Syntactic Detection of Process Divergence and Non-local Choice in Message Sequence Charts*

Hanêne Ben-Abdallah and Stefan Leue

Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
hanene|sleue@swen.uwaterloo.ca

Abstract. Message Sequence Charts (MSCs) are increasingly used in software engineering methodologies and tools to capture, for instance, system requirements, test scenarios, and simulation traces. They have been standardized by ITU-T in Recommendation Z.120 [IT96]. 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.

1 Introduction

The intuitive, graphical notation of Message Sequence Charts (MSCs) increased their popularity within the software engineering community. They have already been adopted within several software engineering methodologies and tools for concurrent, reactive and real-time systems. They are used to document system requirements that guide the system design (e.g., [SGW94]), describe test cases and scenarios (e.g., [Jea92,BR95]), express system properties that are verified against SDL specifications (e.g., [ALHH93]), visualize sample behavior of a simulated system specification (e.g., [SGW94,ALHH93]), and to express legacy specifications in an intermediate representation that helps in software maintenance and reengineering [IIK+91].

The syntax of MSCs is defined by the ITU-T in Recommendation Z.120 [IT96]. An MSC essentially consists of a set of processes that run in parallel and exchange messages in a one-to-one, asynchronous fashion. 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., [LL95,ALHH93,IIK+91])

* 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.

to incorporating constraints pertinent to implementation, e.g., architecture and queuing protocols [AHP96]. 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 [MR94,IT95], or from a global state automaton that is obtained via a translation into an algebraic structure called Message Flow Graph [LL95].

To accommodate industrial-size applications, the standard Z.120 [IT95] evolved to allow the description of a large system by composing *basic* MSCs [IT96]. 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 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.

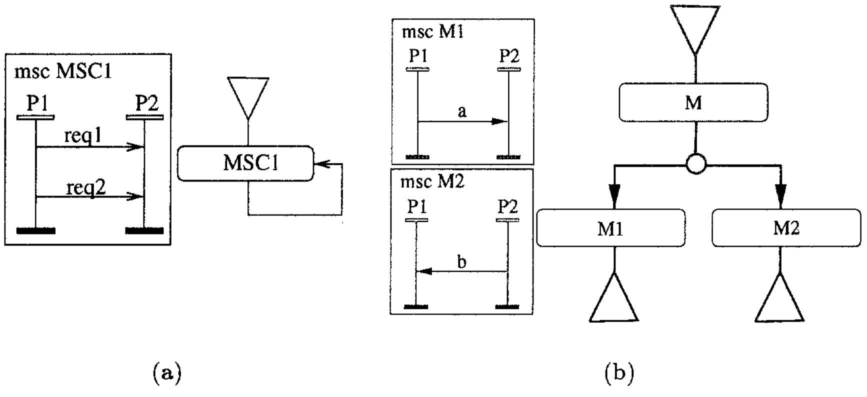


Fig. 1. MSC specification with: (a) process divergence; (b) non-local branching choice

Consider the MSC specification of Figure 1 (a). 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 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. We call such a behavior *process divergence*. As a consequence P1 may overflood P2 with messages req1 and req2. 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., the 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 1 (b) 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.

Currently, basic MSCs can be analyzed for deadlocks [LS], race conditions and timing inconsistencies [AHP96]. These analysis techniques deal only with basic MSCs. In this paper, we syntactically characterize process divergence and non-local branching choice. Our analysis complements the analysis presented in [AHP96] as it is another step towards ensuring that an MSC specification is implementable in a modular fashion and without discrepancies.

2 Message Sequence Chart Specifications

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 [LL95,Leu94].

2.1 Basic Message Sequence Charts

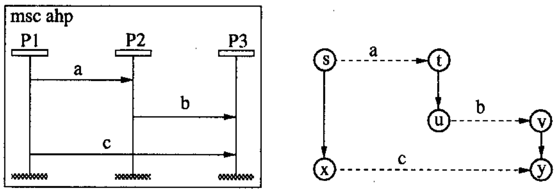


Fig. 2. Basic MSC (left) and corresponding basic MFG (right)

A *basic MSC* (bMSC) describes finite executions of concurrent processes in a system; see Figure 2 for an example. Each vertical line is delimited by a *start* and *end* symbol and represents one process in the system. (Recommendation Z.120 [IT96] 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. The intersections of the vertical lines and arrow tails and heads represent send and receive events, respectively. 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.

The behavior of a bMSC is the set of sequences (or traces) of send and receive events. 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 [LL95]: 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 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. For details, refer to [LL95].

Let M be a bMSC; its corresponding message flow graph is a directed graph $F_M = (S, C, ne, sig, ST, stype, ET, etype)$. Each node represents a communication event in M, i.e., arrow tail or head in M. 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: 1) *next event* edges, set *ne*, which reflect the control flow between communication events within each process of M; and 2) *signal* edges, set *sig*, which reflect the causal precedence of a message's send and receive events. Each signal edge in the set *sig* is labeled with the corresponding message type from ST and which is retrieved with the function *stype*. see Figure 2 for an example of MFG of a bMSC.

We call an MFG that corresponds to a basic MSC a *basic MFG* (bMFG)[1]. Note that in a bMFG, there is a one-to-one mapping between the messages (i.e., arrows) in the bMSC and the set *sig* of signal edges. In addition, the set *ne* of next event edges is a non-branching and cycle-free relation. Further, each maximal-connected components through the *ne* relation corresponds to a process; hence we can also talk about a *process* in the bMFG and we use *ptype*(n) to denote the process to which belongs a node n in the bMFG.

[1] A basic MFG is called *simple* MFG in [LL95].

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. Another advantage is that they provide an algebraic structure from which we can derive the behavior of a bMSC, i.e., all possible states and communication events that the bMSC can execute.

The behavior of a bMSC is described by the 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. 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. The result of a transition on sending an event augments the target state with a signal edge that indicates 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 [IT95], 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 [LL95] 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 $q \xrightarrow{\alpha} q'$.

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.

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 [IT95].

Definition 1. An *MSC specification* is a structure $S = (B, V, suc, ref)$ where

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

The behavior of an MSC specification is obtained in a fashion similar to the behavior of a basic MSC. The bMSC to bMFG translation is extended to account for the hMSC edges which connent referenced bMSCs. When the hMSC contains a loop or a branching the resulting MFG also contains loops and branchings and therefore it is not a basic MFG. Figure 3 illustrates an example of an MFG for an MSC specification. The GSTG of the resulting MFG is derived while accounting for possible branching via a history variable to register the branching decisions made by any process that is ahead of others. For details, refer to [LL95,LLar,BAL96].

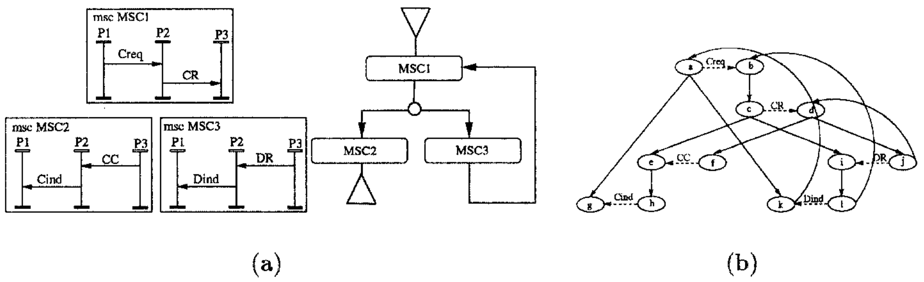


Fig. 3. (a) MSC Specification; (b) its corresponding MFG

3 Deadlock Detection in MSC Specifications

In this paper, the necessity of our syntactic characterization of process divergence and non-local branching choice in MSC specifications will make use of the reachability of a *problematic* state. Reachability can be hampered by *deadlocks*. Since semantic deadlock detection is expensive, one is therefore interested in specifying syntactic ways to detect deadlocks.

The standard syntax of basic MSCs [IT96] indirectly guarantees that a bMSC is deadlock-free via two conditions: an informal constraint on the causality of messages and a drawing rule for message arrows. In Z.120,

> "*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.*" [IT95, Section 4.3]

The Z.120 "connectivity graph" of a bMSC is isomorphic to our bMFG. In addition, both graphs are isomorphic to the *mfg* graphs defined by Ladkin and Simons [LS] 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 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:

> "*Message lines may be horizontal or with downward slope (with respect to the direction of the arrow), ...*" [IT95, Section 2.4]

One can use a topological argument to prove the following conjecture [BAL96].

Conjecture 2. A bMSC that has only horizontal or downwards sloping message arrows is deadlock-free.

The above syntactic characterizations of deadlocks in bMFGs can be easily adapted for MSC specifications. In particular, it is straightforward to prove that for a given MSC specification S, if each of its bMFGs has an acyclic $sig \cup ne$ 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. 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

When concurrent 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 we argued in the introduction, one possible execution of the MSC specification of Figure 1 (a) 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. Further, 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.

It is worth noting that there are MSC specifications for which the above questions are irrelevant. For instance, consider the MSC specification in Figure 4 which slightly differs from the specification in Figure 1 (a): The two processes P1 and P2 in Figure 4 have a hand-shake communication. In this specification, 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 number of messages before the other process can receive any. 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 syntactic examination of the communication between its processes.

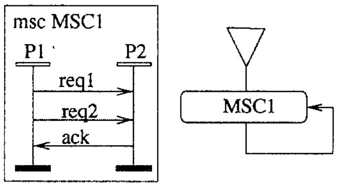


Fig. 4. An MSC Specification with no process divergence

4.1 Semantic Characterization of Process Divergence

In the sequel, for a letter e, a string s and integer m, we use $\#_s(e,m)$ to denote the number of occurrences of e in the prefix of s of length m.

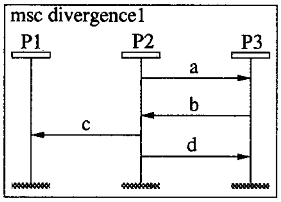
Definition 3. Let $\mathcal{F} = (S, C, ne, sig, ST, stype, ET, etype)$ be an MFG and let $\mathcal{G}_F = (Q, q_0, T_F)$ be its GSTG. We say $\mathcal{F}$ is *divergent* if there exist $\langle x, y\rangle \in sig$ and an infinite sequence of transitions in T_F $q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} q_2 \xrightarrow{\alpha_2} \cdots$ such that for $s = \alpha_0\alpha_1\alpha_2\cdots$, we have

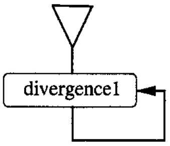
$$\forall n \in \boldsymbol{N} \exists m \in \boldsymbol{N} \;\; \#_s(x,m) > n + \#_s(y,m).$$

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

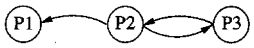
4.2 Syntactic Characterization of Process Divergence

To illustrate the intuition behind our syntactic characterization, let us first examine samples of MSC specifications. The MSC specification in Figure 5 (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 5 (b) contains a





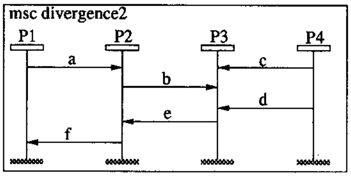
(a)

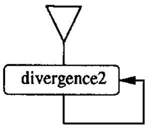


(b)

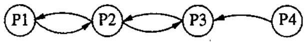
Fig. 5. (a) MSC example `divergence1`; (b) its Coordination graph `coordination1`

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.





(a)



(b)

Fig. 6. (a) MSC example `divergence2`; (b) its Coordination graph

In Figure 6 (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 6 (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).

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. Further, the number of messages exchanged is irrelevant; one message can be enough to cause process divergence.

Definition 4. 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 F;
- $cor_F \subseteq PT_F \times PT_F$ is the set of directed edges such that an edge is from P to Q if P sends a message to Q; formally:
  $cor_F \triangleq \{(P,Q) \in PT_F \times PT_F \mid \exists (a,b) \in sig_F\ (ptype(a) = P \wedge ptype(b) = Q)\}$

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 5. *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.*

Proof. See [BAL96].

Algorithm. The algorithm gets an MSC specification S and returns `DIV_FREE` iff S is not divergent, and returns `DIVERGE` iff S is divergent. 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 [AHU74] which runs in $O(max(|suc|, |V|))$. In the worst case, S has $2^{|I|} - 1$ 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 [AHP96] 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 $O(2^{|I|}(|B|k^2 + k \sum_{M \in B} |sig_M|))$. 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 7 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.

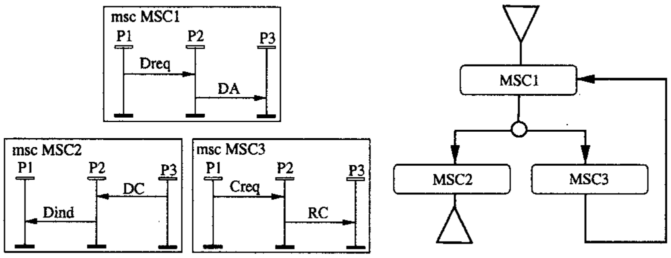


Fig. 7. MSC Specification with a non-local branching choice

For example, consider the specification in Figure 7. 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 [IT96] nor hMSC graphs provide an explicit way to handle such an information exchange. To handle this type of inter-process synchronization, Ladkin and Leue [LL95] 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.

Note that 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 3. 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 [LLar]. When the wait-and-see strategy can be used to resolve a non-determinism within each process, we call the branching a *local branching choice*. Otherwise, when explicit synchronization between the processes is necessary to resolve a non-determinism, we call the branching a *non-local branching choice*.

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 Section 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 [LL95].

Definition 6. Let $S = (B, V, suc, ref)$ be an MSC specification with MFG $F = (S, C, ne, sig, ST, stype, ET, etype)$ 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

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

Informally, the above condition 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 [IT96]. On one hand, this assumption facilitates the interpretation of bMSC sequencing as a "weak sequencing" [IT96], and on the other hand it can be easily supported through a syntactic, pre-processing phase to our analysis; see [BAL96] for one normalization method. The second assumption simplifies the computation of the first event in a sequence of bMFGs (i.e., bMSCs). However, it can be eliminated by modifying the way we compute the first event in a sequence of bMFGs, i.e., bMSCs. This assumption reduces the syntactic verification to checking immediate successors of the branching node b as opposed to successors through the transitive closure of the relation *succ*, as required in Definition 6 [BAL96]. 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*.

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 7. *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 $\Longleftrightarrow$

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

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.
Proof. See [BAL96].

Algorithm. The algorithm gets an MSC specification S and returns the flag NON_LOCAL iff one of the branches in S has a non-local choice; it returns the flag LOCAL iff all branches in S can be resolved locally.

```
Begin
For each intermediate node c
   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
```

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.

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.

6 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.

References

[AHP96] 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.

[AHU74] A. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, chapter 5. Addison-Wesly Publishing Company, 1974.

[ALHH93] 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.

[BAL96] H. Ben-Abdallah and S. Leue. Syntactic analysis of Message Sequence Chart specifications. Tech Report 96-12, Department of Electrical and Computer Engineering, University of Waterloo, November 1996.

[BR95] G. Booch and J. Rumbaugh. *The Unified Method: User Guide Version 0.8.* RATIONAL Software Corporation, October 1995.

[IIK$^+$91] 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.

[IT95] ITU-T. Recommendation Z.120, Annex B: Algebraic Semantics of Message Sequence Charts. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, 1995.

[IT96] ITU-T. Recommendation Z.120. ITU - Telecommunication Standardization Sector, Geneva, Switzerland, May 1996. Review Draft Version.

[Jea92] I. Jacobson and et al. *Object-Oriented Software Engineering - A Use-case Driven Approach.* Addison-Wesley, 1992.

[Leu94] S. Leue. *Methods and Semantics for Telecommunications Systems Engineering.* Doctoral dissertation, University of Berne, Switzerland, December 1994.

[LL95] P. B. Ladkin and S. Leue. Interpreting Message Flow Graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[LLar] S. Leue and P. B. Ladkin. Implementing and verifying scenario-based specifications using Promela/XSpin. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *Proceedings of the 2nd Workshop on the SPIN Verification System, Rutgers University, August 5, 1996.* American Mathematical Society, DIMACS/39, 1997, to appear.

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

[MR94] S. Mauw and M.A. Reniers. An algbraic semantics of basic message sequence charts. *The Computer Journal*, 37(4), 1994.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modelling.* John Wiley & Sons, Inc., 1994.

A Notation and Definitions

Relations. Let $f, g \subseteq R \times R$ denote binary relations over a set R, and S be a set.

$$f \rhd S \triangleq \{(a,b) | (a,b) \in f \ \wedge \ b \in S\} \qquad S \lhd f \triangleq \{(a,b) | (a,b) \in f \ \wedge \ a \in S\}$$
$$domain(f) \triangleq \{a \mid (\exists b \in R)((a,b) \in f)\} \qquad range(f) \triangleq \{b \mid (\exists a \in R)((a,b) \in f)\}$$
$$f \circ g \triangleq \{(a,c) \mid (\exists b)((a,b) \in f \wedge (b,c) \in g)\} \qquad f^1 \triangleq f$$
$$f^+ \triangleq \bigcup_{n>0} f^n \text{ the transitive closre of } f$$

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.