Syntactic Analysis
of
Message Sequence Chart Specifications

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

Technical Report 96-12

hanene|sleue@swen.uwaterloo.ca

November 1996

Abstract

Message Sequence Charts (MSCs) are gaining popularity in software engineering methods for concurrent and real-time systems. They are increasingly supported in software engineering tools to capture, for instance, system requirements, test scenarios, and simulation traces. MSCs have been standardized by ITU-T in Recommendation Z.120 [15]. However, various aspects of environment behavior remain underspecified in MSCs, e.g., the presence of resources for inter-process communication and the coordination of concurrent processes at points of control branching. Such underspecifications can result in ambiguities in an MSC specification and discrepancies between an MSC specification and its implementation. In this paper we characterize two consequences of harmful underspecifications: *process divergence* and *non-local branching choice*. We also present two syntax-based analysis algorithms that detect both problems. The syntactic characterization of these problems requires an MSC specification to be deadlock-free. Therefore, we also discuss *deadlock detection* in MSC specifications.

Contents

List of Figures

1 Introduction

The intuitive, graphical notation of Message Sequence Charts (MSCs) increased their popularity within the software engineering community. MSCs have already been adopted within several software engineering methodologies and tools for concurrent, reactive and real-time systems, e.g., [24],[12], [26, 6], [16], [27], [7], [2], and [11, 3]. MSCs are used to document system requirements that guide the system design [27], describe test cases and scenarios [16, 6, 7], express system properties that are verified against SDL specifications [2], visualize sample behavior of a simulated system specification [27, 2], and to express legacy specifications in an intermediate representation that helps in software maintenance and reengineering [12].

The syntax of MSCs is defined by the ITU-T in Recommendation Z.120 [15]. An MSC essentially consists of a set of processes (called instances in Z.120) that run in parallel and exchange messages in a one-to-one, asynchronous fashion. In addition to exchanging messages, processes can individually execute internal *actions*, use *timers* to describe timing constraints, create and terminate process instances.

The behavior of an MSC essentially consists of sequences (or traces) of messages that are sent and received among the concurrent processes in the MSC. The order of communication events (i.e. sent or received messages) in a trace is deduced from the visual flow of control within each process in the MSC along with a causal dependency between the event of sending and receiving a message. Several approaches have been proposed to formalize the semantics of MSCs. They range from adopting the policy of "what-you-see-is-what-you-get"(e.g., [2, 12]) to incorporating constraints pertinent to implementation, e.g., architecture and queuing protocols [3]. In addition, these approaches differ in terms of their techniques: they derive the traces of an MSC through a translation to either a process algebra [23, 14], an algebraic structure called Message Flow Graph [18], an automaton [17], or Petri Nets [9].

A consequence of using MSCs in industrial-size applications is the tendency to use them in a modular and hierarchical fashion just like other specification languages, e.g., RoomCharts [27]. For this, the standard Z.120 [14] evolved to allow the description of a large system by composing *basic* MSCs [15]. The resulting graphical language, called *High-Level MSCs* (hMSCs), provides for operators to connect basic MSCs to describe parallel, sequential, iterating, and non-deterministic execution of basic MSCs. In addition, hMSCs can describe a system in a hierarchical fashion by combining hMSCs within an hMSC. In this paper, we call an hMSC together with its referenced bMSCs an *MSC specification.*

While the syntax of hMSCs has been well defined in Z.120, the introduction of the sequential and non-deterministic execution can lead to unimplementable MSC specifications or implementations with behavior unintended in the MSC specifications. More specifically, an MSC specification can lead to an implementation with discrepant behavior due to two problems we call *process divergence* and *non-local branching choice.* These two problems are in fact independent of the semantics of basic MSCs, and rather are the result of under-specification of two factors: 1) resource related constraints, e.g., processor speed, system architecture and queuing protocols, and 2) the "environment" from the point of view of individual processes in an MSC specification.

Consider the MSC specification of Figure 1. At this level of abstraction, the visual interpretation is that the basic MSC `MSC1` is iteratively executed. From the point of view of process `P1` in the basic MSC `MSC1`, this process will repeatedly send messages `req1` then `req2` to the process `P2`. Since

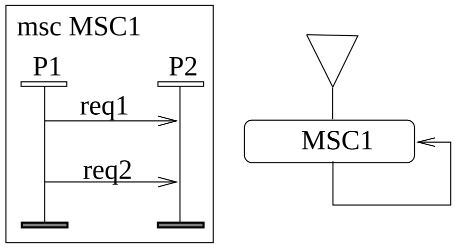


Figure 1: An MSC specification with process divergence

communication is asynchronous and there is no explicit information about the communication link, queuing strategy, nor processor speed, an interpretation of this MSC specification can allow process `P1` to run faster than process `P2` and overflood it with messages `req1` and `req2`. We call such a behavior *process divergence*. Such system behavior is usually unintended in the MSC specification. In addition, its semantic implications, e.g., presence of buffers, are not explicitly accounted for in the MSC specification. Furthermore, it can lead to an implementation that behaves differently from the specification, e.g., implementation loses multiple copies of `req1` and `req2`, or overwrites multiple copies of a message.

The second problem that may impede the implementation of an MSC specification is non-local branching choice. Consider the MSC specification of Figure 2 where after behaving as described in the basic MSC `M`, the system has a choice between behaving either as described in the basic MSC `M1` or `M2`. An implicit assumption in this interpretation is that the processes `P1` and `P2` will synchronize their choices between behaving either as in `M1` or `M2`. However, when examining the two processes, one sees that this implicit assumption can not be implemented, in a modular way, without introducing the unintended behavior where process `P1` chooses to branch left and send an `a` message while process `P2` chooses to branch right and send a `b` message.

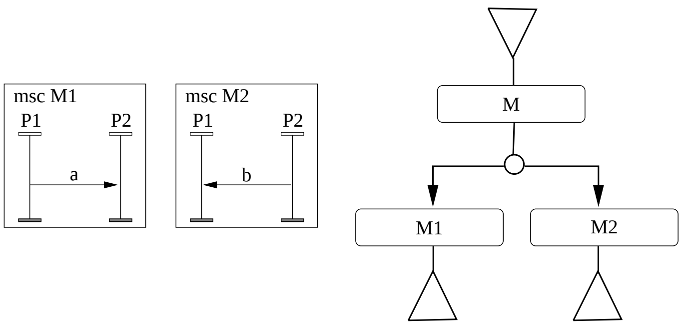


Figure 2: An MSC specification with a non-local branching choice

To avoid a discrepancy between the MSC specification and its implementation, it is therefore essential to detect such anomalies before the implementation phase. The designer would be given the options either to allow the unintended behavior or resolve it by adjusting the processes in the basic MSCs. To be efficient, the anomaly detection mechanisms should operate on the syntax of an MSC specification, as oppose to its interpretation which often is exponential in the size of the

MSC specification.

Currently, basic MSCs can be analyzed for syntactic well-formedness [25], deadlocks [19], and race conditions [3]. These analysis techniques deal with basic MSCs. However, as the earlier examples outlined the sequential and non-deterministic composition of basic MSCs within an hMSC create other behavioral anomalies that must be addressed to facilitate the implementation of the MSC specification. In this paper, we syntactically characterize process divergence and non-local branching choice. Our analysis complements the analysis presented in [3] as it is another step towards ensuring that an MSC specification is implementable in a modular fashion and without discrepancies.

Paper organization. Section 2 briefly reviews the syntax and our semantics of a basic MSC, high-level MSC, and MSC specification. Section 3 reports on syntactic analysis of bMSCs for deadlocks; this analysis can be used as a preprocessing phase to our analyses which assume deadlock-freeness of the MSC specification. Sections 4 and 5, respectively, define the problems of process divergence and non-local branching choice and present our syntax-based detection algorithms. Section 6 reviews relevant work about MSC semantics. Section 7 summarizes the paper and outlines future research directions. Appendix A defines notation used in various parts of the paper. Appendix B contains the detailed proofs of lemmas and theorems stated in the paper.

2 Message Sequence Chart Specifications

Message Sequence Charts (MSCs) are a graphical specification formalism. They describe message exchanges between system processes, and abstract out data and internal computations. The internal process behavior is inferred from the message exchanges in an MSC, in contrast to other formalisms such as LOTOS [13] and SDL [8] where the process behavior is explicit and the communication structure is inferred.

As mentioned in the introduction, the use of MSCs has evolved from describing and visualizing sample, finite system runs and test cases as *basic* MSCs, to describing the behavior of complex systems in a modular and hierarchical fashion as *high-level* MSCs. In this section, we briefly describe the syntax and semantics of the subset of basic and high-level MSCs we use throughout the paper; for details the reader is referred to [18, 20].

2.1 Basic Message Sequence Charts

A *basic MSC* (bMSC) describes finite executions of concurrent processes in a system; see Figure 3 for an example. Each vertical line is delimited by a *start* and *end* symbol and represents one process in the system. (Recommendation Z.120 [15] calls a process an *instance*.) Each horizontal or sloping arrow describes a message sent from the process at the tail of the arrow to the process at the head. Communication is one-to-one and asynchronous, i.e., sending a message is non-blocking. Processes have disjoint name labels, and message arrows have labels that denote message types. Control flows independently within each process from the start symbol to the end symbol.

In Figure 3, the basic MSC `ahp` consists of three concurrent processes, `P1`, `P2` and `P3`. Process `P1` starts by sending a message of type `a` to process `P2` and then sends a `c` message to process

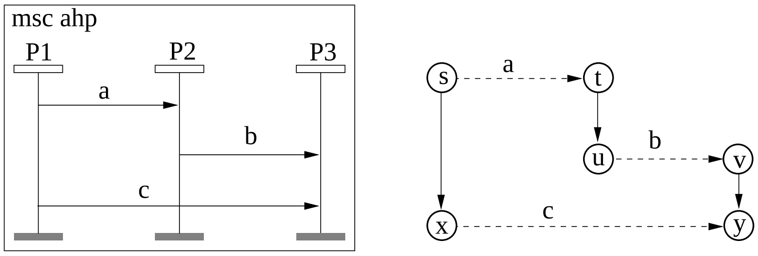


Figure 3: Basic MSC (left) and corresponding basic MFG (right)

P3. After receiving the a message, process P2 sends a b message to process P3. The asynchronous nature of communication allows sending c to be executed before receiving a as well as sending b.

In addition to message exchanges, the Z.120 standard [15] allows a bMSC to contain conditions which describe the state of a subset of processes in the MSC, actions, timers, and process instantiation and stopping. In this paper, however, we restrict our attention to bMSCs only with message exchanges.

The behavior of a bMSC is the set of sequences (or traces) of sent and received messages—called send and receive events, respectively. It is deduced from the order of events within each process in the bMSC together with the causal precedence between sending and receiving a message. Within each process, events are totally ordered according to their position from the start to the end symbols on the process axis. In addition, for each message in the bMSC, its send event is ordered before its receive event. In general, the overall events in a bMSC are partially ordered.

In this paper, to interpret bMSCs we follow the two-step approach presented in [18]: First translate the bMSC into an algebraic structure called Message Flow Graph (MFG); then derive the reachable states and communication events that the MFG (and thus bMSC) can execute. The second step is done via a labeled transition system called Global State Transition Graph (GSTG). Appendix A formally defines MFGs. We next informally describe the correspondence between a bMSC and an MFG, as well as the concept of a GSTG.

Let M be a bMSC; its corresponding message flow graph, $F_M = (S, C, ne, sig, ST, stype, ET, etype)$, is a directed graph. Each node represents a communication event in the basic MSC, i.e., arrow tail or head in the basic MSC. Each node is labeled and has a type that consists of two parts: 1) the type of the corresponding communication event: ! for a send event and ? for a receive event; and 2) the type of the corresponding message arrow which is drawn from ST. The type of a node is retrieved with the function *etype* and belongs to the set $ET = \{!, ?\} \times ST$.

Nodes in the MFG F_M are connected by two types of edges: edges that reflect the control flow between communication events within each process of the bMSC, and edges that reflect the causal precedence of a message's send and receive events. The control flow edges, set *ne*, are called *next event* edges and are graphically represented by solid line arrows. The causal precedence edges, set *sig*, are called *signal* edges and are graphically represented by dashed line arrows. Each signal edge in the set *sig* is labeled with the corresponding message type from ST and which is retrieved with the function *stype*.

Example 2.1 Consider the bMSC and its corresponding MFG in Figure 3. The type of node s is $etype(s) =!a$ and the type of node t is $etype(t) =?a$. The edge (s, x) is a next event edge. The edge

(s, t) is a signal edge with label $stype((s, t)) = a$.

Property 2.1 The MFG which results from translating a bMSC has in particular the following properties:

1. there is a one-to-one mapping between the messages (i.e., arrows) in the bMSC and the set sig of signal edges of the MFG;
2. the set ne of next event edges is a non-branching and cycle-free relation;
3. in the directed graph $(S \cup C, ne)$ each maximal, connected component corresponds to a unique process in the bMSC; also each process in the bMSC has a unique corresponding connected component in $(S \cup C, ne)$. We can therefore talk about *process* in the MFG, and use $ptype(n)$ to denote the process to which belongs a node n in the MFG [18];
4. each component in $(S \cup C, ne)$ has a unique *start* node (a node with no incoming next-event edges) and a unique *finish* node (a node with no outgoing next-event edges).

We call an MFG that corresponds to a basic MSC a *basic MFG* (bMFG)[1].

One of the advantages of MFGs is that they allow us to distinguish between different occurrences of a message with the same type in a bMSC. Note that MFGs are merely an algebraic structure that allows us to reason about basic MSCs, and as we see shortly MSC specifications. In particular, MFGs are used to derive the behavior of a bMSC, i.e., all possible states and communication events that the bMSC can execute. This is described by a labeled transition system called *Global State Transition Graph* (GSTG).

Informally, a state of the GSTG consists of a subset of next-event edges, and a subset of signal edges each of which represents *one* copy of a message sent but not yet received. These edges define all the steps that the system can execute at any particular state. In any state, the *enabled* events, i.e., can be executed, are events that result from two cases: 1) a send event whose next-event edge is in the state, and 2) a receive event whose incoming signal edge and at least one next-event edge is in the state.

A transition in the GSTG consists of sending or receiving an enabled event, i.e., taking an edge. The result of a transition on sending an event augments the target state with a signal edge that represents the fact that a message was sent but not yet received. A transition on receiving a message removes the corresponding signal edge from the target state. One note to make about this semantics is that, in accordance with Z.120 Annex B [14], it does not support any queuing mechanism and assumes that multiple copies of a sent message are disabled by one reception of the message. The reader is referred to [18] for a detailed, formal description of how a GSTG is derived from an MFG. In the remainder of this paper, we denote a transition from state q to state q' and label α as $q \xrightarrow{\alpha} q'$.

Example 2.2 Consider Example 2.1. The GSTG of the bMFG has the initial state $q_0 = \{(s, x)\}$ where the event $etype(s) = !a$ is enabled and thus the only transition out of q_0 is

$$q_0 \xrightarrow{!a} \{< s, t >, (x, y)\}.$$

[1] A basic MFG is called *simple* MFG in [18].

The signal edge $< s, t >$ represents the fact that a message of type a was sent but not yet received. After the above transition, there are two possible transitions:

$$\{< s, t >, (x, y)\} \xrightarrow{?a} \{(x, y), (t, u)\} \quad \text{or} \quad \{< s, t >, (x, y)\} \xrightarrow{!c} \{< s, t >, < x, y >\}$$

The choice between the two is non-deterministic. The next possible transitions are:

$$\{(x, y), (t, u)\} \xrightarrow{!c} \{< x, y >, (t, u)\} \qquad \text{or} \quad \{(x, y), (t, u)\} \xrightarrow{!b} \{(x, y), < u, v >\}$$

or

$$\{< s, t >, < x, y >\} \xrightarrow{?a} \{< x, y >, (t, u)\}$$

and so on so forth until a state is empty.

2.2 High-Level MSCs and MSC Specifications

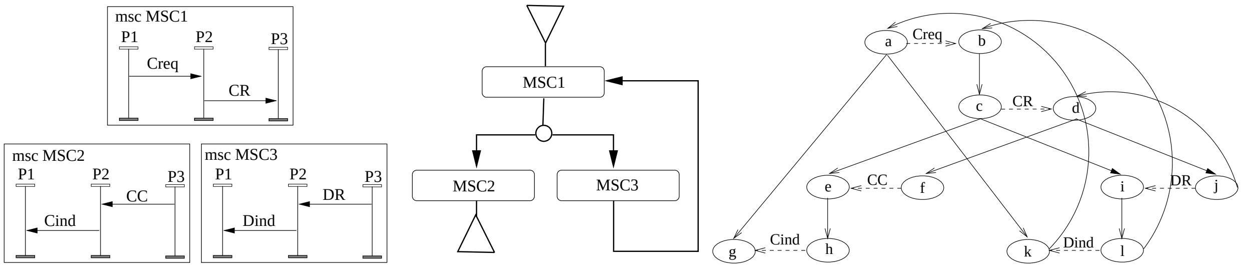
Reactive systems often consist of non-terminating and non-deterministic processes. To provide for such systems, the recommendation Z.120 suggests *High-Level MSCs* (hMSCs) to compose basic MSCs to specify systems with recursive and non-deterministic behavior. An hMSC is a digraph where nodes refer to bMSCs and edges indicate possible continuations of bMSCs by others. In addition, an hMSC has two distinguished types of nodes: one required *start* node that indicates the beginning of the specification, and optional *end* nodes that indicate the termination of the specification. For example, the hMSC in Figure 2 describes a system that first starts behaving as described by the bMSC `M`, then can either behave as described by the bMSC `M1` or `M2`; afterwards, the system stops its execution. The hMSC in Figure 1, on the other hand, describes a system whose behavior is represented by the non-terminating, iterative execution of bMSC `MSC1`.

To simplify the presentation of our analysis technique, in the sequel we assume that nodes in an hMSC only refer to bMSCs. However, our analysis can be easily extended to allow hMSCs with nodes that refer to other hMSCs as defined in recommendation Z.120 [14].

Definition 2.1 *An* MSC specification *is a structure* $S = (B, V, suc, ref)$ *where*

- B *is a finite set of bMFGs;*
- $V = \top \cup I \cup \bot$ *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 bMFG in* B*.*

From an MSC Specification to an MFG. We also define the semantics of an MSC specification through an MFG by extending the bMSC to bMFG translation as follows. First, for each intermediate node that references a bMSC in the hMSC, create the bMFG corresponding the bMSC as described in the previous section. Secondly, for each edge `(b1,b2)` from the node referencing bMSC `M1` to the node referencing bMSC `M2` in the hMSC, add a next-event edge from each finish



(a) (b)

Figure 4: (a) MSC Specification; (b) its corresponding MFG.

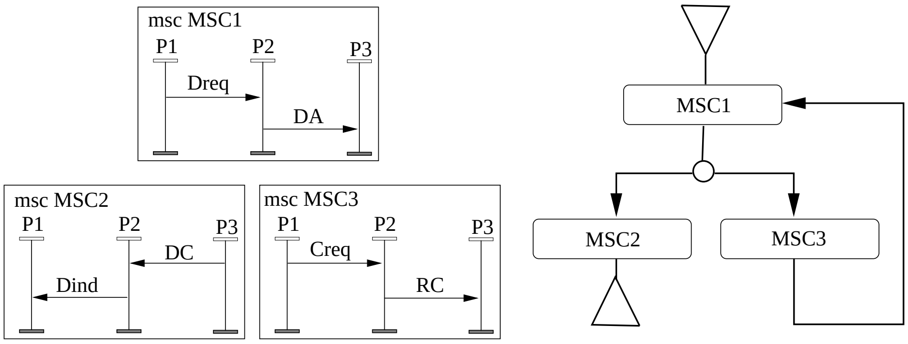


Figure 5: MSC Specification example IV.

node in the bMFG of `M1` to the start node in the bMFG of `M2` such that the finish and start nodes belong to the same process. It is easy to prove that the result of this translation is an MFG. Figures 4 and 6 illustrate other examples of MFGs for MSC specifications.

Note that when the hMSC of an MSC specification contains a loop or a branching the resulting MFG also contains loops and branchings and therefore it is not a basic MFG. However, the resulting MFG satisfies the remaining conditions in Property 2.1.

From MFG to GSTG. In [18, 21], the semantics of an MSC specification is derived from its MFG while accounting for possible branching. Informally, the translation uses a history variable to register the branching decisions made by any process that is ahead of others. One important note to make about this semantics is that it does not implement the delayed choice semantics to resolve non-determinism as late as possible; this is a deviation from the recommendation Z.120 [14].

3 Deadlock Detection in MSC Specifications

Semantically, a deadlock is a system state where no further execution steps are possible. Semantic deadlock detection is however expensive: for systems consisting of a collection of communicating

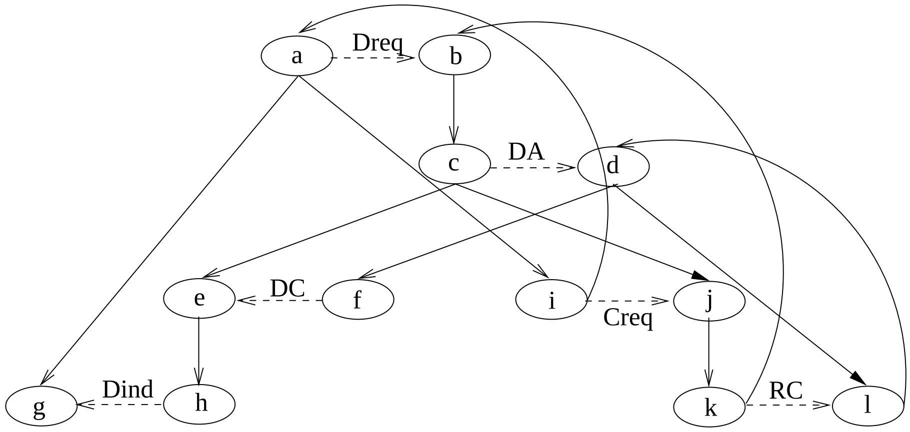


Figure 6: MFG corresponding to MSC Specification example IV.

finite state machines the problem is PSPACE complete at best, and it becomes undecidable if the message buffers have unbounded length [10]. One is therefore interested in specifying syntactic ways to detect deadlocks.

Z.120 Message Syntax and Drawing Rules

The standard syntax of basic MSCs [15] indirectly guarantees that a basic MSC is deadlock-free via two conditions: an informal constraint about the causality of messages and a drawing rule for message arrows in a bMSC.

Section 4.3 of Z.120 (*"Message"*) [14] informally specifies the following syntactic constraint on messages in a bMSC:

> "*It is not allowed that the ⟨message output⟩ is causally depending on its ⟨message input⟩ via other messages or general ordering constructs. This is the case if the connectivity graph contains loops.*"

The Z.120 "connectivity graph" of a bMSC is isomorphic to our bMFG. In addition, both connectivity graphs and bMFGs are isomorphic to the *mfg* graphs defined by Ladkin and Simons [19] who prove that an *mfg* is deadlock-free if and only if:

1) its $sig \cup ne$ relation is acyclic, and

2) each of its nodes has a matching node with which it participates in a communication action.

It is clear that bMSCs and bMFGs satisfy the second condition. Hence, syntactic deadlock detection in an arbitrary bMFG that does not necessarily satisfy the above Z.120 constraint is as hard as cycle detection in a directed graph. However, those bMSCs composed in accordance with the above Z.120 informal constraint are deadlock-free.

A simpler Z.120 syntactic constraint that eliminates deadlocks in a bMSC is a drawing rule. In Section 2.4 (*"Drawing Rules"*) Z.120 [14] restricts the way message arrows are drawn as follows:

"Message lines may be horizontal or with downward slope (with respect to the direction of the arrow), ..."

Conjecture 3.1 *A bMSC that has only horizontal or downwards sloping message arrows is deadlock free.*

Proof sketch The proof is based on a topological argument. Assign to each event a number that reflects its vertical distance from the beginning of the process in which it resides. Note that a cycle involves, for each process, a receive node (i.e., event) before (i.e., with a shorter distance) a send node such that the receive node is reachable through the $ne \cup sig$ relation from the send node. Now, start at any send node; if sig edges can only go across (i.e. horizontally) or downwards, and ne edges only go downwards, then following a path can only lead to nodes with larger distances. Therefore, we cannot reach a receive node above its corresponding send node from which we started, since its measure is less. Thus, the above drawing rule for message arrows guarantees that the bMSC has no cycles. The absence of a cycle in turn implies absence of deadlocks [19]. ∎

MSC Specifications

The above syntactic characterizations of deadlocks in bMFGs can be easily adapted for MSC specifications. In particular, it is easy to prove that for a given MSC specification $S = (B, V, suc, ref)$, if each bMFG $F \in B$ has an acyclic $sig_F \cup ne_F$ relation and if S has no branching, then S is deadlock-free. In the presence of branching, a deadlock can happen in S if processes branch into different basic MSCs. We will revisit this topic in Section 5.

In the sequel, the necessity of our syntactic characterization of process divergence and non-local branching choice in MSC specifications will make use of reachability of a *problematic* state. The presence of a branching does not preclude the reachability of a state; however, sequential composition of bMSCs with cycles does. We will therefore assume throughout the paper that each MSC specification has bMFGs with an acyclic $sig \cup ne$ relation.

4 Process Divergence

Processes in an MSC specification execute concurrently and exchange messages through asynchronous communication. When processes iterate in an MSC specification, the asynchronous nature of communication can lead to *process divergence*: a system execution where one process sends a message an unbounded number of times ahead of the receiving process. Since an MSC specification makes no assumption about the speed of its processes, in the absence of a *hand-shake* mechanism, a sender process can run "faster" than a receiver process—possibly flooding the receiver with messages.

Process divergence can lead to discrepancies between the specification and implementation, e.g., message over-writing and unexpected deadlocks, as well as unimplementable specifications, e.g., one that requires message queues with infinite sizes. It is therefore essential to detect potential process divergences in an MSC specification prior to implementation.

As an example of process divergence, consider the MSC specification of Figure 1. One possible execution of this MSC specification is the infinite trace !`req1` !`req2` !`req1` !`req2` $\cdots$ which is the result of process P1 sending messages without process P2 receiving any one. To handle such a potential execution, the implementation must answer several questions: What is the network architecture between the processes P1 and P2? Is there any queuing mechanism and protocol? How are multiple copies of a not-yet received message handled?

Regardless of the answers to the above questions, none of them is based on information explicitly described in the given MSC specification. In addition, different answers may result in different implementations. Furthermore, while the above questions seem pertinent to the implementation phase, we view process divergence as *unintended* behavior of the specification that must be rather detected and brought to the designer's attention. This allows the designer to decide either to modify the specification to resolve the problem (e.g., by adding explicit hand-shakes), or to postpone the problem to the implementation phase which refines the specification.

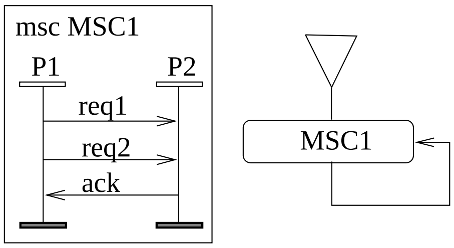


Figure 7: An MSC Specification with no process divergence

It is worth noting that there are MSC specifications for which the above questions are irrelevant. For instance, consider the MSC specification in Figure 7 which slightly differs from the specification in Figure 1: The two processes P1 and P2 in Figure 7 have a hand-shake communication. The resulting specification can exhibit an execution with an unbounded numbers of events. However, before starting any new iteration, process P1 must wait for the reception of ack before sending `req1`; similarly, process P2 must wait for `req1` (and `req2`) before sending `ack`. Thus, neither process can send an unbounded numbers of messages before the other process can receive any. As a result, questions about queuing messages are irrelevant for the specification of Figure 7.

Note also that in the above two examples we showed the presence or absence of process divergence irrespectively of any particular semantic interpretations or implementation related constraints. We analyzed the MSC specifications simply by examining the communication between its processes. In the remainder of this section, we formalize our notion of process divergence and present our syntactic approach to detect their presence.

4.1 Semantic Characterization of Process Divergence

In the sequel, we use E^ω to denote the set of strings over a finite set of alphabet E, and $\mathbb{N}$ to denote the set of integers. The function $\# \subseteq ((E^\omega \times E \times \mathbb{N}) \times \mathbb{N})$, for a string $s \in E^\omega$, letter $e \in E$, and an integer m, computes the number of occurrences of e in the prefix of s of length m; we write it as $\#_s(e, m)$.

Definition 4.1 *Let* $\mathcal{F} = (S, C, ne, sig, ST, \text{stype}, ET, \text{etype})$ *be an MFG and let* $\mathcal{G}_F = (Q, q_0, T_F)$ *be its GSTG.*

We say $\mathcal{F}$ is divergent *if there exist $< x, y > \in sig$ and an infinite sequence of transitions in T_F*

$$q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \cdots$$

such that for $s = \alpha_0\alpha_1\alpha_2\cdots$, we have

$$\forall n \in \boldsymbol{N} \exists m \in \boldsymbol{N} \quad \#_s(x, m) > n + \#_s(y, m) \quad .$$

When an MFG $\mathcal{F}$ is not divergent, we say $\mathcal{F}$ is non-divergent *or* divergence-free.

Definition 4.2 *An MSC specification is* divergent *if its MFG is divergent.*

In the special case of an MFG with a finite-state GSTG, the Definition 4.1 for divergence can be implied by a weaker property as stated in the next lemma.

Lemma 4.1 *Let $\mathcal{F} = (S, C, ne, sig, ST, \mathrm{stype}, ET, \mathrm{etype})$ be an MFG and let $\mathcal{G}_F = (Q, q_0, T_F)$ be its finite-state GSTG.*
$\mathcal{F}$ is divergent if there exist $< x, y > \in sig$ and an infinite sequence of transitions in T_F

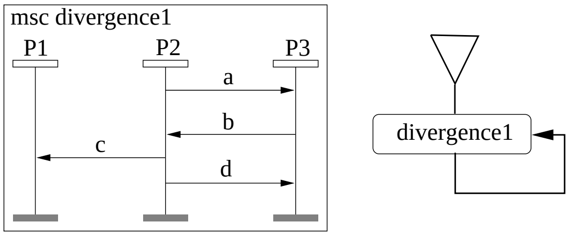
$$q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \cdots$$

such that for $s = \alpha_0\alpha_1\alpha_2\cdots$, we have

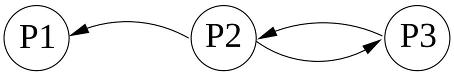
$$\exists n \in \boldsymbol{N} \forall m \geq n \quad \#_s(x, m) > 1 + \#_s(y, m) \quad .$$

4.2 Syntactic Characterization of Process Divergence

Semantic detection of process divergence is often expensive. In this section we syntactically characterize process divergence. To illustrate the intuition behind our syntactic characterization, let us first examine samples of MSC specifications.



(a)



(b)

Figure 8: (a) MSC example `divergence1`; (b) its Coordination graph `coordination1`

The MSC specification in Figure 8 (a) contains a process divergence: processes `P2` and `P3` may jointly race ahead of process `P1`. Since divergence is tied to the way processes exchange messages, let us abstract out the number and order of exchanged messages. Figure 8 (b) contains a directed graph, coordination1, that describes the messages exchanged between the three processes of MSC divergence1. Each node represents a process and a directed edge between two nodes represents a message sent from the source process to the target process. Note that in the graph coordination1 processes P2 and P3 exchange messages in both directions and thus have a hand-shake mechanism. Such a tight dependency forces the two processes to synchronize their progress and thus eliminate potential divergence of either one with respect to the other. On the other hand, process P2 sends messages to P1 without receiving any which allows it to send a potentially unbounded number of messages.

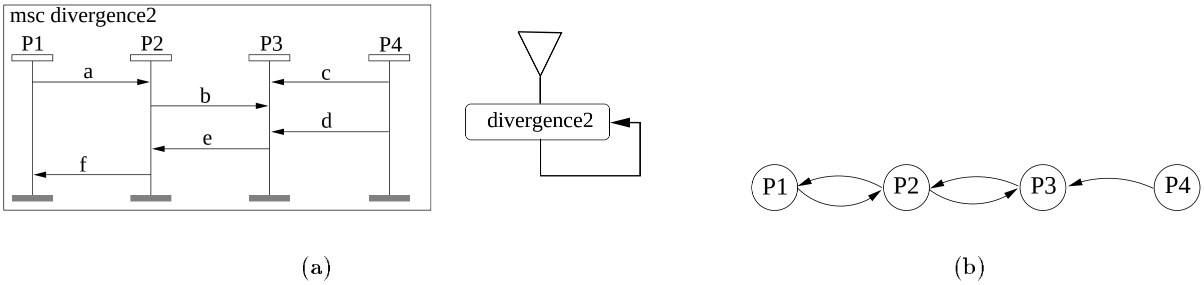


Figure 9: (a) MSC example divergence2; (b) its Coordination graph

In Figure 9 (a) it is process P4 that alone may race ahead of the other processes in the specification. Here again when we examine the communication pattern between the processes of this specification (Figure 9 (b)), we see that process P4 is not involved in any hand-shakes to coordinate its progress with other processes. On the other hand, the remaining processes coordinate their progress either directly (e.g., P1 and P2), or indirectly (e.g., P1 and P3 through P2).

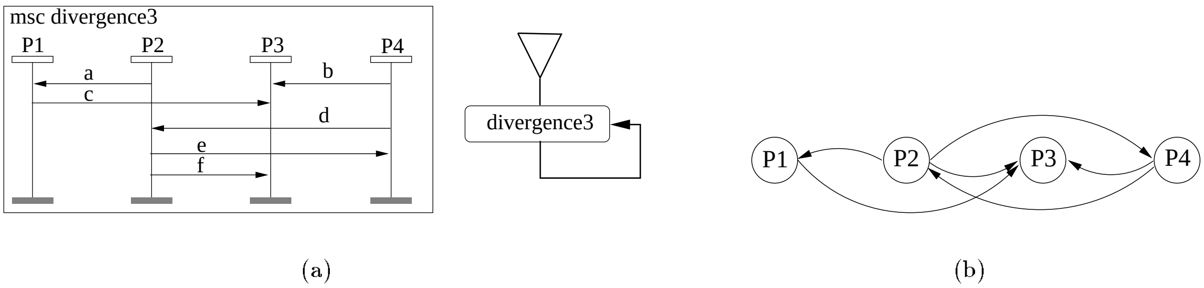


Figure 10: (a) MSC example divergence3; (b) its Coordination graph

Figure 10 presents another divergent MSC specification where divergence is not obvious. In this example, P2 and P4 may jointly race ahead of P1 and P3. Note again the communication pattern in this MSC specification: There is a message exchange in both directions between the processes P2 and P4, whereas for all other processes communication is only in one-way. The hand-shake between the processes P2 and P4 synchronize their progress with respect to each other. Such a synchronization is however lacking with the remaining processes.

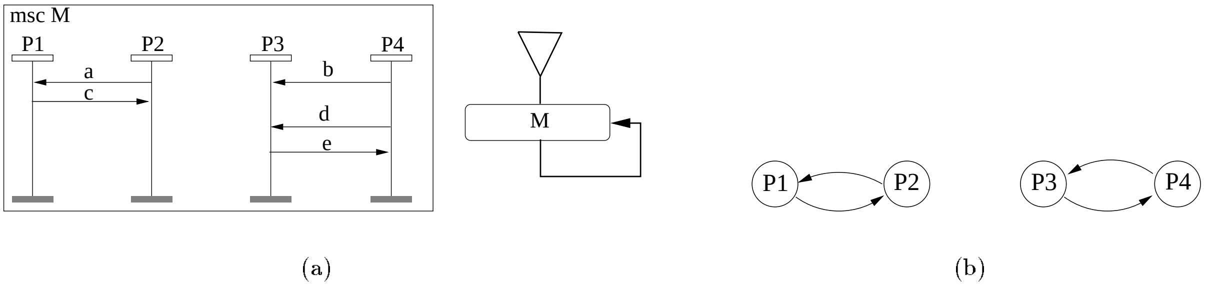


Figure 11: (a) an MSC specification with no process divergence; (b) its Coordination graph

Figure 11 shows an MSC specification where not all processes exchange messages yet the MSC specification does not suffer from process divergence. The graph representing the communication pattern of the MSC specification contains two connected components, each of which has processes that are involved in a hand-shake communication. Note that an implementation of this specification may allow any set of processes in a component to run faster than other processes in another component. However, since our notion of process divergence is defined in terms of messages sent, such an implementation does not exhibit the anomalous behavior where a message is sent an unbounded number of times without being received.

From the above examples, we can see that a two-way message exchange between two processes synchronizes their progress and eliminates the possibility that one races ahead of the other. In addition, such a message exchange need not be direct but can be through an intermediate process. Furthermore, the number of messages exchanged is irrelevant; one message can be enough to cause process divergence.

Definition 4.3 *The* coordination graph *of an MFG F is a directed graph $C_F = (PT_F, cor_F)$ where:*

- *PT_F is the set of nodes where each node corresponds to a process in the MFG F; and*
- *$cor_F \subseteq PT_F \times PT_F$ is the set of directed edges such that an edge is from P to Q if P sends a message to Q; formally:*

$$cor_F \triangleq \{(P, Q) \in PT_F \times PT_F \mid \exists (a, b) \in sig_F \, (ptype(a) = P \wedge ptype(b) = Q)\} \quad .$$

As the earlier examples illustrate, divergence in an MSC specification occurs through a loop of basic MSCs. Thus, our syntactic characterization of divergence focuses only on the bMSCs that are involved in a loop. A loop in an MSC specification $S = (B, V, suc, ref)$ is a sequence of nodes (i.e., bMFGs), $b_1, b_2, \cdots, b_n$, such that $(b_i, b_{i+1}) \in suc$ for $i = 1, \cdots n-1$ and $(b_n, b_1) \in suc$. A loop is called *simple* if all nodes are distinct except the first and last nodes which are identical.

In the sequel, we denote the transitive closure of a relation R as R^+ and its reflexive, transitive closure as R^*.

Theorem 4.1 *An MSC specification S is not divergent iff for each simple loop of basic MSCs, $M_1, M_2, \cdots, M_n, M_1$ in S, such that, for the corresponding MFGs F_i of M_i and coordination graphs $C_{F_i} = (PT_{F_i}, cor_{F_i})$, we have $\bigcup_{i=1}^{n} cor_{F_i}^{+}$ is symmetric.*

Algorithm. The algorithm gets an MSC specification S and returns `DIV_FREE` iff S is not divergent, and returns `DIVERGE` iff S is divergent. For each simple loop in S, the algorithm: 1) constructs the collective coordination relation of all bMSCs in the loop; 2) constructs the transitive closure of the relation; 3) checks if the relation is not symmetric in which case it returns `DIVERGE` and terminates. When all simple loops are examined, the algorithm returns `DIV_FREE`. In the next algorithm, we use `k` to denote the number of processes in each bMFG in S, the operation `OR` to denote the (boolean) disjunction operation over matrices, and we use `cor(M)` to denote the coordination relation of a bMFG `M`.

```
Begin
1.  For each simple loop L in S
2.     Let cor be a k by k matrix initialized with zeros
3.     For each bMFG M in L
4.        construct the coordination relation cor(M) of M
5.        cor = cor  OR  cor(M)
6.     If cor+ is not symmetric
7.     Then Return DIVERGE
8.  Return DIV_FREE
End
```

Let $S = (B, V, suc, ref)$ be the MSC specification to be analyzed. Finding all simple loops in S can be done through a modified DFS algorithm to find all strongly connected components in the directed graph (V, suc), e.g., Tarjan's algorithm [1] which runs in $O(max(|suc|, |V|))$. In the worst case, S can have $2^{|I|} - 1$ loops where $|I|$ is the total number of intermediate nodes in S. To construct the coordination graph of a bMFG an algorithm basically simplifies the relation sig which represents all message exchanges in the bMSC; thus step 4 runs in a worst time of $O(|sig|)$. In step 5, to update the `cor` relation, we need k^2 time units. To construct the transitive closure of a coordination relation, we can use the algorithm in [3] with the coordination relation representing the relation $\ll$ relation. This algorithm is a special case of the Floyd-Warshall algorithm and it runs in $O(k^2 + lk)$ time where l is an upper bound on the number of processes directly related in the coordination graph. In our case, l is bounded by $\sum_{\mathrm{M\ in\ L}} |sig_M|$, i.e., the number of messages in all the bMSCs in the loop L. To verify that the transitive closure of the coordination relation is symmetric takes $O(k^2)$ time. Thus, the overall worst case time of the above algorithm is $O(max(|suc|, |V|) + 2^{|I|}(\sum_{M\in B}(|sig_M| + k^2) + k^2 + k\sum_{M\in B}|sig_M| + k^2)) = O(2^{|I|}(|B|k^2 + k\sum_{M\in B}|sig_M|))$. In other words, the above algorithm is linear in the total number of messages in the MSC specification. This is efficient compared to examining potentially all executions of an MSC specification which can be exponential in the number of messages.

5 Non-local Branching Choice

An MSC specification can compose basic MSCs to express alternative behavior. Figure 5 illustrates an example which describes a system where `MSC1` is followed by either `MSC2` or `MSC3`. At this level of abstraction, all current interpretations assume that all processes choose the same alternative flow of control so that the overall system behavior is described by one basic MSC at a time. In terms of implementation of individual processes, such an assumption can however be non-trivial as it requires additional, dynamic information about which alternative other processes in the specification took. In terms of interpretation, this assumption may result in an infinite state space.

For example, consider the specification in Figure 5. Assume that, after executing the `Dreq` event, process `P1` is the first process to decide whether to go 'left', i.e. the next MSC to execute is `MSC1`. In order to implement properly the semantics of choice, the processes `P2` and `P3` must be informed about `P1`'s decision so that they branch accordingly. However, neither the MSC semantics as presented in Annex B of Z.120 [15] nor hMSC graphs provide an explicit way to handle such an information exchange. To handle this type of inter-process synchronization, Ladkin and Leue [18] suggested the use of global history variables that keep track of early process branching choices. Their approach, however, can result in an infinite-state semantic representation (i.e., global system transition graph) which can impede formal analysis.

However, not all branchings in an MSC specification require global history variables to keep track of early process branching choices. Consider for instance the MSC specification in Figure 4. In this example, the type of the first received message can be used to determine the choices made by other processes in the specification. Consider process `P3`; since sending messages is non-blocking, this process can decide to precede either as `MSC2` or `MSC3` independently of other processes. It can therefore either send message `CC` or `DR`, respectively, by making a local decision to resolve the non-determinism. On the other hand, since the first event in process `P2` is to receive either message from `P3`, process `P2` can learn about the decision that `P3` made based on the type of message it receives: if it receives a `CC` message, it knows that the `MSC2`-branch has been chosen and proceeds with sending a `Cind` to `P1`; otherwise it receives a `DR` message, knows that a branching to `MSC3` has occurred, and follows accordingly by sending a `Dind` to `P1`. Finally, process `P1` can also resolve the nondeterminism based on the type of message it receives from process `P2`. This strategy of *wait-and-see* can be easily implemented and eliminates the need for global history variables.

When the wait-and-see strategy can be used to resolve a non-determinism within each process, we call the branching as *local branching choice*. Otherwise, when explicit synchronization between the processes is necessary to resolve a non-determinism, we call the branching as *non-local branching choice*. We next formally define non-local branching choices and then syntactically characterize them.

5.1 Semantic Characterization of Non-local Branching Choice

Recall that a state in the GSTG contains a subset of: 1) next-event edges, and 2) signal edges that indicate an event was sent but not yet received. Also, as mentioned in Property 2.1, given a signal edge, we can trace its unique corresponding process in the bMSC via the bMFG. Thus, for each state in the GSTG, we can trace the processes and bMSCs to which they belong through the subset of signal edges in the state.

Given an MSC specification $S = (B, V, suc, ref)$ and its MFG $F = (S, C, ne, sig, ST, stype, ET, etype)$ with GSTG $G = (Q, q_0, T)$ and set of processes PT, we define the following three functions:

- $ptype : (S \cup C) \longrightarrow PT$ returns for each node in the MFG F the process to which the node belongs;

- *Snode* : $(S \cup C) \longrightarrow V$ returns for each node in the MFG F the corresponding node in the hMSC of S; and
- *Fnodes* : $Q \longrightarrow \mathcal{P}(S \cup C)$ returns for each state in the GSTG the set of MFG-nodes that correspond to all events enabled in the state.

The formal definitions of the above functions can be found or derived from auxiliary functions in [18].

Definition 5.1 *Let* $S = (B, V, suc, ref)$ *be an MSC specification with MFG* $F = (S, C, ne, sig, ST, \mathrm{stype}, ET, \mathrm{etype})$ *and GSTG* $G = (Q, q_0, T)$. S has a non-local branching choice *if there exists a finite sequence of transitions in* T*,*

$$q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} q_n$$

such that

$$\begin{aligned}\exists n_1, n_2 \in Fnodes(q_n)(\ & ptype(n_1) \neq ptype(n_2) \\ & \wedge \\ & \exists b \in V\, \exists b_1, b_2 \in range(\{b\} \lhd suc)\,(\, b_1 \neq b_2 \wedge \\ & \quad Snode(n_1) \in range(\{b_1\} \lhd suc^+) \wedge Snode(n_2) \in range(\{b_2\} \lhd suc^+)\,)\ )\end{aligned} \tag{1}$$

∎

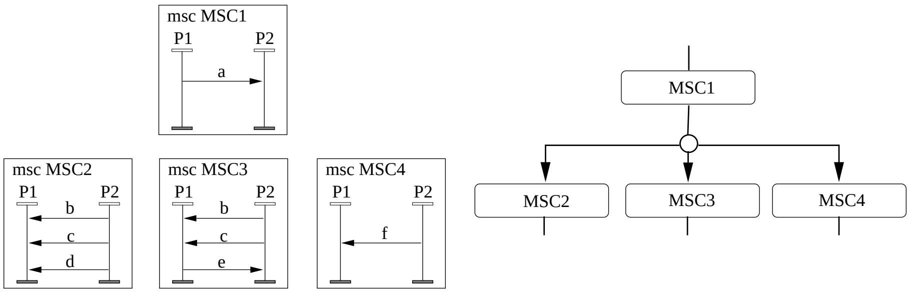
Informally, the condition (1) ensures that the reachable state q_n contains nodes from two processes in bMFGs that are reached by branching in different direction for each process.

5.2 Syntactic Characterization of Non-local Branching Choice

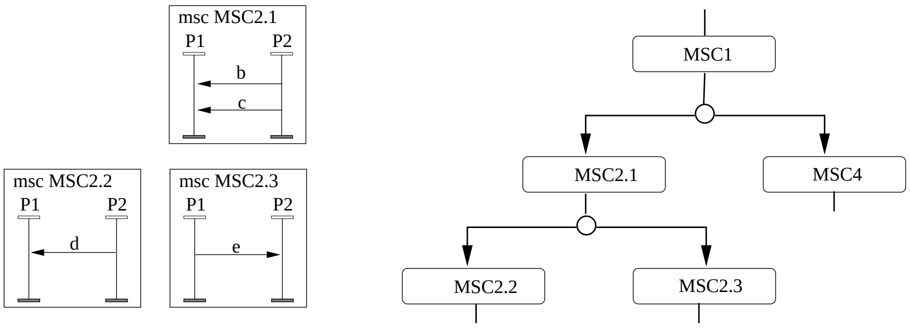
Our syntactic characterization of non-local branching choice relies on the "first" (according to the visual order) message exchanged in a bMSC. For this, we will assume in the remainder of this section that the MSC specification S to be analyzed satisfies the next two conditions:

1. S is *normalized*: for each branching node in S, the successor bMSCs do not have a common prefix of ordered sequence of message exchanges; and
2. each process in each bMSC in S exchange at least one message with other processes in the bMSC.

The first assumption is a minor deviation from the general syntax of MSC specifications in Z.120 [15]. On one hand, this assumption facilitates the interpretation of bMSC sequencing as a "weak sequencing" [15], and on the other hand it can be easily supported through a syntactic, pre-processing phase to our analysis. Figure 12 shows an example of normalization where the prefix common to the bMSCs `MSC2, MSC3` is arranged into a separate bMSC `MSC2.1` preceeding the remaining portions of `MSC2` and `MSC3`. The second assumption simplifies the computation of the first event in a sequence of bMFGs (i.e., bMSCs). However, this assumption can be eliminated by modifying the way we compute the first event in a sequence of bMFGs, i.e., bMSCs. As we see in Lemma 5.1, this



(a)



(b)

Figure 12: (a) part of an MSC specification not normalized; (b) its normalized version

assumption simplifies the semantic definition of an MSC specification with a non-local branching choice—Definition 5.1. A consequence of the second assumption is that each bMFG in S has a non-empty set of first events all of which are of type *send*.

To prove Theorem 5.1 we make use of the following Lemma.

Lemma 5.1 *Let $S = (B, V, suc, ref)$ be a normalized MSC specification with MFG $F = (S, C, ne, sig, ST, \text{stype}, ET, \text{etype})$ and GSTG $G = (Q, q_0, T)$.*
If each process in each bMFG in B exchanges at least one message, then we have: S has a non-local branching choice implies there exists a finite sequence of transitions in T,

$$q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} q_n$$

such that:

$$\begin{aligned}\exists n_1, n_2 \in Fnodes(q_n)(\ & ptype(n_1) \neq ptype(n_2) \\ & \wedge \\ & \exists b \in V\ \exists b_1, b_2 \in range(\{b\} \lhd suc)\ (\ b_1 \neq b_2 \wedge \\ & \quad Snode(n_1) = b_1 \wedge Snode(n_2) = b_2\ )\ )\end{aligned} \tag{2}$$

In the sequel, we use the following notation which is formally defined in the appendix. For a bMFG F, the partial order relation of F is $p_F = sig_F \cup ne_F$ and its set of first nodes (i.e., nodes from which an event can be sent first) is $firstnodes(p_F)$; for an MSC specification $S = (B, V, suc, ref)$, the set of nodes with a branching is $branchnodes(suc)$, and the set of nodes successors to a node n is $range(\{n\} \lhd suc)$.

Theorem 5.1 *Let $S = (B, V, suc, ref)$ be a normalized MSC specification where each process in each bMFG exchanges at least one message with another process.*

S has no non-local branching choice $\Longleftrightarrow$

$$\forall b \in branchnodes(suc)\ \Big|\bigcup_{c \in range(\{b\} \rhd suc)} \{ptype(n) | \, n \in firstnodes(p_{ref(c)})\}\ \Big| = 1 \tag{3}$$

Informally, an MSC specification S has no non-local branching choice iff at each of its branching points, the first events in all bMSCs are sent by the same process.

Algorithm. The algorithm first computes the *firstnodes*, i.e., first sender processes, of all bMFGs in the MSC specification to be analyzed. It then starts from each branching node in the MSC specification and verify that its successor bMFGs have a unique first sender process. If at any branching node the algorithm finds more than one first sender process, it stops and returns the flag NON_LOCAL to indicate that the specification has a non-local choice. If all branching nodes are successfully visited, the algorithm returns the flag LOCAL to indicate that all branchings can be resolved locally.

```
1. Begin
2. For each intermediate node c
3.    compute firstnodes(p(ref(c))
```

```
For each branching node b
   first_proc = NULL
   For each node c successor of b
      If ( |firstnodes(c)| != 1 )
         Return NON_LOCAL
      Else
         n = firstnode(c)
         If ( first_proc == NULL )
            first_proc = ptype(n)
         Else
            If ( first_proc != ptype(n) )
              Return NON_LOCAL
Return LOCAL
End
```

In the worst case, the above algorithm visits once all intermediate nodes, i.e., bMFGs in the MSC specification. To compute the set of first nodes in each bMFG F takes in the worst case $O(|S_F \cup C_F|)$ where $|S_F \cup C_F|$ is the number of nodes in the bMFG F. All remaining operations take a constant time. Thus, the above algorithm runs in the worst case in $O(\sum_{F \in B}(|S_F \cup C_F|)$, where B is the set of bMFGs in the MSC specification being analyzed. In other words, the algorithm runs in a time linear with the total number of messages exchanged in the MSC specification.

Removing Non-local Branching Choices

Once detected, a non-local branch choice can be easily removed from an MSC specification. For instance, consider the partial MSC specification in Figure 13 and assume that it only contains messages `a`, `b` and `c`. At the branch point, the bMSCs `MSC2` and `MSC3` have two different processes that execute the first send event; this violates the necessary condition for the absence of non-local branch choices. To eliminate the non-local branch choice, let us chose process `P1` to be the unique first sender in both bMSCs, and add the messages `x1` and `x2` to the specification as illustrated in Figure 13. The resulting branching has alternative bMSCs with a unique process that detremines the first event. Consequently, the remaining, receiving processes can determine which branch control much flow by using a wait-and-see strategy. The above strategy of resolving non-local branching choices can extended for specifications with more processes and branching alternatives.

Our syntactic analysis relieves a designer from the burden of explicitly coordinating the process branchings in an early design. Detecting and resolving non-local branching choices can be used as a refinement step of the design, in which a designer can introduce a coordination protocol, e.g., through additional messages.

Non-local Branching Choices and Deadlocks

The syntactic characterization of non-local branching choices can be used to generalize the syntactic characterization of deadlocks in an MSC specification through loop detection [19]. We conjecture the following.

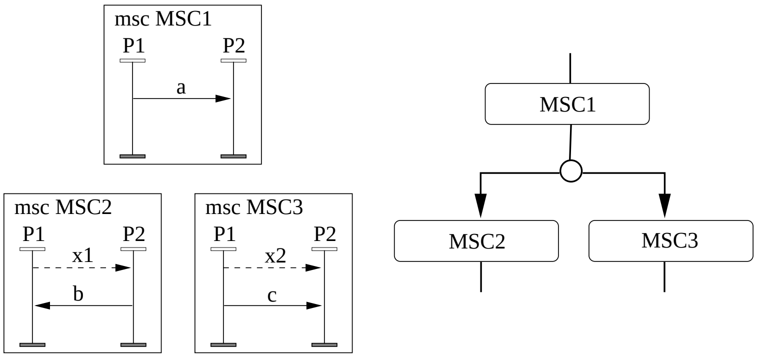


Figure 13: Removing non-local branching choice from an MSC Specification

Conjecture 5.1 *An MSC specification is deadlock-free if it has no non-local choices and each of its bMFGs has an acyclic* $sig \cup ne$ *relation.*

Non-local Branching Choices with Process Divergence

Recall that our notion of process divergence is characterized by an execution sequence where a message is sent an unbounded number of times more than it is received. A stronger notion of process divergence accounts for the speed of processes: process divergence occurs if at least one process can run faster than others in the MSC specification. For example, the MSC specification in Figure 11 has a process divergence according to this stronger notion. This notion of process divergence together with the presence of a non-local branching choice lead to an MSC interpretation with an infinite state space as it requires an infinite history variable. In the next conjecture, the notion of divergence is the stronger notion.

Conjecture 5.2 *An MSC specification that is divergent and has a non-local branching choice requires an interpretation with an infinite history variable and infinite-state space.*

As an example, the MSC specification in Figure 14 has a process divergence and a non-local branching choice. In this example, the loop of MSCs `MSC1`, `MSC3`, `MSC1` has a process divergence in the extended notion. In addition, the choice between MSC `MSC2` and `MSC3` is non-local since each MSC has two processes that can send first events. To interpret this MSC, we need an infinite history variable and therefore infinite-state space.

6 Current MSC Interpretations

In this section, we review current MSC interpretations and how they deal with process divergence and non-local branching choice.

Process algebra. The standard Z.120 Annex B [14] defines the formal semantics of basic MSCs via a translation to a customized version of the untimed process algebra ACP [5]. In this translation, sending and receiving messages, setting a timer, and timer expiration are interpreted as basic,

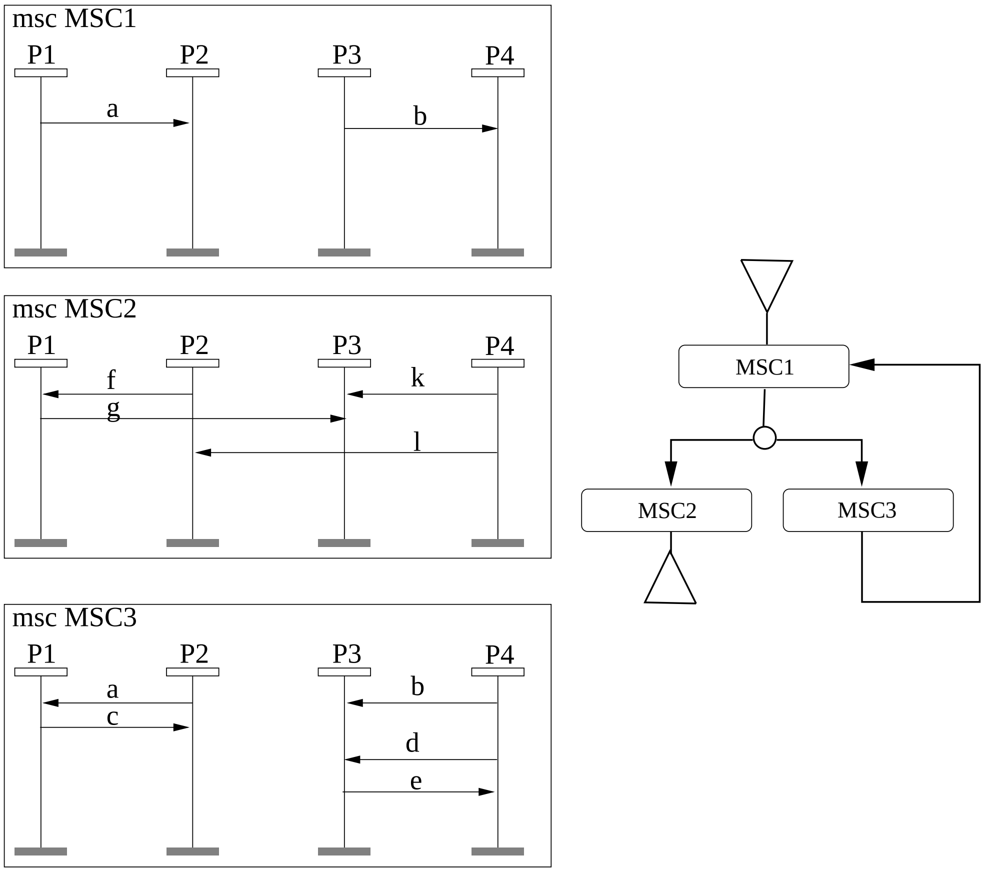
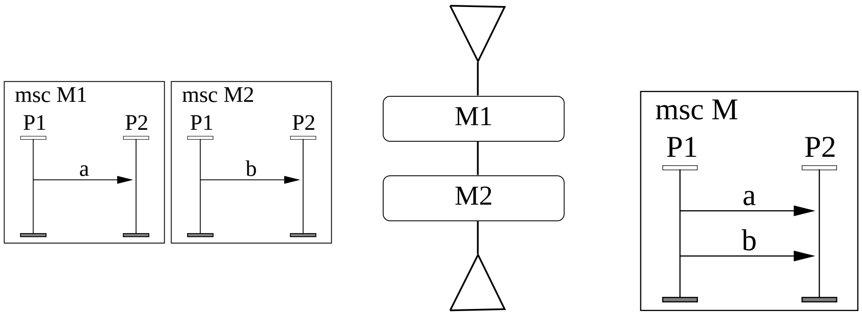


Figure 14: MSC Specification with process divergence and non-local branching choice

atomic *actions* in the algebra; conditions are interpreted as *empty* actions, i.e., actions without a meaning. Each process in a basic MSC is translated into a sequential, ACP *process* where ordered events are translated through the action-prefix operator and coregions (i.e., unordered events) are translated through the parallel operator into sub-processes in the sequential process.

A basic MSC is translated by first translating each of its processes, combining the resulting ACP processes through the parallel operator, and then applying an I/O filtering function to enforce the causal dependency between sending and receiving a message. The I/O filtering function is in fact applied on the expanded version of the parallel process where only prefix and choice operators are used. (The Z 120 Annex B [14] defines a set of axioms that allows the expansion. Mauw and Reniers [23] extend the axioms to a complete set with respect to bisimulation.) The I/O filtering function does not model any queuing strategy. In particular, multiple copies of a sent message is deactivated by one receive of the message.

Baeten and Mauw [4] augment the above standard semantics to deal with the non-deterministic and sequential compositions of basic MSCs. In this work, the authors use the ACP choice operator as well as the delayed choice operator to interpret, respectively, intended and unintended non-deterministic composition of basic MSCs. Informally, a non-determinism is *unintended* between two basic MSCs, if the two share a common prefix sequence of message exchanges. This translation however does not address iterative behavior (i.e., composition of basic MSCs in a loop) since the ACP version used does not have a recursion operator. In addition, the authors define the behavior of the sequential composition of two basic MSCs, `M1` and `M2`, as the behavior of the process `P1` that corresponds to `M1` followed by the process `P2` that corresponds to `M2`, i.e., the ACP process `P1.P2`. This semantics, however, differs from the informal semantics of sequential composition as presented in Z 120 [15].



(a) (b)

Figure 15: An hMSC specification (a) and a basic MSC (b) with the same behavior

Consider the hMSC specification of Figure 15 (a). The semantic interpretation of the sequential composition as presented in [4] produces the following process

$$\begin{aligned} P &= \lambda_\emptyset(!a \mid\mid ?a) \,.\, \lambda_\emptyset(!b \mid\mid ?b) \\ &= !a \,.\, ?a \,.\, !b \,.\, ?b \end{aligned}$$

where the function λ_M is the I/O filtering function. However, the informal, standard semantics interprets sequential composition of basic MSCs as "the weak sequencing operation where only events on the same instance [process] are ordered." [15] In other words, the hMSC must be interpreted in the same way as the basic MSC `M` of Figure 15 (b); that is, its behavior is described by the process

$$\begin{aligned} P' &= \lambda_\emptyset((!a \,.\, !b) \mid\mid (?a \,.\, ?c)) \\ &= !a \,.\, ?a \,.\, !b \,.\, ?b + !a \,.\, !b \,.\, ?a \,.\, ?b \end{aligned}$$

The two interpretations differ in the amount of interleavings: in the process P', sending `b` can interleave sending and receiving `a`. However, this behavior is not allowed in the process P. The interpretation of sequential composition as presented in [4] does not deal with process divergence as it only operate on finite composition of basic MSCs; even when extended to iterating high-level MSCs, it will not address process divergence nor will it address non-local branching as it uses a strong notion of sequential composition, one where all processes in an MSC must finish their execution before proceding to the next MSC.

Message Flow Graphs and Automata. Ladkin and Leue [18] propose a finte-state, interleaved semantics for MSC specifications. An MSC specification is interpreted in two steps: first, the MSC specification is translated into an isomorphic algebraic representation, called *Message Flow Graphs* (MFGs); second, the global system states and execution steps of the MSC are defined through a labeled transition system called Global State Transition Graph (GSTG). The acceptable execution sequences (or communication event traces) of an MSC are obtained by unfolding the GSTG. In order to express underspecified liveness properties the GSTS is then augmented with a notion of end states to obtain a Büchi automaton. The authors present two ways to define an end state: by inspection and by augmenting the GSTG with temporal logic formulas.

The semantics definition in [18] avoids neither process divergence nor non-local branching choice. However, [18] explains how the semantics can be upgraded by a history variable that resolves non-local branching choice situations.

Ordering relations. Most tools that support MSC specifications interpret them through relations that order the events in a bMSC. The ordering relation can be either a total ordering, or a partial ordering. A total ordering is derived from the visual order including the vertical distances from the start of the processes; e.g., see [2]. However, this can lead to ambiguities in the case of events that are at an equal vertical distance from the start of processes. A partial ordering overcomes such ambiguity. In [3], Alur et al incorporate queuing architecture and protocol within partial orderings. The resulting partial orderings are compared with the visual partial ordering to detect race conditions, i.e., orderings of events that differ from the visual ordering.

Note. Currently none of the available interpretations of MSC specifications is based on a timed model. All interpretations either ignore timers, or treat them as a special class of events. In addition, as we illustrated in this paper the semantics of an MSC is affected by the network architecture, capacities of queues, and queuing strategies. These resources are in particular critical in the case of iterating MSCs. None of the available interpretations of MSCs account for all these resources. The only possible, indirect way to account for these resources is through the interpretation of MSCs via a translation into Promela.

Promela. Leue and Ladkin [22] recently explored the interpretation of MSCs via a translation to Promela [10]. An MSC specification is translated into Promela as follows: processes in the MSC specification are mapped into concurrent Promela processes; each message arrow is mapped into a communication channel (FIFO-queue) with a capacity = 1; and each communication event in the MSC specification is translated into a Promela communication statement with a special care to ensure the event atomicity. Branchings and iterations in the MSC specification are modeled through labels and goto statements in Promela. The MSC to Promela translation in [22] describes how to implement a history variable-based synchronization algorithm to execute an MSC specification with non-local choices. It also illustrates how to introduce channels with capacities greater than 1. This translation makes the send event blocking if a queue is full; this is a deviation from the non-blocking send in MSCs. An overflow of a queue is therefore detected as a deadlock in the Promela specification.

7 Conclusion

We have highlighted two potential problems in MSC specifications that are due to implicit assumptions about the environment behavior. Both problems can lead to interpretations with an infinite state space, discrepancies between a specification and its implementation, as well as unimplementable specifications. One problem, process divergence, is the result of iterating basic MSCs and implicit assumptions about the queuing mechanism between communicating processes. It leads to a specification where one or more processes run faster than others flooding them with multiple copies of messages that they may not receive. The second problem, non-local branching choice, appears in MSC specifications where basic MSCs can be executed in an alternative way. It results in MSC specifications that are either unimplementable or implemented with unintented deadlocks. We have semantically defined the above two problems and syntactically characterized them. We also have proposed detection algorithms that run in an order linear in the total number of messages exchanged in the MSC specification being analyzed.

We are currently implementing a tool for the requirements and design phases of reactive systems based on MSC specifications. One major functionality of the tool is static analysis of MSC specifications, that incorporates the presented algorithms in addition to others, e.g., deadlock detection [19]. We are also examining how to augment our semantics and analysis techniques to support *actions* and timing assumptions.

Acknowledgements

This work was in part supported by the Information Technology Research Centre of the Province of Ontario and by the National Science and Engineering Research Council of Canada. ObjecTime Limited provided further support. Peter Ladkin suggested the proof sketch for Conjecture 3.1.

References

[1] A. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and ANalysis of Computer Algorithms*, chapter 5. Addison-Wesly Publishing Company, 1974.

[2] 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.

[3] 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.

[4] J.C.M. Baeten and S. Mauw. Delayed choice: an operator for joining message sequence charts. In *Proceedings of the 7th International Conference on Formal Description Techniques (FORTE'94)*, pages 327–341, October 4-7 1994.

[5] J. A. Bergstra and J. W. Klop. Algebra of Communicating Processes. Technical Report CS-R8420, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, 1984.

[6] Grady Booch and Jim Rumbaugh. *Unified Method: User Guide Version 0.8.* RATIONAL Software Corporation, 1995.

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

[8] CCITT. Recommendation Z.100: CCITT Specification and Description Language (SDL). CCITT, Geneva, 1992. (CCITT has been renamed ITU-T).

[9] P. Graubmann, E. Rudolph, and J. Grabowski. Towards a Petri Net based semantics definitions for message sequence charts. In O. Faergemand and A. Sarma, editors, *Proceedings of the 6th SDL Forum, SDL'93: Using Objects*, October 1993.

[10] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.

[11] G. J. Holzmann. Early fault detection tools. 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 1–13. Springer Verlag, 1996.

[12] 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.

[13] ISO. Information processing systems - open systems interconnection - LOTOS : A formal description technique based on temporal ordering of observed behavior, international standard 8807. International Standards Organisation, 1988.

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

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

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

[17] P. B. Ladkin and S. Leue. An automaton interpretation of message sequence charts. Technical Report IAM 92-012, University of Berne, Institute for Informatics and Applied Mathematics, 1992.

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

[19] P. B. Ladkin and B. B. Simons. Static analysis of communicating processes. To appear, Springer Lecture Notes in Computer Science.

[20] S. Leue. *Methods and Semantics for Telecommunications Systems Engineering.* Doctoral dissertation, University of Berne, Switzerland, December 1994.

[21] S. Leue and P. B. Ladkin. Implementing and verifying scenario-based specifications using Promela/XSpin. In *Participants Proceedings of the 2nd International Workshop on the SPIN Verification System*, pages 129–146. DIMACS/Bell Labs/INRS-Télécommunications, 1996.

[22] S. Leue and P. B. Ladkin. Implementing and verifying scenario-based specifications using Promela/XSpin. In J.-C. Grégoire, G. Holzmann, and D. Peled, editors, *Proceedings of the DIMACS Workshop SPIN96, the 2nd International Workshop on the SPIN Verification System, Rutgers University, August 1996.* American Mathematical Society, 1997, to appear.

[23] S. Mauw and M.A. Reniers. An algbraic semantics of basic message sequence charts. *The Computer Journal*, 37(4), 1994.

[24] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J.R.W. Smith. *Systems Engineering Using SDL-92.* Elsevier Science B.V. (North-Holland), 1994.

[25] M.A. Reniers. Syntax requirements of message sequence charts. In R. Braek and A. Sarma, editors, *Proceedings of the 7th SDL Forum*, pages 63–74. Elsevier Science Publishers B.V., October 1995.

[26] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International, 1991.

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

A Notation and Definitions

Relations. Let $f \subseteq R \times R$ denote a binary relation over a set R, let $x, y \in R$ and S a set. We define the following *restrictions* and *operators* on a relation f.

$$f \rhd S \triangleq \{(a,b) | (a,b) \in f \ \wedge \ b \in S\} \qquad domain(f) \triangleq \{a \mid (\exists b \in R)((a,b) \in f)\}$$
$$S \lhd f \triangleq \{(a,b) | (a,b) \in f \ \wedge \ a \in S\} \qquad range(f) \triangleq \{b \mid (\exists a \in R)((a,b) \in f)\}$$

Closure Operators. Let $f, g \subseteq R \times R$ denote binary relations over a set R. The *composition* of f and g is $f \circ g \triangleq \{(a,c) \mid (\exists b)((a,b) \in f \wedge (b,c) \in g)\}$. The n-th power of f (*iterated composition*) is defined recursively by $f^1 \triangleq f$ and $f^{n+1} \triangleq f^n \circ f$. The *transitive closure* f^+ is defined as $f^+ \triangleq \bigcup_{n>0} f^n$.

Digraphs. Let V denote a set and let $E \subseteq V \times V$, then we call $T = (V, E)$ a *digraph*. $(V, E, type, labels)$ is a *digraph with node labels* iff $E \subseteq V \times V$, $type : V \rightarrow labels$, and $labels = range(type)$. $(V, E, type, labels)$ is a *digraph with edge labels* iff $E \subseteq V \times V$, $type : E \rightarrow labels$, and $labels = range(type)$. For a digraph $T = (V, E)$ we define: $branchnodes(E) \triangleq \{v \in V \mid (| \{v\} \lhd E |) > 1\}$.

Message Flow Graphs. Let S and C denote two arbitrary disjoint sets, the elements of which we call *sending* events and *receiving* events, respectively. Furthermore, let ST and ET denote arbitrary disjoint sets (also disjoint from S and C), whose elements we call *signal* and *event* types. We define a *Message Flow Graph* as a tuple $\mathcal{G} = (S, C, ne, sig, ST, stype, ET, etype)$ where $(S \cup C, ne, etype, ET)$ is a digraph with node labels and $(S \cup C, sig, stype, ST)$ is a digraph with edge labels satisfying the following conditions:

1. $sig \subseteq S \times C$ is a (necessarily bipartite) bijective relation, where $S = domain(sig)$ and $C = range(sig)$;

2. The set $ET = (\{!, ?\} \times ST)$ contains the *event types* (we write $!t$ for $(!, t)$ and $?t$ for $(?, t)$).

3. If the type of a signal is t, then the corresponding send and receive events are of type $!t$ and $?t$ respectively: $(a,b) \in sig \rightarrow (\exists t \in ST)(stype((a,b)) = t \ \wedge \ etype(a) = !t \wedge etype(b) = ?t)$;

4. Every component of the ne relation graph contains at most one start event:

$$(e, e' \notin range(ne) \wedge (e, e') \in ne^*) \rightarrow (e = e').$$

We denote the partial order *precedence relation* of the MFG $\mathcal{G}$ as $p_{\mathcal{G}} \triangleq sig \cup ne$, and the *first* nodes in $\mathcal{G}$ according to $p_{\mathcal{G}}$ as $firstnodes(p_{\mathcal{G}}) = \{e \in S \mid (p_{\mathcal{G}} \rhd \{e\}) = \emptyset\}$, that is the set of nodes from which a first event can be sent.

B Proofs

Proof of Lemma 4.1. We want to prove that $\mathcal{F}$ is divergent if there exist $< x, y > \in sig$ and an infinite sequence of transitions in T_F

$$q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \cdots$$

such that for $s = \alpha_0\alpha_1\alpha_2\cdots$, we have

$$\exists n \in \mathbb{N} \forall m \geq n \quad \#_s(x, m) > 1 + \#_s(y, m) \quad .$$

Proof: Let $< x, y > \in sig$, and let the infinite sequence of transitions in T_F

$$q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \cdots$$

be such that for $s = \alpha_0\alpha_1\cdots$ and $n \in \mathbb{N}$ we have $\forall m \geq n \quad \#_s(x, m) > 1 + \#_s(y, m) \quad .$

Since $\mathcal{G}_F$ is finite-state, then the above infinite sequence of transitions comes from a loop in T_F. Let $j, k \in \mathbb{N}$ be such that

1. $n \leq j \leq k$ and
2. $q_0 \xrightarrow{\alpha_0} \cdots \xrightarrow{\alpha_{j-1}} q_j \xrightarrow{\alpha_j} \cdots \xrightarrow{\alpha_{k-1}} q_k \xrightarrow{\alpha_k} q_j$ and
3. for $s' = \alpha_j \cdots \alpha_k$ we have $\#_{s'}(x, k-j+1) \geq 1 + \#_{s'}(y, k-j+1)$.

The existence of j and k such that condition 2 is satisfied is straightforward. We will shortly argue condition 3. First note that the semantics ensures that the following sequence is also in T_F:

$$q_0 \xrightarrow{\alpha_0} \cdots \xrightarrow{\alpha_{j-1}} q_j \xrightarrow{\alpha_j} \cdots \xrightarrow{\alpha_{k-1}} q_k \xrightarrow{\alpha_k} q_j \xrightarrow{\alpha_j} \cdots \xrightarrow{\alpha_{k-1}} q_k \xrightarrow{\alpha_k} q_j \xrightarrow{\alpha_j} \cdots$$

that is, the execution obtained by traversing the loop an infinite number of times is a trace of $\mathcal{F}$. Let $s'' = \alpha_0 \cdots \alpha_{j-1} s' s' \cdots$; it is easy to see that since s'' contains an infinite number of copies of s' we have

$$\forall n \in \mathbb{N} \exists m \in \mathbb{N} \; \#_{s''}(x, m) \geq n + \#_{s''}(y, n) \quad .$$

Which by Definition 4.1 implies that $\mathcal{F}$ is divergent, and we are done.

We just need to prove there is a loop of transitions where condition 3 is satisfied. Assume that for all possible loops (i.e. for all $n \leq j \leq k$) in the original infinite sequence, we have

$$\#_{s'}(x, k-j+1) \leq \#_{s'}(y, k-j+1) \quad .$$

Let $s_1 = \alpha_0 \cdots \alpha_{j-1}$ as defined by the original infinite sequence of transitions. Now, by hypothesis and since $n \leq j \leq k$, we have

$$\#_{s_1 s'}(x, k) > 1 + \#_{s_1 s'}(y, k)$$

from which we can infer that

$$\#_{s_1}(x, j) + \#_{s'}(x, k-j+1) > 1 + \#_{s_1}(y, j) + \#_{s'}(y, k-j+1) \quad .$$

Also, by hypothesis we have

$$\#_{s_1}(x, j) > 1 + \#_{s_1}(y, j) \quad .$$

Thus, we get

$$\#_{s'}(x, k-j+1) > \#_{s'}(y, k-j+1)$$

which is a contradiction. ∎

Proof of Theorem 4.1. Note that the semantics of the sequential composition of a finite sequence of bMSCs, $M_1, M_2, \cdots, M_n$, corresponds to the semantics of a bMSC that results from "glueing" the bMSC M_1, followed by M_2, etc, all the way to M_n. Based on this note, in the proof of this theorem, when we quatify over the existence of a simple loop of bMSCs $M_1, M_2, \cdots, M_n, M_1$, we will use the bMSC M to denote the sequence of bMSCs, $M_1, M_2, \cdots, M_n$. Note also that it is straightforward to prove that the MFG of the MSC specification S has a subgraph F_M which is (graphically) isomorphic to the MFG of M.

Let S be an MSC specification with MFG $F = (S, C, ne, sig, ST, stype, ET, etype)$ and its (finite-state) GSTG be $G = (Q, q_0, T)$.

In the sequel, we use F_M to denote the bMFG from F which corresponds to M which is described above; $C_M = (PT_M, cor_M)$ the coordination graph of the F_M, and $G_M = (Q_M, q_{0_M}, T_M)$ the GSTG of F_M. Note that G_M is a subgraph of G.

If: We proceed by contradiction: Assume S is divergent and that for all simple loops of bMSCs $M_1, \cdots, M_n, M_1$ in S we have cor_M^+ is symmetric where M is the bMSC corresponding to $M_1, \cdots, M_n$.

By Lemma 4.1, if S is divergent then F has an edge $< x, y > \in sig$ and an infinite sequence of transitions

$$q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \cdots$$

such that for $s = \alpha_0\alpha_1\alpha_2\cdots$, we have

$$\exists n \in \mathbb{N} \forall m \geq n \quad \#_s(x, m) > 1 + \#_s(y, m) \quad .$$

Since G is finite-state, then any infinite sequence comes from a loop in F, and thus S. Assume that the loop comes through a bMSC M corresponding to a simple loop in S. Thus, such a bMSC M has at least two processes $P_x, P_y \in PT_M$ such that $(P_x, P_y) \in cor_M$. Since cor_M^+ is symmetric, then $(P_y, P_x) \in cor_M^+$. This can be the result of two cases: 1) P_x directly depends on P_y; or 2) P_x transitively depends on P_y.
We can use induction on the number of intermediate processes between P_x and P_y to prove in both cases we reach a contradiction. We next show the cases of zero and one intermediate process.

Case 1: $(P_y, P_x) \in cor_M$. In this case, each such bMSC M has one of the four possible segments shown in Figure 16 where $x = !a$ and $y = ?a$ and we eliminated other events for simplicity.

In the case (1), the states of the GSTG of M belong to one of the following five disjoint classes:

1. Class $G_1 = \{G_{1i} \in Q | \, (*, !a) \in G_{1i} \wedge <!b, ?b> \notin G_{1i}\}$
2. Class $G_2 = \{G_{2i} \in Q | \, <!a, ?a> \in G_{2i} \wedge (*, !b) \notin G_{2i}\}$
3. Class $G_3 = \{G_{3i} \in Q | \, (*, !b) \in G_{3i} \wedge <!a, ?a> \notin G_{3i}\}$
4. Class $G_4 = \{G_{4i} \in Q | \, <!b, ?b> \in G_{4i} \wedge (*, !a) \notin G_{4i}\}$
5. Class $G_5 = \{G_{5i} \in Q | \, (*, !a) \notin G_{5i} \wedge (*, !b) \notin G_{5i} \wedge <!a, ?a> \notin G_{5i} \wedge <!b, ?b> \notin G_{5i}\}$

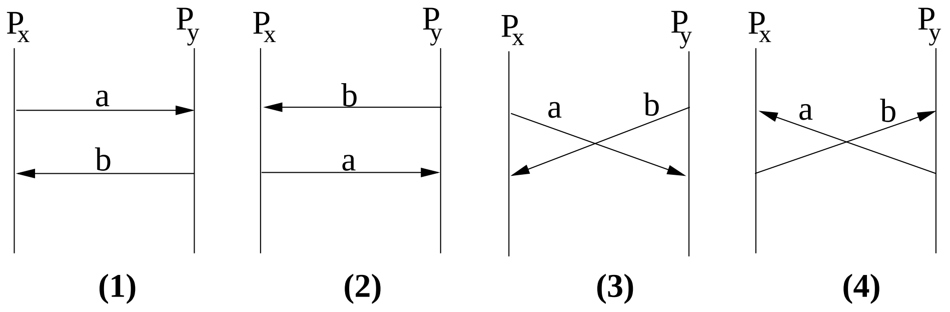


Figure 16: Possible segments in M for case 1

In addition, any sequence of transitions in the GSTG of M with states from the first four classes must visit states in the following order:

$$G_{1i} \xrightarrow{!a} G_{2j} \xrightarrow{?a} G_{3k} \xrightarrow{!b} G_{4l} \xrightarrow{?b} G_{1m}$$

possibly interspersed by transitions connecting states from G_5. Furthermore, it is straightforward to see that the above four classes of states satisfy the following properties:

1. any sequence of transitions ending at a state from G_1 has $\#!a = \#?a$;
2. any sequence of transitions ending at a state from G_2 has $\#!a = \#?a + 1$;
3. any sequence of transitions ending at a state from G_3 has $\#!a = \#?a$; and
4. any sequence of transitions ending at a state from G_4 has $\#?a = \#!a$.

Thus, we can conclude that any sequence of transitions in the GSTG of S satisfies the following property: $\#?a \leq \#!a \leq 1 + \#?a$. From this we can infer that a is not involved in the divergence, which is a contradiction.

We can reason in a similar fashion in the cases (2) and (3) of Figure 16 to prove that each trace in the GSTG of M has a number of $!a$ that is bounded by the number of $?a$, which contradicts the assumption that a is involved in the divergence. In the case (4), the two processes P_x and P_y are deadlocked right at the point where they need to receive a and b, respectively. Thus the event a cannot be involved in the divergence– again a contradiction.

Case 2: There exists a process P' such that $(P_y, P') \in cor_M$ and $(P', P_x) \in cor_M$. In this case, each bMSC M can have one of the ten segments shown in Figure 17; again other messages in M are omitted from Figure 17 for simplicity.

In case (7) the event a will never be sent, since the three processes P_x, P_y and P deadlock right before receiving b, a and d, respectively. In the remaining cases of Figure 17, we can use similar reasoning to Case 1 to show that the number of times a is sent is bounded by the number of times it can be received, i.e., a cannot be involved in a divergence, which leads to a contradiction

Only if: Assume that S is divergence-free and that there exists a simple loop $M_1, \cdots, M_n, M_1$ in S such that cor^+_M is not symmetric, where M is the bMSC corresponding to $M_1, \cdots, M_n$. Then

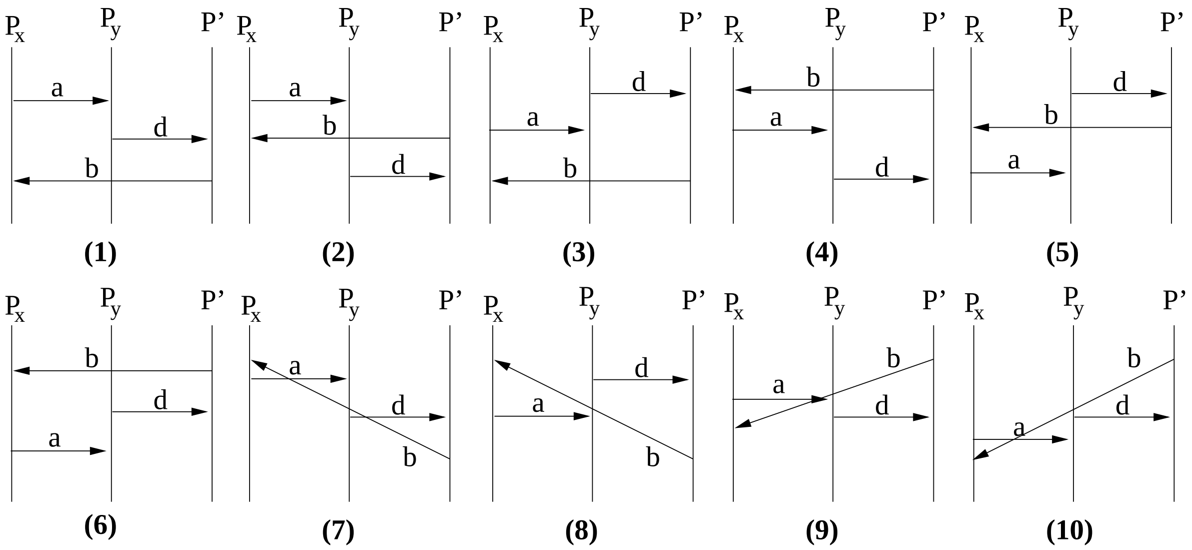


Figure 17: Possible segments in M for case 2

such a loop (i.e. corresponding bMSC M) has two processes P, P' such that $(P, P') \in cor_M^+$ but $(P', P) \notin cor_M^+$. Thus, for any process P'' of M, if $(P', P'') \in cor_M^+$ then $(P'', P) \notin cor_M$. In other words, every process P'' that (transitively) depends on P' can not send messages to P.

We can therefore infer that the set of processes in the bMSC M can be partitioned into three disjoint sets: 1) a set of processes that communicate with P and among one another but not with P'; 2) a set of processes that communicate with P' and among one another but not with P; and 3) a set of processes that communicate with neither P nor P'.

In addition, M has at least one communication from P to P'. W.l.o.g. assume that there is only one communication from P to P' through the signal edge $< x, y >$. Then, the states of the GSTG of S can be divided into eight classes depending on the position of control in P and P' with respect to nodes x and y. In particular, we list the following three classes:

1. G_1 set of states such that control in P is before node x, control in P' is before node y;
2. G_2 set of states such that control in P is at node x, control in P' is before node y; and
3. G_3 set of states such that control in P is after node x, and control in P' is before node y;

In addition, the following sequence of transitions are in T:

$$\begin{aligned} G_{1i} &\xrightarrow{\alpha_0} G_{1i'} \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} G_{1i''} \\ &\xrightarrow{\alpha_n} G_{2j} \xrightarrow{x} G_{3k} \\ &\xrightarrow{\beta_0} G_{3k'} \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_{l-1}} G_{3k''} \xrightarrow{\beta_l} G_{1m} \end{aligned}$$

where each of the $\alpha_0, \cdots \alpha_n$ and $\beta_0, \cdots \beta_l$ is distinct from x and from y. The above sequence of transitions allowed P to progress in its other communications, send x to P', and then get back to a point where it can send x again. All of this while P' remained at a point before reaching the node y where it can receive the message x. This sequence is possible due to three facts: 1) there is no deadlock in F; 2) M is involved in a loop; and 3) P's progress, i.e., communications other than with P' are independent of those of P'. This is a consequence of the assumption that there is no indirect dependency between P and P' as the cor_M^+ relation indicates.

Now, it is clear that the above sequence of transitions can produce an infinite trace where x is repeated an unbounded number of times without seeing a y by looping through states from G_1; thus, S is divergent which is a contradiction. ∎

Proof of Lemma 5.1. The proof is by contradiction. Assume S has a non-local branching choice, however, for any reachable state q_n and any branching in S, q_n does not satisfy the condition (2); that is, we have

$$\begin{array}{l}\forall q_1, \cdots, q_n \in Q\, \forall \alpha_1, \cdots, \alpha_n \in ST(q_0 \xrightarrow{\alpha_1} q_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} q_n) \\ \Longrightarrow \\ \quad \forall b \in V\, \forall b_1, b_2 \in range(\{b\} \lhd suc)(\ \{b_1, b_2\} \not\subseteq \{Snodes(n) |\, n \in Fnodes(q_n)\} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall n_1, n_2 \in Fnodes(q_n)\, (ptype(n_1) = ptype(n_2))\ )\end{array}$$

The above property can be the result of two cases:
Case 1. $\{b_1, b_2\} \not\subseteq \{Snodes(n) |\, n \in Fnodes(q_n)\}$ for all reachable states q_n and branching nodes b. This results from one of three subcases:

1. $\{b_1, b_2\} \cap \{Snodes(n) |\, n \in Fnodes(q_n)\} = \{b_1\}$;
2. $\{b_1, b_2\} \cap \{Snodes(n) |\, n \in Fnodes(q_n)\} = \{b_2\}$; or
3. $\{b_1, b_2\} \cap \{Snodes(n) |\, n \in Fnodes(q_n)\} = \emptyset$.

The subcase 1 implies that the bMFG in node b_2 does not contribute to any behavior of S. Since we are quantifying over all reachable states and branching nodes b and successors of b, this implies that either (1) $sig_{ref(b_2)} \cup ne_{ref(b_2)}$ is cyclic which contradicts an assumption about the bMFGs of S; or (2) $sig_{ref(b_2)} = \emptyset$ which contradicts the assumption that each bMFG in S has processes that exchange at least one message. The subcases 2 and 3 can be argued in a way similar to subcase 1.
Case 2. Assume that $\{b_1, b_2\} \subseteq \{Snodes(n) |\, n \in Fnodes(q_n)\}$, but $\forall n_1, n_2 \in Fnodes(q_n)\, (ptype(n_1) = ptype(n_2))$. This implies that S has no non-local branching choice as per Definition 5.1, which is a contradiction. ∎

Proof of Theorem 5.1.

⟹: Let the MSC specification $S = (B, V, suc, ref)$ be such that S has no non-local branching choice but

$$\forall b \in branchnodes(suc)\ |\bigcup_{c \in range(\{b\} \rhd suc)} \{ptype(n) |\, n \in firstnodes(p_{ref(c)})\}\ | > 1 \tag{5}$$

The condition (5) can be due to the following cases:

Case 1. S has a branching node $b \in branchnodes(suc)$ such that one of its successors $b' \in range(\{b\} \triangleright suc)$ has a bMFG with at least two first nodes, i.e., two processes that can send first. Let P_1 and P_2 be two processes in the bMFG referenced by b' and that can first send events. Now there are two subcases:

1. P_1 or P_2 is not the first sender in another bMFG referenced by a node successor of b. Assume that P_2 is not the first sender in the bMFG of $b'' \in range(\{b\} \triangleright suc)$ but P_1 is a first sender in the bMFG of b''. In this case, there is a state q in the GSTG of S where P_1 sends its first event in b' and P_2 branches to wait on receiving an event in b''.
2. both P_1 and P_2 are first senders in all bMFGs referenced by successors of b. Since we assumed that S is normalized, then P_1 and P_2 must send different first events in all these bMFGs. In this case, the GSTG of S has a state q where P_1 sends its first event in b' and P_2 sends its first event in another successor of b.

Case 2. S has a branching node $b \in branchnodes(suc)$ such that (at least) two of its successors, say $b_1, b_2 \in range(\{b\} \triangleright suc)$, reference bMFGs with two distinct processes that can send first. Let P_1 be the process to send the first event in the bMFG referenced by b_1 and let $P_2 \neq P_1$ be the process to send the first event in the bMFG referenced by b_2. Since P_1 does not send first in the bMFG of b_2, then it must first receive an event. Thus, P_1 have no way to resolve the choice between sending an event as described in the bMFG of b_1 or receiving an event as described in the bMFG of b_2. Similar comment holds for P_2. Thus, the GSTG of S has a state q where P_1 sends its first event in the bMFG referenced by b_1 and P_2 sends its first event in the bMFG referenced by b_2.

In all the above cases, the state q satisfies condition (1) in Definition 5.1. Also since we assumed that all bMFGs in S have acyclic preorder relations, such a state is reachable from the start state of the GSTG of S. This contradicts the assumption that S has no non-local branching choice.

$\Longleftarrow$: Let $S = (B, V, suc, ref)$ be normalized with each process exchanges at least one message. Assume that condition (3) holds for S but S has a non-local branching choice.

Now, S has a non-local branching choice implies, by Lemma 5.1, that there is a reachable state q_n in the GSTG of S such that condition (2) holds. This implies that there exists a branching node $b \in branchnodes(suc)$ such that

$$\mid \bigcup_{c \in range(\{b\} \triangleright suc)} \{ptype(n) \mid n \in firstnodes(p_{ref(c)})\} \mid \geq 2$$

which contradicts condition (3). Thus, S has no non-local branching choice. ∎