Deciding Properties of Message Sequence Charts

Anca Muscholl
LIAFA, Université Paris 7
2 place Jussieu
75251 Paris Cedex 05
France

Doron Peled
Department of Computer Science
The University of Warwick
Coventry, CV4 7AL
United Kingdom

October 1, 2004

Abstract

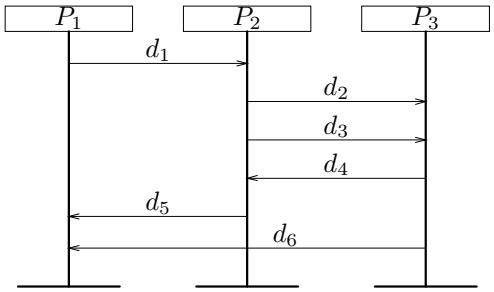
Message Sequence Charts (MSCs) is a notation used in practice by protocol designers and system engineers. It is defined within an international standard (ITU Z120), and is also included, in a slightly different form, in the popular UML standard (called there *sequence diagrams*). We present some of the main results related to this notation, in the context of specification and automatic verification of communication protocols. We look at issues related to specification and verification. In particular, we look at automatic verification (model checking) of MSCs. We study the expressiveness of MSCs, in particular the ability to express communication protocols, and appropriate formalisms for specifying properties of MSC systems.

1 Introduction

Specifying the behavior of software systems is of major importance for engineers. When concurrency is involved, the specification becomes even more challenging. Even before considering the actual notation to be used for specification, there is a large choice of models of execution. Different models vary in the detailed information they carry, the intuition they provide and the difficulty of checking properties of the modeled systems.

Perhaps the most successful model for describing concurrent systems is the interleaving model. An interleaved execution is simply an alternating sequence of actions and states, where each action is *enabled* in the preceding state, and after *executing* it, results in a new state. In this model, all the events are linearly ordered, and concurrently executed events are modeled by ordering them in an arbitrary way. Simple formalisms, such as linear temporal logic, are available for describing properties of interleaving sequences. In the finite case, there are simple decision procedures for checking properties of such models. The *partial order* model allows events to be unordered, if they can independently (concurrently) occur. After the selection of the model, we are still left with a wide choice of notation, affecting our level of abstraction and the complexity of deciding their properties.

Message sequence charts (MSCs) is a partial-order based standard formalism [15]. It has a visual notation, which clearly demonstrates the interaction between the involved concurrent processes. It is already used in practice by protocol designers, a fact which gives it an advantage over other formalisms in technology transfer. On the




```
msc MSC;
inst P1: process Root,
P2: process Root,
P3: process Root;
instance P1;
out d1 to P2;
in d5 from P2;
in d6 from P3;
endinstance;
instance P2;
in d1 from P1;
out d2 to P3;
out d3 to P3;
in d4 from P3;
out d5 to P1;
endinstance;
instance P3;
in d2 from P2;
in d3 from P2;
out d4 to P2;
out d6 to P1;
endinstance;
endmsc;
```

Figure 1: Visual and textual representation of an MSC

other hand, working with an existing standard, which was developed by a committee, initially without a full view of algorithms and complexity issues, can be challenging. In this survey we discuss several issues in specification and verification using message sequence charts.

2 Preliminaries

Each MSC describes a scenario where some processes communicate with one another. Such a scenario includes a description of the messages sent, messages received, the local events, and the ordering between them. In the visual description of MSCs, each process is represented as a vertical line with process name in a box at the top. We usually end a process line with a horizontal line at the bottom. A message is represented by a horizontal or slanted arrow from the sending process to the receiving one, as appears in the left part of Figure 1. The corresponding textual representation of that MSC appears in the right part of Figure 1.

Definition 2.1 *[15] An MSC M is given as a tuple $\langle V, <, \mathcal{P}, \mathcal{N}, L, K, N, m\rangle$, where*

- *V is a finite and nonempty set of events,*
- *$< \subseteq V \times V$ is an acyclic relation (with further details below),*
- *$\mathcal{P}$ is a set of processes,*
- *$\mathcal{N}$ is a set of* message names,

- $L : V \to \mathcal{P}$ *is a mapping that associates each event with a process,*
- $K : V \to \{\mathsf{s}, \mathsf{r}, \mathsf{l}\}$ *is a mapping that describes the* kind of *each event as send, receive or local, respectively,*
- $N : V \to \mathcal{N}$ *maps every event to a name,*
- $m \subseteq V \times V$ *is a nonempty relation called* matching *that pairs up send and receive events. Each send is paired up with exactly one receive. Events* v_1 *and* v_2 *can be paired up with each other, only if* $N(v_1) = N(v_2)$, $K(v_1) = \mathsf{s}$ *and* $K(v_2) = \mathsf{r}$.

A type *is a triple* (i, j, C)*, including the indexes of the sending process* $P_i \in \mathcal{P}$ *and receiving process* $P_j \in \mathcal{P}$*, and a message name* $C \in \mathcal{N}$*. Each send or receive event has a type, according to the origin and destination of the message, and the label of the message. The type of a local event of process* $P_i \in \mathcal{P}$ *is* (i, i)*. Matching events have the same type. A message consists of a pair of matched send and receive events. For two events* v_1 *and* v_2*, we have* $v_1 < v_2$ *if and only if one of the following holds:*

- v_1 *and* v_2 *are matching send and receive events, respectively.*
- v_1 *and* v_2 *belong to the same process, with* v_1 *appearing before* v_2 *on the process line.*

We assume FIFO (first in first out) message passing, i.e., message arrows on the same channel do not cross each other:

$$(m(v_1, v_1') \wedge m(v_2, v_2') \wedge v_1 < v_2 \wedge \\ L(v_1) = L(v_2) \wedge L(v_1') = L(v_2')) \Rightarrow v_1' < v_2'$$

Denote by $u \longrightarrow v$ the fact that $u < v$ and either u and v are matching *send* and *receive* events, or u and v belong to the same process and there is no event between u and v on the same process line. We say in this case that u *immediately precedes* v.

Definition 2.2 *The transitive closure* $<^*$ *of the relation* $<$ *is a partial order called the* visual ordering *of events. Clearly, the visual ordering can be defined equivalently as the transitive closure of the relation* $\longrightarrow$*. A chain of events* $e_1 <^* e_2 \ldots <^* e_n$ *is called a* causal chain*.*

The MSC notation represents a partial order execution, where the fact that two events u, v are ordered according to the visual order means that u happens before v. A *linearization* of an MSC $M = \langle V, <, \mathcal{P}, \mathcal{N}, L, K, N, m\rangle$ is a total order on V, which extends the relation $(V, <)$.

Example 2.3 Consider the example MSC given in Figure 1. For each message d_i, $1 \leq i \leq 6$, denote by s_i the *send* event and by r_i the receive event. Then we have $V = \{s_1, \ldots, s_6, r_1, \ldots, r_6\}$, $\mathcal{P} = \{P_1, P_2, P_3\}$, $\mathcal{N} = \{d_1, \ldots, d_6\}$ and $N(s_i) = N(r_i) = d_i$ for each i. The events located on P_1 are $L^{-1}(P_1) = \{s_1, r_5, r_6\}$, with $K(s_1) = \mathsf{s}$, $K(r_5) = K(r_6) = \mathsf{r}$, and $s_1 < r_5 < r_6$. This ordering is the time ordering of events on P_1. We also have $m(s_i, r_i)$ and $s_i < r_i$ for each i (message ordering). In particular, $s_1 < r_1 < s_2 < r_2$.

The partial order between the *send* and *receive* events of Figure 1 is shown in Figure 2. In this figure, only the 'immediately precedes' order $\longrightarrow$ is shown. Notice for example that the *send* events v_5 and v_6, of the two messages, d_5 and d_6, respectively, are unordered.

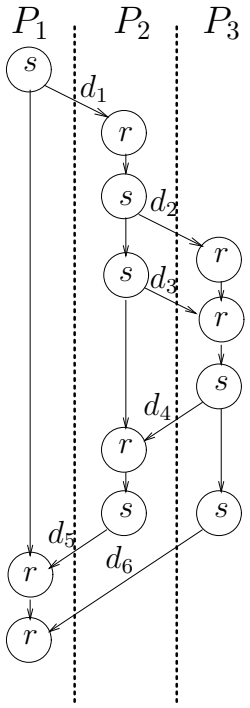


Figure 2: The partial order between the events of the MSC in Figure 1.

Definition 2.4 *The* concatenation $M_1 M_2$ *of two MSCs,*

$$M_k = \langle V_k, <_k, \mathcal{P}, \mathcal{N}_k, L_k, K_k, N_k, m_k \rangle, \text{ for } k = 1, 2.$$

over a common set of processes $\mathcal{P}$ and disjoint sets of events $V_1 \cap V_2 = \emptyset$ (we can always rename events so that the sets become disjoint) is defined as $\langle V_1 \cup V_2, <, \mathcal{P}, \mathcal{N}_1 \cup \mathcal{N}_2, L_1 \cup L_2, K_1 \cup K_2, N_1 \cup N_2, m_1 \cup m_2 \rangle$, where

$$< = <_1 \cup <_2 \cup \{(u, v) \in V_1 \times V_2 \mid L_1(u) = L_2(v)\}.$$

That is, the events of M_1 precede the events of M_2 *for each process*, respectively. If $M = M_1 M_2$, we say that M_1 is a *prefix* of M, and denote this by $M_1 \sqsubseteq M$. This also means containment between the different process events of the MSCs M_1 and M. Notice that no synchronization of the different processes is assumed in the definition of concatenation. Thus, $M_1 M_2$ does not necessarily describe a behavior that starts according to M_1 and after completing all the events from M_1 progresses to behave according to the events in M_2. In particular, it is possible in $M_1 M_2$ that one process is still involved in some actions of process P_i, while another process has advanced to events from another process P_j. Such a situation is demonstrated later in this section. The infinite concatenation of finite MSCs is defined in a similar way, and it allows defining infinite MSCs as well.

Definition 2.5 *Let $M_1, M_2, \ldots$ be an infinite sequence of finite MSCs. Define a sequence $M_1{}', M_2{}', \ldots$ as follows: Let $M_1{}' = M_1$, and for $i > 1$, $M_i{}' = M'_{i-1} M_i$. (Thus, for $i < j$, $M_i{}' \sqsubseteq M_j{}'$.)*

Let $M'_i = \langle V_i, <_i, \mathcal{P}, \mathcal{N}_i, L_i, K_i, N_i, m_i \rangle$. Then, $V_i \subseteq V_{i+1}$, $<_i \subseteq <_{i+1}$, $\mathcal{N}_i \subseteq \mathcal{N}_{i+1}$, $L_i \subseteq L_{i+1}$, $K_i \subseteq K_{i+1}$, $N_i \subseteq N_{i+1}$ and $m_i \subseteq m_{i+1}$. The infinite concatenation $M_1 M_2 \ldots$ is defined as the infinite MSC $M = \langle V, <, \mathcal{P}, \mathcal{N}, L, K, N, m \rangle$ where $V = \cup_{i \geq 1} V_i$, $\mathcal{N} = \cup_{i \geq 1} \mathcal{N}_i$, $L = \cup_{i \geq 1} L_i$, $N = \cup_{i \geq 1} N_i$, $K = \cup_{i \geq 1} K_i$, $N = \cup_{i \geq 1} N_i$, $m = \cup_{i \geq 1} m_i$ and $<= \cup_{i \geq 1} <_i$. Each component defining M is thus the limit of the partial unions for the same component in the finite prefixes M'_i.

Since a communication system usually involves multiple (or even infinitely many) MSC scenarios, a high-level description is needed for combining them. The standard description consists of a graph called HMSC (high-level MSC), where each node contains one finite MSC as in Figure 3. Each maximal path in this graph (i.e., a path that is either infinite or ends with a node without outgoing edges) that starts from a designated initial state corresponds to a single *execution* or *scenario*.

Definition 2.6 *[15] An HMSC is a 4-tuple $\langle \mathcal{S}, \mathcal{M}, c, \tau, \mathcal{S}_0 \rangle$ where $\mathcal{S}$ is a finite set of nodes, $\mathcal{M}$ is a set of finite MSCs with sets of events disjoint from one another. The mapping $c : \mathcal{S} \rightarrow \mathcal{M}$ associates a node $g \in \mathcal{S}$ with an MSC $c(g)$. By $\tau \subseteq \mathcal{S} \times \mathcal{S}$ we denote the* edge relation*. The* initial *nodes $\mathcal{S}_0$ are a subset of $\mathcal{S}$. An* execution *of the HMSC is a (finite or infinite) MSC $c(g_0)\, c(g_1)\, c(g_2) \cdots$ associated with a maximal*[1] *path $g_0, g_1, \ldots$ of the HMSC that starts with some initial node $g_0 \in \mathcal{S}_0$.*

The set of executions of an HMSC is also referred to as the set of MSC *generated* by that HMSC. Figure 3 shows an example of an HMSC. The node in the upper left corner, denoted M_1, is the starting node, hence it has an incoming edge that is connected to no other node. Initially, process P_1 sends a message to P_2, requesting a connection (e.g., to an internet service), according to node M_1. This can result in either an approval message from P_2, according to the node M_2, or a failure message, according to node M_3. In the latter case, a report message is also sent from P_2 to some supervisory process P_3. There are two progress choices, corresponding to the two arrows out of node M_3. We can decide to try and connect again, by choosing the arrow from M_3 to M_1, or to give up and send a service request (from process P_1 to process P_3), by choosing to progress according to the arrow from M_3 to M_4. Note how the HMSC description abstracts away from internal process computation, and presents only the communications. The executions of this system are either finite or infinite. Consider the path $M_1\, M_3\, M_4$. According to the HMSC semantics, process P_2 in Figure 3 does not necessarily have to send its Report message in M_3 before the execution of process P_1 has progressed according to M_4 sent its Req_service message. However, process P_3 must receive the Report message before the Req_service message.

According to the ITU standard [15], an HMSC can be hierarchical, i.e., an HMSC node can be mapped into another (lower level) HMSC. We ignore this feature, which is orthogonal to the discussion in this survey and refer to [9] for algorithms on hierarchical HMSCs.

3 Expressiveness

Message sequence charts (MSCs) (including the extension to High level MSCs, i.e., HMSCs) is a formalism that is used in practice by protocol developers and software

[1] By maximality we mean that a path is either infinite, or terminates with a node that has no successor according to the relation τ.

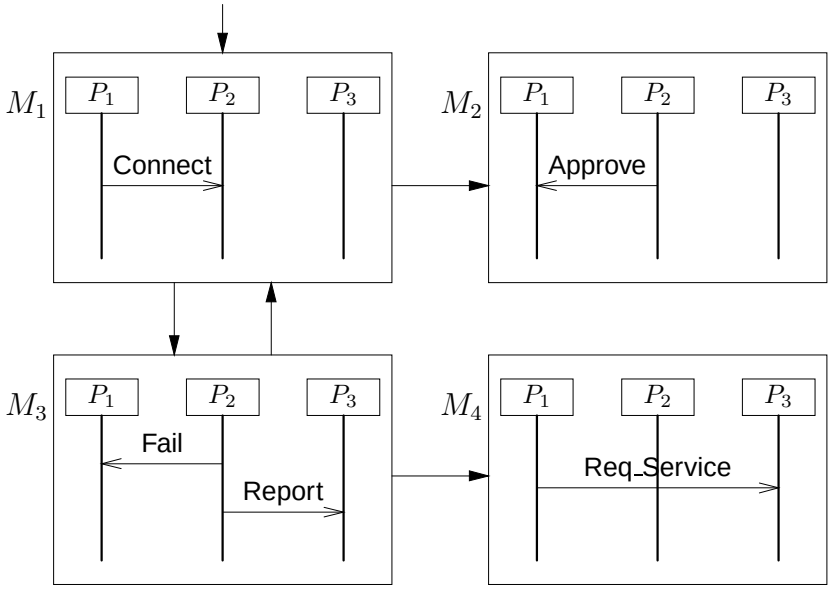


Figure 3: An HMSC graph

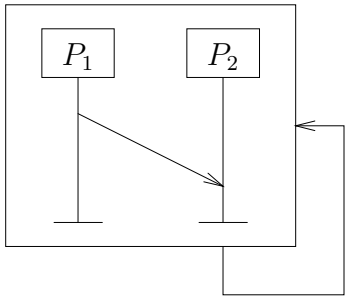


Figure 4: Simple example with infinite state space

engineers. Unlike some other specification formalisms, it was not designed by researchers to fit into existing theory or tools. This calls for the study of its properties, in an attempt to adapt some formal methods techniques, or develop new ones.

There are several interesting aspects of the MSC notation that pose a challenge to the researchers and the developers of tools. For example, the HMSC notation does not necessarily represents *finite* state systems, as there is no bound on the size of message channels and due to concurrent processes. This fact has implications on the ability to automatically verify properties of HMSCs. Consider for example the HMSC in Figure 4. This is the simplest example of an HMSC with infinitely many global states. In order to formalize this observation, we define the notion of a *global state* of an MSC.

Definition 3.1 *Let* $M = \langle V, <, \mathcal{P}, \mathcal{N}, L, K, N, m \rangle$ *be a finite or infinite MSC (the latter case is obtained, e.g., by an infinite execution of an HMSC). A* global state G *is a finite subset of the events of* V*, such that if* $f \in G$ *and* $e <^* f$*, then* $e \in G$*. (We say*

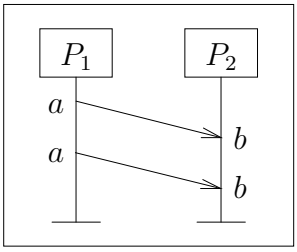


Figure 5: An MSC with two messages

that G is 'history closed'.)

Now, it is easy to see that the states of the unique and infinite execution of the HMSC in Figure 4 consists of k *send*s and l *receive*s for *any* natural numbers $k \geq l$. A global state of M is usually defined, in the context of software verification, as an assignment function from the program variables to their values. In the MSC context, the assignment can return the sequence of pending messages on each channel, together with the last event on each process.

It is interesting to know what is the expressive power of HMSCs. In order to remain within the domain of formal languages, we will look at the *linearizations* of MSC executions, i.e., their completions into total orders. We will label each event in an MSC node with a label from a finite alphabet Σ. We allow (but do not force) labeling of different events of the *same type and kind* by the same letter.

Consider the MSC in Figure 5. It has two messages, i.e., 4 events. We labeled the *send*s with a, and the *receive*s with b. This MSC generates two linearizations (words): $abab$ and $aabb$. These languages of linearizations are closed under certain permutation of adjacent occurrences of events. We have three *permutation rules*:

1. If b is a *receive* of a message from P_i to P_j, and a is a *send* from P_i to P_j, then we can permute $\sigma_1 ba\sigma_2$ ($\sigma_1, \sigma_2 \in \Sigma^*$) to obtain $\sigma_2 ab\sigma_2$. Note this rule does not necessarily permit us to permute in the reverse direction, i.e., from $\sigma_1 ab\sigma_2$ to $\sigma_1 ba\sigma_2$.

2. If a is a *send* from P_i to P_j, and b is a *receive* from P_i to P_j, we can permute a with b in $\sigma_1 ab\sigma_2$ provided that the following condition hold: $\#_a\sigma_1 > \#_b\sigma_1$, where $\#_c\sigma$ denotes the number of c's appearing in the word σ.

3. If a and b belong to different processes, and their types do not match as in the previous case, then we can permute a with b. (In fact, we can also permute b with a, from the symmetry of this condition.)

The reason that reverse permutation of the first rule is not necessarily allowed is that it may cause a *receive* to appear before the corresponding *send*. For example, given the linearization $abab$ of the MSC in Figure 5, we cannot permute the first a with the first b to obtain $baab$. The second rule specifies the condition under which the reverse permutation is allowed. Under this rule, the adjacent a and b, which can be permuted, are *not* a matching pair. Also note that for MSCs, it is not possible to use a fixed symmetric independence relation between events, as in *trace theory* [23].

We can define the *language of an HMSC* as follows. Let $\mathcal{L}(M)$ be the (finite) language of an HMSC node M. Let $\mathcal{K}$ be the language of the graph of the HMSC, where

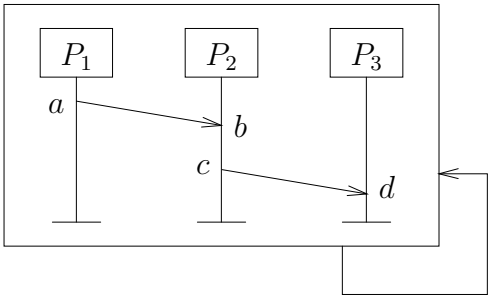


Figure 6: An MSC with context-free behavior

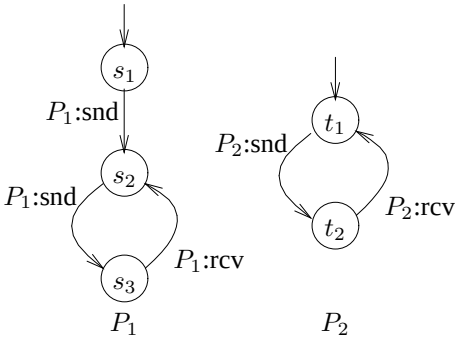


Figure 7: A simple two process protocol

each node in the graph is assigned some unique letter (disjoint from the letters in Σ). According to Kleene's construction, the language $\mathcal{K}$ is a regular language. Substitute in $\mathcal{K}$ each letter corresponding to a node M by the language of $\mathcal{L}(M)$. This is still a regular language, denoted $\tilde{\mathcal{K}}$. Now close $\tilde{\mathcal{K}}$ under the permutation rules to obtain $[\tilde{\mathcal{K}}]$. Such permutations are achieved by using context sensitive grammar rules of the form $XabY \rightarrow XbaY$. Hence the language $[\tilde{\mathcal{K}}]$ of an HMSC is context sensitive. Note that the language of an HMSC H is the set of all linearizations of executions of H. Note also that we can only permute events according to the first and third permutation rules given above. This is sufficient due to the fact that we took *all* the linearizations of each separate MSC node. This is because a *send* event a from P_i to P_j in a node g and a *receive* event b, also from P_i to P_j of a later node h can never be commuted; the event a necessarily precedes the *send* event that matches with the *receive* b.

Thus, HMSC languages are obtainable from regular languages (ω-regular in the case of infinite executions) by closing under a given set of permutations. To show that the language of HMSCs is, in general, not regular or context free, consider the example in Figure 6. The global states of this example have l times a events, m times c events, and n times d events, where $l \geq m \geq n$ (also the number of b events is the same or greater than, by exactly one, than the number of c events). This can be easily shown not to be in the class of context-free (and hence also not regular) languages.

On the other hand, we show that the HMSC notation does not allow representing all the possible communication skeletons of finite state communication protocols [11].

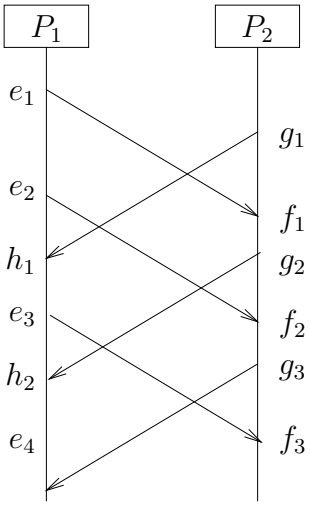


Figure 8: A prefix of an MSC execution that cannot be decomposed .

This makes HMSCs incomparable with regular languages.

As an example, consider the infinite MSC that is generated from the simple protocol in Figure 7. A finite prefix of the MSC description of the (unique and infinite) execution of this protocol appears in Figure 8. We show that this infinite MSC cannot be decomposed into a concatenation of finite MSCs. We start with the *send* event e_1 and *receive* event f_1. Obviously, because of the compulsory matching between corresponding *send* and *receive* events in HMSCs, they must belong to the same MSC node. We have the *send* event g_1 preceding f_1, on the same process line, while its corresponding *receive* event h_1 succeeds the *send* e_1. Thus, the events g_1 and h_1 cannot be in an MSC preceding the one containing the events e_1 and f_1, nor it can be in an MSC succeeding it. Consequently, these four events must be in the same HMSC node. For the same reason, we have that e_2 and f_2 must belong to the same node with g_1, and h_1, and so forth.

The problem lies within the restriction of the MSC nodes to contain matched messages. A different view of the expressiveness problem is that any global state that corresponds to a finite path in an HMSC (i.e., a global state that contains complete MSC nodes) has a matched set of *send* and *receive* events. In the partial order execution in Figure 8, there is no global state with this property. Hence, we cannot decompose this execution into finite MSCs (which will occur infinitely many times along some path of an HMSC).

3.1 Compositional MSC

An extension of the HMSC notation is described in [11]. It allows MSC nodes with unmatched *send* and *receive* events. Thus, a *send* event in one node may be matched with a *receive* event in a later node.

In order to represent communication protocols, whose description could only be approximated using standard MSCs, we suggest an extension of the MSC standard. Intuitively, a *compositional MSC*, or CMSC, may include *send* events that are not matched

by corresponding *receive* events and vice versa. An unmatched *send* event in one node in a path may be matched in future HCMSC nodes on that path. Similarly, an unmatched *receive* event may be matched in previous HCMSC nodes. The definition of a CMSC is hence similar to an MSC, except that unmatched *send* and *receive* messages are allowed.

Definition 3.2 *[11] A CMSC M is defined as in Definition 2.1, except for the following modification:*

- $m \subseteq V \times V$ *is a* partial *function called* matching *that pairs up send and receive events. Each send event is paired up with* at most *one receive event and vice versa. Events that are paired up are called* matched*, otherwise, they are* unmatched*. Matching events must have the same type.*

Unmatched *send* events are supposed to be matched by *receive* events belonging to subsequent nodes, whereas unmatched *receive* events are supposed to be matched by *send* events belonging to preceding nodes. The above definition allows unmatched *receive* events that do not correspond to any unmatched *send* event. (Allowing unmatched *send* events that do not correspond to a later *receive* is a lesser problem, as this can actually happen in communication protocols.)

We denote an unmatched *send* by a message arrow, where the *receive* end (the target of the arrow) appears within an empty circle. Similarly, an unmatched *receive* is denoted by an arrow where the *send* part (the source of the arrow) appears within a circle. CMSC arrows where both the *send* and the *receive* events are unmatched events are forbidden. In Figure 9, we can see an HCMSC that represents the execution that is approximated in Figure 8.

Definition 3.3 *A CMSC is called* left-closed*, if it does not contain unmatched receive events, or any unmatched send event that precedes another matched send of the same type (the latter condition excludes send events that could never be matched without violating the FIFO order).*

Definition 3.4 *Consider two CMSCs $M_1 = \langle V_1, <_1, \mathcal{P}, \mathcal{N}_1, L_1, K_1, N_1, m_1\rangle$ and $M_2 = \langle V_2, <_2, \mathcal{P}, \mathcal{N}_2, L_2, K_2, N_2, m_2\rangle$ over disjoint events sets. Define the matching function m' that pairs up unmatched send events of M_1 with unmatched receive events of M_2 according to their order on their process lines. That is, the ith unmatched send in M_1 is paired up with the ith unmatched receive event of the same type in M_2.*

The concatenation $M_1 M_2$ *is then defined as $\langle V_1 \cup V_2, <, \mathcal{P}, \mathcal{N}_1 \cup \mathcal{N}_2, L_1 \cup L_2, K_1 \cup K_2, N_1 \cup N_2, m_1 \cup m_2 \cup m'\rangle$, where*

$$\begin{aligned} < \;=\; & <_1 \cup <_2 \cup \\ & \{(v_1, v_2) \in V_1 \times V_2 \mid L_1(v_1) = L_2(v_2)\} \cup m' \end{aligned}$$

provided that $M_1\, M_2$ is a CMSC satisfying the FIFO property when restricting the events to the matched pairs of events.

Clearly, the concatenation of CMSCs is not associative anymore. Hence, when we write $M_1 \cdots M_k$ we mean the concatenation $(\cdots (M_1 M_2) M_3) \cdots M_k)$. Again, we can define the prefix relation $M_1 \sqsubseteq M$ if there exists M_2 such that $M_1\, M_2 = M$. The definition of an infinite concatenation for CMSCs follows the lines of Definition 2.5. Note that in an infinite concatenation, there can be infinitely many unmatched messages sent from one process to another.

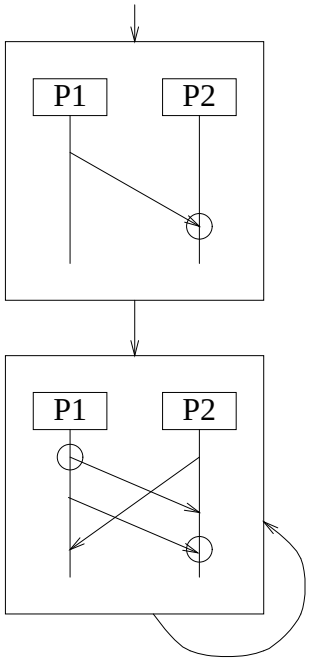


Figure 9: A decomposition of the execution in Figure 8.

An HCMSC is a graph whose nodes are CMSCs. Except for that, the definition of HCMSCs is the same as Definition 2.6. Similarly, an HCMSC *execution* is the CMSC $c(g_0)c(g_1)\ldots$ associated with a path $g_0, g_1, \ldots$ in the HCMSC graph, starting with some initial node g_0, as in Definition 2.6.

3.2 Safe HCMSC

The definition of HCMSCs allows obtaining some "unreasonable" paths in HCMSCs, e.g. in which at some points there are more *receive* events than the corresponding *send* events for some ordered pair of processes. It is not clear how to treat such paths. One way, is to disregard such paths as executions of the HCMSC system. Another approach, which will be taken in this section, is to forbid HCMSCs with such paths.

Remark 3.5 [15] An HCMSC is *safe*[2] if the execution of every finite path starting with the initial state is a left-closed CMSC.

Note that we explicitly allow executions with unmatched *send* events. The HCMSC of Figure 9 is such that every finite execution is a left-closed CMSC with unmatched *send* events. However, the unique maximal execution corresponds to an infinite MSC, where all the events are pairwise matched. Definition 3.3 of left-closedness guarantees that no unmatched *send* cannot may prevent the system to satisfy the FIFO condition by matching it later.

[2]Such HCMSCs are called realizable in [11]. This name usually refers to the realizability/implementability problem, so we prefer to recast it into "safe".

We will show how to test whether an HCMSC is safe. From the definition of safe HCMSCs, we can focus on messages sent from each P_i to another process P_j separately. There are three situations that violate the safety of a HCMSC on a given prefix of a path:

1. There are more unmatched *receive* events than *send*s.
2. Reaching a matched *send-receive* pair, the kth unmatched *send* is before the matched pair, but the kth unmatched *receive* comes after that matched pair. This will generate a non-FIFO behavior.
3. The kth unmatched *send* has a message name C, while the kth unmatched *receive* has a message name D, where $D \neq C$.

To check whether an HCMSC is safe [11], we construct a nondeterministic pushdown automaton $\mathcal{S}_{i,j}$ for each ordered pair of processes $P_i,\ P_j$ that exchange messages in the HCMSC. A pushdown automaton is a quadruple, $\mathcal{S} = \langle Q, \Gamma, \Sigma, \Delta \rangle$, such that

- Q is a finite set of control states,
- Γ is a finite stack alphabet which in our case will be $\Gamma = \{\perp, 1\}$, where $\perp$ is the 'stack bottom' symbol,
- Σ is the input alphabet, which includes *unmatched-send* C, *unmatched-receive* C, or *matched-*C, such that C is a message name from $\mathcal{N}$, and $\Delta \subseteq (Q \times \Sigma \times \Gamma) \times (Q \times \{pop, push, skip\})$ is the set of transition rules. Depending on the current state and symbol at the top position at the stack and the current input symbol, a pushdown automaton has a choice of
  - the next state and
  - whether to *pop* the current top element from the stack, *push* another symbol on top of it, or *skip*, i.e., keep its current contents. The stack contents in our case always belongs to $\perp 1^*$.

The stack is used as a counter, where the counter value is the number of '1' symbols on the stack, and a zero is represented by a stack containing only '$\perp$'. We can partition the transitions according to their effect on the number of '1' symbols in the stack: incrementing, decrementing, or testing whether the contents of the stack is zero.

For every pair of processes $P_i,\ P_j$ we define the pushdown automaton $\mathcal{S}_{i,j}$ by replacing each node in the HCMSC by a linearization (total ordering) of the matched and unmatched *send* and *receive* events. We allow only linearizations in which unmatched *receive* events of some type precede all the unmatched *send* events of the same type. It follows easily from the definitions that such a linearization always exists. The automaton $\mathcal{S}_{i,j}$ will follow such events in a node, and then will continue according to the events of a successor of the current node and so forth (nondeterministically, as there can be more than one HCMSC successor). The pushdown automaton will reach an *accept* state exactly when it discovers that the HCMSC is not safe due to communications from P_i to P_j.

We describe now the automaton $\mathcal{S}_{i,j}$ informally. It contains two phases. In the first phase, it increments each time an unmatched *send* event occurs, and decrements each time an unmatched *receive* occurs. It moves to an *accept* state when either the stack is empty (containing only $\perp$), and an unmatched *receive* occurs, or when a matched *send-receive* event occurs and the stack is not empty. This takes care of the cases 1

and 2 above. To take care of case 3, upon the occurrence of an unmatched *send*, the automaton can nondeterministically 'guess' that the corresponding *receive* has a different name. It saves the message name C in its finite control and ignores all subsequent events, except for unmatched *receive* events, where it decrements one '1' from the stack. Upon reaching an empty stack, it compares the last *receive* name D with the name stored C. If $C \neq D$, it transfers to an *accept* state, and otherwise, it just ignores the rest of the events. Reaching an *accept* state means that the HCMSC is *not* safe.

The motivation behind the definition of compositional MSCs was to capture finite state communication protocols, like the one of Figure 8:

Theorem 3.6 *[11] Every finite state communication protocol can be transformed into an equivalent safe HCMSC (in polynomial size).*

Clearly, the converse of the theorem above does not hold. This happens for the same reasons as for HMSCs, as demonstrated in Figure 6.

4 Undecidable Versions of Model Checking for HMSCs

Once we characterize HMSC languages as context sensitive languages, it is not too surprising that certain decision problems become undecidable. The state based model checking (see e.g. [14, 19, 30, 7]) prescribes using a finite state model for representing the execution sequences of a system and another finite state automaton (over finite or infinite words) for representing the specification. The specification describes the *bad* executions, i.e., the ones we do not want the system to have. We take the intersection of the languages of the system automaton and the specification automaton to find whether there are bad sequences allowed by the system.We can try, along these lines, to specify the bad or unwanted executions of a system using the HMSC formalism. If the intersection of the linearizations of two HMSCs is nonempty, we can easily take one and generate back an MSC.

Alternatively, we can use a specification of the *good* sequences, i.e., the executions we allow. However, in this case we need to perform a test for language inclusion, which is often of higher complexity when using HMSCs. The reason is that contrary to logical specifications, that can be negated without any blow-up, HMSCs cannot be always complemented. As an example, consider the trivial HMSC with one node labeled by the empty MSC over the process set $\mathcal{P}$. This HMSC generates the empty set and its complement (i.e., the set of all MSCs over $\mathcal{P}$) cannot be generated by an HMSC (neither by a safe CHMSC).

The corresponding HMSC model checking problem is to intersect two HMSCs, one corresponding to the system description, and another representing the 'bad' MSC executions. It is known that the emptiness of the intersection of two context sensitive languages is undecidable. We still have to prove that for HMSC languages, as they form a subset of the context sensitive languages:

Theorem 4.1 *[27] The problems of intersection of two HMSCs is undecidable.*

Proof. By reduction from Post Correspondence Problem (PCP). The input for PCP is a finite sequence of pairs of words

$$(w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n)$$

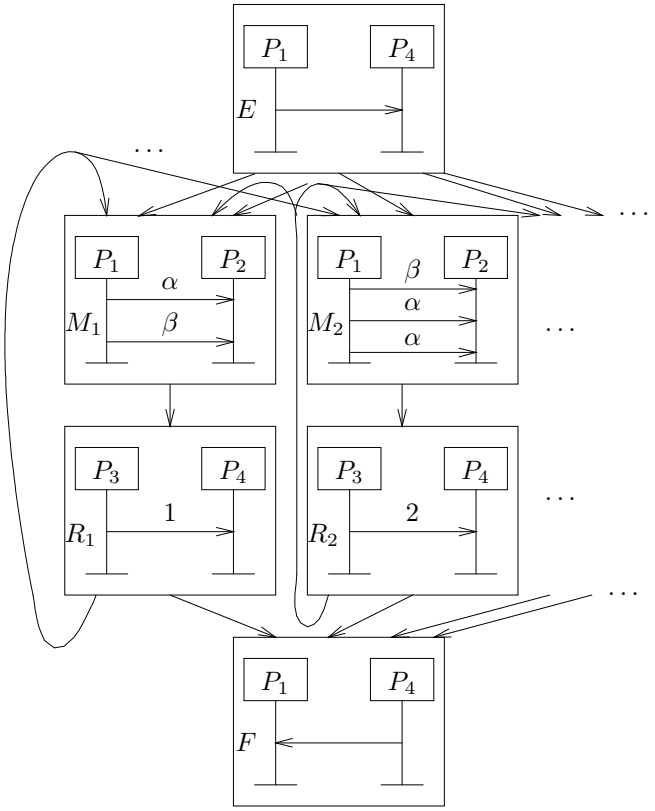


Figure 10: An HMSC graph for the PCP reduction

The problem is to decide whether there is a finite sequence of indexes $i_1, i_2, \ldots, i_m$ such that $w_{i_1} w_{i_2} \ldots w_{i_m} = v_{i_1} v_{i_2} \ldots v_{i_m}$.

We construct two HMSCs. One for concatenating words that appear in the left components of the above pairs, and one for concatenating words that appear in the right components. Consider the HMSC for the left components. We have 4 processes $P_1, \ldots P_4$. For each word w_j, we construct an MSC node M_j with messages from P_1 to P_2 labeled by the letters of w_j. We also have a node R_j, with one message, from P_3 to P_4, labeled by the index j. We also have an initial node E, with a message from P_1 to P_4, and a node F, with a message from P_4 to P_1. The structure of the automaton can be represented by the regular expression $E(\sum_{j=1..n} M_j R_j)^+ F$, which is also demonstrated in Figure 10. That is, we need to start with the initial node E, then repeatedly make a nondeterministic choice of $M_j R_j$ for $1 \leq j \leq n$, and finally end with node F.

The automaton for concatenating the right components is constructed similarly. Now notice that the events in the M_j components can commute with the events in the R_j components, since they involve disjoint processes. Therefore, any word in the intersection has the same characters according to the sequence of M_js, and the same indexes according to the sequence of R_js. ∎

Another attempt for providing model checking is to write the specification (or the

negation of the specification, describing the bad executions) using an automaton over finite or infinite words, or using linear temporal logic (LTL). Unfortunately, the intersection of HMSC languages with regular languages, or the language of words satisfying linear temporal logic formulas, is undecidable as well.

To see this, replace in the previous proof the HMSC for the right components (the 'specification automaton') by an LTL formula (or regular expression, or a finite state automaton over infinite words) that represents some of the linearizations of the HMSC as follows: for an MSC node M, let lin(M) be the single linearization of M that includes matching *send* and *receive* events appearing adjacent. (Note that his kind of linearization is not always possible for an MSC, but is possible in our case because of the particular construction of the nodes in the reduction.) Thus the linearization of M_j, representing the word $w_j = \alpha\beta\beta\alpha$ will be $s_\alpha r_\alpha s_\beta r_\beta s_\beta r_\beta s_\alpha r_\alpha$, where s_ρ represents a *send* of a message labeled by ρ, and r_ρ represents a *receive* of that message. The LTL formula will represent the language $\text{lin}(E)(\bigcup_{j=1..n} \text{lin}(M_j)\text{lin}(R_j))^+\text{lin}(F)$ (this is a counter-free language, and thus can be represented using LTL).

The intersection of the (language of the) HMSC, representing the left words in the PCP problem, and the language of the LTL formula above, representing the right words, would include exactly the words that are solutions to the PCP problem. That is, we have the same concatenation of words, with the same sequence of indexes. Hence, LTL model checking of HMSCs is undecidable.

5 Decidable Versions of Model Checking for HMSCs

There are several positive solutions for providing model checking algorithms for HMSCs. One possibility is to consider restricted classes of HMSCs. The most restrictive approach considers regular HMSCs, that correspond to finite state systems for which the usual model checking approaches can be used. Other solutions are listed below.

5.1 Regular and Cooperative HMSCs

A constraint for HMSCs ensuring regularity is the following [5, 26].

Definition 5.1 *The communication graph $CG^M = \langle P, \rightarrow \rangle$ of an MSC M contains the processes $P \in \mathcal{P}$ of M that occur in M, and with edges $P_i \rightarrow P_j \in E$ if there is a message from P_i to P_j in M.*

Definition 5.2 *[5, 26] An HMSC H is* regular, *if for each loop σ in the graph of H, the communication graph CG^M of the MSC M labeling σ is strongly connected.*

The definition of regular HMSCs is syntactic, and be checked in co-NP [26, 5]. Model checking becomes decidable for regular HMSCs [26, 5], since their languages are regular. More precisely:

Theorem 5.3 *[17, 18] A set of MSCs is generated by a regular HMSC if and only if it has a regular set of linearizations and is generated by a finite set of MSCs.*

An *HMSC state* is a global state associated with an execution M of the HMSC H. We will show that the number of pending messages (i.e., messages that are sent but not yet received) in any HMSC state of a regular HMSC is finite. Note however that a bound on pending messages does not suffice for representing HMSC linearizations

using a finite state automaton. As an example, consider a (non-regular) HMSC over processes P_1, P_2, P_3, P_4 consisting of two nodes g_0, g_1. The initial node g_0 has a self-loop and a transition to the sink node g_1. Node g_0 is labeled by the following MSC with 4 messages: there is a message from P_1 to P_2, and back, and a message from P_3 to P_4, and back. Node g_1 is labeled by the empty MSC. Now, the number of pending messages is at most 2, but the set of linearizations is not regular.

Theorem 5.4 *For any regular HMSC there is a bound on the number of pending messages in any HMSC state.*

Proof. It is sufficient to show that for each pair of processes P and Q, there is a bound on the number of occurrences of a regular HMSC node g that can contribute to the pending messages.

Let $n = |\mathcal{P}|$, i.e., the number of processes. An upper limit on the number of graphs with n nodes, and also on the number of simple paths in such a graph is $k = 2^{n^2}$. Consider a global state G generated for a maximal path (i.e., a finite MSC) σ. Consider the occurrences of unreceived *send* events from process P to process Q on σ in G. Let g be a node of the HMSC that includes such an event. Assume for the contradiction that there are $l = nk + 3$ such occurrences g_0, g_1, ..., g_{nk+2}, of g that contribute to the global state G.

There are nk cycles, g_i to g_{i+1} for $1 \leq i \leq nk + 1$, *after* the first occurrence g_0 of g and *before* the last occurrence g_{nk+2}. Each such cycle σ_i is a subpath of σ. By the choice of l, considering the communication graphs corresponding to the cycles σ_i, at least one such graph repeats n times. Let μ be a simple path in such a communication graph from the node corresponding to process Q to the node corresponding to process P. Hence μ consists of at most $n - 1$ edges.

Distinguish s and r as a *send–receive* pair of g_0, from process P to process Q, where s is in G but r is not. Similarly, let s' and r' be a similar pair of g_{nk+2}. We can now construct a causal (according to $<^*$) chain of events in the subpath of σ as follows: from the σ_i cycle we select a *send–receive* pair according to the ith edge of μ. (We may not assume that a chain of events appears according to the order in μ in one cycle σ_i, hence we need to form the chain by collecting events from different cycles.) This forms a causal chain of events, as each *receive* selected precedes the following *send* on the same process line. The first *send* on this chain appears later than the event r. It appears in g_1 and both belong to process Q. The last *receive* precedes the event s'. Both events belong to process P. According to our assumptions, r is not included in G while s' is included. Thus by our construction, $r <^* s'$. This is a contradiction, since a global state must be history closed. ∎

This result is also related to the star problem in trace languages [28]. The restriction to regular HMSCs is quite strong, for instance the simple protocol in Figure 4 is not regular. However, this HMSC is *globally-cooperative*, and belongs to a large subclass of HMSCs with a decidable model-checking problem.

Definition 5.5 *[13] An HMSC H is* globally-cooperative, *if for each loop σ in the graph of H, the communication graph CG^M of the MSC M labeling ρ is weakly connected.*

It is interesting to note that regular HMSCs are precisely globally-cooperative HMSCs that use only bounded channels.

Model-checking globally-cooperative HMSCs is decidable, and has the same complexity as for regular HMSC [13]. Instead of having a regular set of linearizations, globally-cooperative HMSCs have a regular set of *representative* linearizations, which suffice for doing model-checking operations.

Theorem 5.6 *[13] Checking intersection of two globally-cooperative (regular, resp.) HMSCs is PSPACE-complete. Checking inclusion of two globally-cooperative (regular, resp.) HMSCs is EXPSPACE-complete.*

Allowing 'gaps' in the semantics of the specification HMSC gives another decidable case for model checking. A *specification* HMSC representing the bad executions is interpreted in a different way than the HMSC representing the system. The former represents only part of the events. In particular, two adjacent events a and b on the same process line of the specification HMSC may match some nonadjacent events of the same type in the system HMSC. The (scattered) pattern matching problem between these two HMSCs is decidable, and is in NP-hard, in the size of the HMSCs [27].

5.2 The Logic TLC$^-$

Using a partial order based specification formalism can also regain decidability of model checking. Consider a specification that has a language $\mathcal{L}$ that is regular and is already closed under the permutation rules. The emptiness of the intersection of such a specification with an HMSC language can be decided. The reason is that an HMSC language $[P]$ is generated from a regular language P by closing it under permutations. If $\mathcal{L} = [\mathcal{L}]$, then $\mathcal{L} \cap P \neq \emptyset$ iff $\mathcal{L} \cap [P] \neq \emptyset$. Thus, it is sufficient to check the emptiness of the intersection of $\mathcal{L}$ with the regular generator P of the HMSC language. Similarly, for the inclusion problem we have $P \subseteq \mathcal{L}$ iff $[P] \subseteq \mathcal{L}$ and this can be decided, provided that the specification $\mathcal{L}$ is complementable[3].

A solution that involves partial order based formalisms is the use of a subset of the logic TLC [4], as applied on HMSCs in [29]. According to this solution, we use temporal modalities to reason over the events of the MSC system. We use the same modalities symbols as in LTL, but give them a different interpretation; over paths of events, generated by the $<$ relation, rather than over linearizations of the partial order.

The logic TLC$^-$ is a subset of the logic TLC [4]. A model of the logic is a finite or infinite partial order $\zeta = (V, <, \longrightarrow)$, where $<\subset V \times V$ is a partial order relation, and $\longrightarrow \subset <$ is the 'immediately precedes' relation. The set of formulas $\mathcal{L}$ of TLC$^-$ over a set of atomic formulas AP is as follows: *true*, *false* $\in \mathcal{L}$, if $p \in AP$, then $p \in \mathcal{L}$, and if φ, ψ are in $\mathcal{L}$ then $\varphi \wedge \psi$, $\varphi \vee \psi$, $\neg\varphi$, $\exists \bigcirc \varphi$, $\forall \bigcirc \varphi$, $\varphi U \psi$, $\varphi R \psi \in \mathcal{L}$.

An *interpretation function* $I : V \mapsto 2^{AP}$ assigns to each event of V a set of propositions from AP. Each proposition in AP represents some property (e.g., of an event, or the local state before or after the event, when the events are taken from some system execution). Then, $I(v)$ returns the set of atomic propositions that hold for v. The semantics of the logic is defined as follows.

$(\zeta, v) \models$ *true*.

$(\zeta, v) \models p$ if $p \in I(v)$

$(\zeta, v) \models \varphi \wedge \psi$ if $(\zeta, v) \models \varphi$ and $(\zeta, v) \models \psi$.

$(\zeta, v) \models \neg\varphi$ if it is not the case that $(\zeta, v) \models \varphi$.

$(\zeta, v) \models \exists \bigcirc \varphi$ if for some w such that $v \longrightarrow w$, it holds that $(\zeta, w) \models \varphi$.

[3] As in the case of logics, as described next. Note however that HMSCs cannot be complemented.

$(\zeta, v) \models \varphi U \psi$ if there is a path $v = v_0 \longrightarrow v_1 \longrightarrow \ldots \longrightarrow v_n$, such that $(\zeta, v_n) \models \psi$, and for $0 \leq i < n$, $(\zeta, v_i) \models \varphi$.

We define *false* $\equiv \neg$*true*, $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$, $\varphi R \psi \equiv \neg(\neg\varphi U \neg\psi)$, $\forall \bigcirc \varphi \equiv \neg\exists \bigcirc \neg\varphi$. Two additional modalities, $\Diamond$ and $\Box$, can be defined in terms of the previous ones: $\Diamond\varphi \equiv$ *true*$U\varphi$, and $\Box\varphi \equiv$ *false*$R\varphi$. For TLC$^-$ we have selected an *existential until* 'U' operator, hence its dual *release* 'R' operator is universal. The full logic TLC contains also a universal *until*, an existential *release*, and a *concurrent with* operator '$||$'. The modalities U and R satisfy the following equations: $\varphi U \psi \equiv \psi \vee (\varphi \wedge \exists \bigcirc \varphi U \psi)$, $\varphi R \psi \equiv \psi \wedge (\varphi \vee \forall \bigcirc \varphi R \psi)$. A TLC$^-$ formula φ can then be interpreted over an HMSC execution M, treated as a partially ordered set of events. We can denote $M \models \varphi$ when M satisfies φ Like in the case of LTL, where satisfaction is extended from a single execution to the collection of executions of a system [25], we can extend TLC$^-$ satisfiability and define $H \models \varphi$ for an HMSC H when $M \models \varphi$ for each execution M of H.

Thus, the assertion $\bigcirc\varphi$ holds for events that have an immediate successor under the relation $<$ for which φ holds. $\Diamond\varphi$ holds for events e from which there is a path according to $<$, leading to some event f for which φ holds (thus, $e <^* f$). Similarly, for $\psi \mathcal{U} \varphi$ to hold for e, we require, that ψ holds for each event along such a path from e to some event f where φ holds. Finally, in order to satisfy the usual duality $\Box\varphi = \neg\Diamond\neg\varphi$, we interpret $\Box\varphi$ as follows: it holds for events e that satisfy that for every event f such that $e <^* f$, φ holds for f.

Some examples for TLC$^-$ specification are as follows:

$\Box(req \rightarrow \Diamond ack)$ Every request is causally followed by an acknowledgement.

$\Box(recA \rightarrow \exists \bigcirc sendB)$ A message B is sent immediately after receiving a message A.

$\neg\Diamond(tranA \wedge \Diamond(tranB \wedge \Diamond tranA))$ Transaction B cannot interfere with the events of transaction A.

$\Box(beginA \rightarrow \exists \bigcirc (tranA\, U\, finishA))$ The execution of transaction A is not interrupted by any other event.

One intuition behind the decidability (and model checking algorithm) of TLC$^-$ over HMSC is that although HMSC linearizations are not regular languages, they are 'almost regular', up to some commutations, as shown in Section 3. The TLC$^-$ logic does not distinguish between linearizations that are equivalent up to such commutations. A TLC$^-$ formula can thus be equally be interpreted over a regular subset of *representatives* linearizations. More precisely, for the permutation rules the situation is actually a little bit subtler than in Section 3. The reason is that from a TLC$^-$ formula we cannot get the set of *all* linearizations of its MSC models, since this would involve counting of pending messages. We can compute instead the set of all linearizations where is the number of pending messages is bounded. The bound can be provided by the HMSC that is model-checked. Another decidable model checking solution with the same flavor is based on using second order monadic logic over partial orders [22].

6 Other Decision Problems

A natural problem that arises with MSCs is whether the MSCs contain *race conditions*. A race condition can result from the fact that we have only a limited control on the order between pairs of events that include at least one receive event (except for two

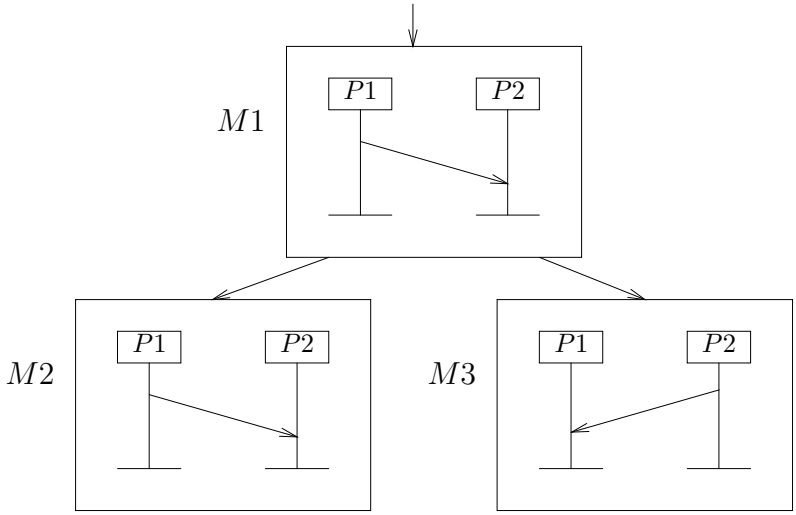


Figure 11: A non-local choice

receives corresponding to messages sent from the same process, according to the FIFO semantics). For example, the MSC in Figure 1 contains two receive events of process P_1 (of messages d_5 and d_6). Since each process line is one dimensional, the MSC notation forces choosing one of the receive events to appear above the other. However, these two messages were sent from different processes, P_2 and P_3, and it might happen that d_6 arrives quicker than d_5. Thus, there is no reason to trust that these messages will arrive in the particular order depicted using the MSC.

Formally, we can define a race condition for pairs of MSC *receive* events $p, q \in V$ for messages sent from different processes such that $L(p) = L(q)$, i.e., p and q appear on the same process line. A race occurs if $p < q$, i.e., p appears above q on the process line, and it is *not* the case that $p <^* q$, i.e., there is no path from p to q according to the relation $<$. Detecting races in an MSC is thus simple. All we need is to calculate the transitive closure $<^*$ and compare it against relation $<$.

It is shown in [3] that the calculation of the transitive closure $<^*$ of $<$ is quadratic in the number of events, and not cubic as is the general case for transitive closure. This stem from the fact that the number of *immediate* successors of each event p under $<$ (i.e., events q such that $p < q$, and there is no r such that $p < r < q$) if limited to 2.

We can define the race conditions for HMSCs. This turns out to be an undecidable problem [27]. We regain decidability by limiting the structure of the HMSCs, as described in Section 5.

Another problem related to HMSC specification is that of *non-local branching choice* [6, 26]. A problem potentially arises when different processes behave according to different choices in the HMSC graph, resulting in a behavior that is not following any of the branching choices.

Consider the example in Figure 11. After Process P_1 sends a message to Process P_2 in M_1 it may proceed according to M_2 and send another message to P_2. However, the HMSC allows also the possibility that after receiving the message in M_1, P_2 would send some acknowledge message, according to the node M_3. If P_1 proceeds according to M_2 and P_2 proceeds according to M_3, we obtain a behavior that is not consistent

with any path of the HMSC.

The definition of non-local branching choice is difficult because it is not clear what would constitute a problematic behavior. In the above example, it is possible that P_1 initially decides on the choice, and lets P_2 know about it through the message that is sends in M_1. On the other hand, it could be argued that in that case, we should have split M_1 into two nodes, according to the branch into M_2 and M_3. One solution is to try and detect whether some non-local choice occurs, while another is to restrict the HMSCs so that they would not allow such a choice [6, 26].In the first case consider *local-choice* HMSCs, i.e., HMSCs that do not have any non-local branching choice. Such specifications are very interesting, since they can be implemented without deadlock by CFMs [13] with additional control data. Although local-choice is a syntactic property, it can be decided whether an HMSC is equivalent to a local-choice HMSC [10].

The problem of implementing HMSCs by CFM has deserved a lot of attention in past years, since it represents an important validation step when using HMSC specifications. The implementation notion used in [1] assumes that the CFM does not use additional data or messages compared to the HMSC. Unfortunately, this notion is not decidable in general, even for regular HMSCs [2], or very expensive if we ask for deadlock-free implementations [21]. The paper [16] shows that local-choice HMSCs cannot be implemented without deadlock if no control (message) data is allowed. For regular HMSCs [24] and globally-cooperative HMSCs [13] implementations with additional (bounded) control data have been proposed.

Acknowledgement

We would like to thank our collaborators in studying message sequence charts: Rajeev Alur, Blaise Genest, Elsa Gunter, Gerard Holzmann, Markus Lohrey and Zhendong Su for many fruitful and pleasant collaborations. We would also like to thank the two anonymous reviewers of this paper for their insightful remarks, which helped to improve the presentation.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. of the 22nd Int. Conf. on Software Engineering*, 304–313, ACM, 2000.

[2] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *Proc. of (ICALP'01)*, LNCS 2076, pp. 797–808, 2001.

[3] R. Alur, G. H. Holzmann, and D. A. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[4] R. Alur and D. Peled and W. Penczek. Model Checking of Causality Properties. In *Proc. of Logic in Computer Science (LICS'95)*, pp. 90-100, 1995.

[5] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. of CONCUR'99*, LNCS 1664, pp. 114–129, 1999.

[6] H. Ben-Abdulla, S. Leue. Symbolic Detection of Process Divergence and Non-local Choice in Message Sequence Charts. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, LNCS 1217, pp. 259–274, 1997.

[7] E. M. Clarke, O. Grumberg, D. Peled, *Model Checking*, MIT Press, 1999.

[8] J. Esparza, D. Hansel, P. Rossmanith, S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. of CAV'00*, LNCS 1855, pp. 232-247, 2000.

[9] B. Genest and A. Muscholl. Pattern matching and membership for Hierarchical Message Sequence Charts. In *Proc. of LATIN 2002*, LNCS 2286, pp. 326–340, 2002.

[10] B. Genest and A. Muscholl. Don't choose globally, implement easily. Internal LIAFA report, 2003.

[11] E. Gunter, A. Muscholl, D. Peled, Compositional Message Sequence Charts. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, 496–511. Journal version *International Journal on Software Tools for Technology Transfer (STTT)* 5(1): 78-89 (2003).

[12] B. Genest, A. Muscholl, and D. Peled. Message sequence charts. In *Lectures on Concurrency and Petri Nets*, LNCS 3098, pp. 537–558, 2003.

[13] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state High-level MSCs: Model-checking and realizability. In *Proc. of ICALP'02*, LNCS 2380, pp. 657–668, 2002. Journal version to appear in *JCSS*.

[14] G. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, 1992.

[15] ITU-T Recommendation Z.120, Message Sequence Chart (MSC), 1996.

[16] L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from HMSCs. In *5th International Workshop on Formal Methods for Industrial Critical Systems*, Berlin, 2000.

[17] J. G. Henriksen, M. Mukund, K. Narayan Kumar, and P.S. Thiagarajan. On Message Sequence Graphs and finitely generated regular MSC languages. In *Proc. of ICALP'00*, LNCS 1853, pp. 675–686, 2000.

[18] J. G. Henriksen, M. Mukund, K. Narayan Kumar, and P.S. Thiagarajan. Regular collections of message sequence charts. *Proc. of MFCS'00*, LNCS 1893, pp. 405–414, 2000.

[19] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[20] D. Kuske. Regular sets of infinite message sequence charts. *Information and Computation*, (187):80–109, 2003.

[21] M. Lohrey. Safe realizability of high-level message sequence charts. In *Proc. of CONCUR'02)*, LNCS 2421, pp. 177–192, 2002.

[22] P. Madhusudan. Reasoning about sequential and branching behaviours of message sequence graphs. In *Proc. of ICALP'01*, LNCS 2076, pp. 809–820, 2001.

[23] A. Mazurkiewicz, Basic notions of trace theory, REX workshop 1988, pp. 285–363, LNCS 354, Springer, 1988.

[24] M. Mukund, K. Narayan Kumar, and M. Sohoni. Synthesizing distributed finite-state systems from MSCs. In *Proc. of CONCUR'00*, LNCS 1877, pp. 521–535, 2000.

[25] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1991.

[26] A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz In *Proc. of MFCS'99*, LNCS 1672, pp. 81–91, 1999.

[27] A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Proc. of FoSSaCS'98*, LNCS 1378, pp. 226–242, 1998.

[28] E. Ochmanski. Recognizable trace languages. In *The Book of Traces*, V. Diekert, G. Rozenberg, (eds.), World Scientific, 1995, 167–204.

[29] D. Peled. Specification and verification of message sequence charts. In *Proc. of FORTE/PSTV 2000*, 2000.

[30] M. Y. Vardi, P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of Logic in Computer Science (LICS'86)*, pp. 332–344, 1986.