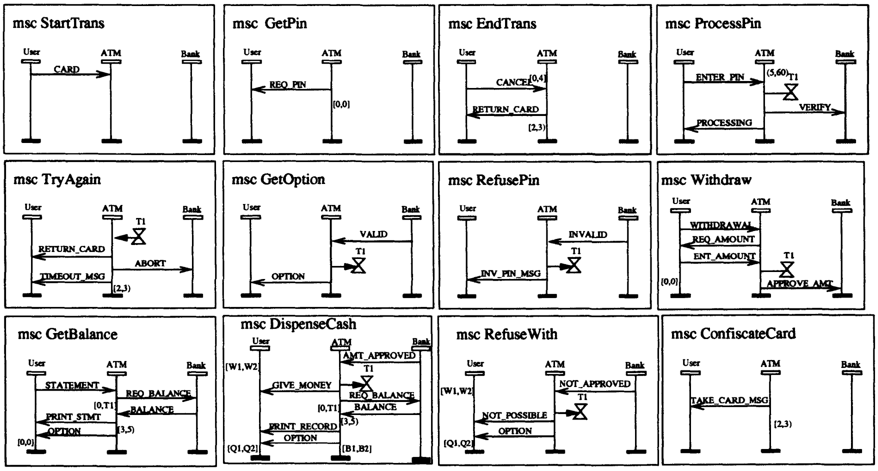


Figure 5 Basic MSCs in the ATM example

5 ATM EXAMPLE

To illustrate our timing analysis, consider the MSC specification of an automatic teller machine (ATM) system shown in Figures 4 and 5. The ATM system consists of three components: potential customers (process **User**), the ATM controller (process **ATM**), and a host computer in a bank (process **Bank**).

Initially, the ATM controller waits to receive the customer's bank card. Then, it either receives a request to cancel the transaction within $[0,4]$ seconds (bMSC **EndTrans**), or receives the customer's pin number within $[5,60]$ seconds (bMSC **ProcessPin**). 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 return to its initial state. The ATM expects a reply from the bank within T_1 seconds. This timing constraint is expressed through the `T1` timer as well as the $[0, T_1]$ delay interval. In (bMSC `TryAgain`), when `T1` times out, the card is returned, an appropriate message is then displayed, and the ATM takes again between $[2, 3)$ seconds to return to its initial state. Our specification also describes the following constraints: a customer expects a withdraw request to be processed within $[W_1, W_2]$ seconds relative to the time of entering an amount; a customer takes $[Q_1, Q_2]$ seconds to decide whether to make another transaction while the ATM has the card; the ATM takes $[B_1, B_2]$ seconds for book-keeping after dispensing cash; the ATM takes $[3, 5)$ seconds to print a receipt after receiving the balance information from the bank. Each ATM-customer communication has a delay of $[0, 2)$ seconds and each vertical line without a time delay has a default delay of $[0, \infty)$, which we do not explicitly represent in the chart. Note that when timing constraints extend over more than just one bMSC it is necessary to use auxiliary $[0, 0]$ constraints.

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

Timing Analysis. It is easy to check that our MSC specification of the ATM satisfies the syntactic conditions of Theorem 4.1. We used our analysis tool (Ben-Abdallah & Leue 1996) 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. The tool generated 43 infinite closed sequential components whose temporal graphs were then examined for cycles with a negative cost. For the case (1) in Table 2, the user does not impose any timing constraints on the system, which in fact makes any value acceptable for the remaining variables. In the case (2), the user expects the ATM to process their withdraw request within $[0, 3]$ seconds which leads to a timing inconsistency. The tool detects the event send `ENT_AMOUNT` as being in a cycle with a negative cost of -1. This gave us the hint to increase the upper-bound of the delay to 4 which gave us timing consistency (case (3)). In the cases (4) and (5), we examine the effects of the book-keeping time $[B_1, B_2]$. Due to the implicit $[0, \infty)$ bounds we only needed to vary the lower bound. 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.

6 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 (Meng-Siew 1993, Alur et al. 1996) can describe timing constraints for events within a process and events that are directly related, i.e., via the control edge or message arrow. To express more general timing constraints, e.g., to relate events within different basic MSCs or processes, these extensions must be further augmented with temporal predicates (Booch et al. 1996).

Following the analysis techniques of temporal constraint networks (Dechter et al. 1991), timing consistency of a basic MSC is reduced to checking cycles with negative cost in a directed graph (Meng-Siew 1993, Alur et al. 1996). 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 have 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. Furthermore, we have extended in this paper our syntactic analysis of process divergence in iterating MSCs in the presence of timing constraints.

Acknowledgements. This work was supported by ObjecTime Limited and the Information Technology Research Centre of the Province of Ontario.

REFERENCES

Algayres, B., Lejeune, Y., Hugonment, F. & Hantz, F. (1993), The AVALON project: a validation environment for SDL/MSC descriptions, *in* O. Faergemand & A. Sarma, eds, 'Proceedings of the 6th SDL Forum, SDL'93: Using Objects'.

Alur, R., Holzmann, G. J. & Peled, D. (1996), An analyzer for message sequence charts, *in* T. Margaria & B. Steffen, eds, 'Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1055', Springer Verlag, pp. 35–48.

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

Ben-Abdallah, H. & Leue, S. (1997*a*), Expressing and analyzing timing constraints in message sequence chart specifications, Technical Report 97-

04, Department of Electrical & Computer Engineering, University of Waterloo. 24 p.

Ben-Abdallah, H. & Leue, S. (1997*b*), Syntactic detection of process divergence and non-local choice in message sequence charts, *in* E. Brinksma, ed., 'Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1217', Springer Verlag, pp. 259–274.

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

Dechter, R., Meiri, I. & Pearl, J. (1991), 'Temporal constraint networks', *Artificial Intelligence* **49**, 61–95.

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

ITU-T (1995), 'Recommendation Z.120, Annex B: Algebraic Semantics of Message Sequence Charts', ITU - Telecommunication Standardization Sector, Geneva, Switzerland.

ITU-T (1996), 'Recommendation Z.120', ITU - Telecommunication Standardization Sector, Geneva, Switzerland. Review Draft Version.

Jacobson, I. & et al. (1992), *Object-Oriented Software Engineering - A Use-case Driven Approach*, Addison-Wesley.

Ladkin, P. B. & Leue, S. (1995), 'Interpreting Message Flow Graphs', *Formal Aspects of Computing* **7**(5), 473–509.

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

Papadimitriou, C. & Steiglitz, K. (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.

Selic, B., Gullekson, G. & Ward, P. (1994), *Real-Time Object-Oriented Modelling*, John Wiley & Sons, Inc.

Hanêne Ben-Abdallah received the B.S. degree in computer science and mathematics from the University of Minnesota, Minneapolis, in 1989, the M.S.E. degree in 1991 and Ph.D. degree in 1996 in computer science from the University of Pennsylvania, Philadelphia. From 1996 to 1997 she was a postdoctoral research fellow in the Dept. of Elect. & Comp. Eng. at the University of Waterloo. Her research interests include real-time systems and formal methods in software engineering.

Stefan Leue received his Diplom-Informatiker degree from the University of Hamburg (Germany) in 1990 and his Ph.D. from the University of Berne (Switzerland) in 1995. He is currently an Assistant Professor in the Department of Electrical & Computer Engineering of the University of Waterloo (Canada).