Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Compositional Message Sequence Charts

Elsa L. Gunter, Anca Muscholl and Doron Peled

[1] Department of Computer Science
New Jersey Institute of Technology
Newark, NJ 07102, USA

[2] LIAFA, Université Paris 7
2, place Jussieu, case 7014
75251 Paris Cedex 05, France

[3] Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712, USA

Received: date / Revised version: date

Abstract. A Message sequence chart (MSC) is a standard notation for describing the interaction between communicating objects. It is popular among the designers of communication protocols. MSCs enjoy both a visual and a textual representation. High-level MSCs (HMSCs) allow specifying infinite scenarios and different choices. Specifically, an HMSC consists of a graph, where each node is a finite MSC with *matched* send and receive events, and vice versa. In this paper we demonstrate a weakness of HMSCs, which disallows one to model certain interactions. We will show, by means of an example, that some simple finite state communication protocol cannot be represented using HMSCs. We then propose an extension to the MSC standard, which allows HMSC nodes to include unmatched messages. The corresponding graph notation will be called HCMSC, which stands for High-level *Compositional* Message Sequence Charts. With the extended framework, we provide an algorithm for automatically constructing an MSC representation for finite state asynchronous message passing protocols.

Keywords: Message Sequence Charts, finite-state communication protocols, partial order methods.

1 Introduction

It is commonly agreed that catching software bugs is a highly important task. On the other hand, people that develop formal methods techniques and tools often find it very hard to transfer their technology into the software development industry. One of the reasons of this apparent contradiction is that formal methods research often result in formalisms that are different than the ones already used by software designers and developers. Some of the formalisms, such as temporal logic or the Z notation, have some moderate success outside the community of developers of formal methods. However, the bulk of software specification is still done informally, while the most commonly used method for software reliability is testing.

One partial solution for this situation is the use of visual formalisms. These are supposed to be more intuitive than textual formalisms, in the sense that "a picture is worth a thousand words". Describing software in different ways, from different perspectives, and by emphasizing different aspects, can certainly lead to a better understanding, and perhaps to some new insight. This is also the motivation for the design of the *Unified Modeling Language* (UML).

Another group of formalisms are the ITU and ISO standards, including SDL [10] and LOTOS [5]. These formalisms have both a textual as well as a visual representation. We will focus here on the ITU message sequence chart (MSC) notation. The MSC formalism is often used in the design of communication protocols and was developed in conjunction with the SDL standard. A similar formalism is used in UML by describing interactions between objects. MSCs allow abstracting away parts of the system that are less relevant for a particular view, thus being perfectly suited for specifying abstractly the required features. MSCs are among the most frequently used formalisms for designing communication protocols. They have been known for a long time as sequence or timing diagrams. In the last few years, we observe a growing interest for the development of tools and algorithms for the manipulation of MSC based designs [1,2, 4,11,17,18], mainly motivated by their integration into UML.

The standard visual and textual notation [13] by ITU allows representing a single execution scenario, as well as a collection of scenarios, including choices and repetition. This is achieved by a notation called *High-level Message Sequence Chart* (HMSC), which consists of a graph, where each node contains a single MSC. The system's behavior can follow the paths on that graph, starting from some initial node.

MSCs are based on partial order semantics and do not impose any limitation on buffers (except for the often used convention that buffers deliver messages in a FIFO way). Therefore, HMSCs have infinite state space due to concurrency of events and to unbounded buffers. This makes some verification problems such as model-checking [4] or detecting race conditions [17] undecidable, even without exploiting the potentially unbounded buffers.

One possible solution is to restrict HMSCs in order to get a finite state space. To this purpose, bounded (locally-synchronized, respectively) HMSCs have been proposed in [4,17]. This restriction makes automatic verification feasible but the price to pay is the high complexity of the algorithms. For instance, positive model-checking (inclusion) of bounded HMSCs is EXPSPACE-complete [17], and negative model-checking (intersection) of bounded HMSCs is PSPACE-complete [4,17].

Another drawback of bounded HMSCs is that they impose a limit on the buffer size, which may be an unwanted limitation at the level of abstraction in which HMSCs are used. One alternative solution for making model-checking decidable is to use partial-order temporal logics for specifying the desired property of an HMSC. Work done in [21] shows how to do model checking on a fragment of the local logic TLC [3], whereas [15] uses monadic second order logic on configurations of the partial order of executions of HMSCs.

In this paper we consider a problem with the expressiveness of HMSCs in the ITU standard. We show, by means of an example, a limitation of HMSCs. This limitation stems from the constraint that each MSC node in an HMSC must have only *matched send* and *receive* events. That is, each node must contain both the *send* and *receive* events of each message. We show a typical example of a real communication protocol where one cannot break a possibly infinite computation of a finite state system into finitely many nodes with matched communication events. (A finite execution can always be represented as a single node.) It is interesting to note that our counterexample holds even with bounded communication channels, i.e., it does not make use of the potentially unbounded communication channels. To circumvent this weakness of HMSCs, we suggest an extension to the MSC (HMSC, respectively) standard, called *Compositional Message Sequence Charts* (CMSC and HCMSC, respectively). This extension allows specifying MSCs with unmatched *send* and *receive* events. The semantics of the new construct prescribes how to combine such MSCs together. The semantics is simple. It matches unmatched *send* events in the current node with unmatched receive events in succeeding nodes on a given execution path. We use the extended notation to suggest an algorithm for the automatic generation of HCMSC representations for finite state systems.

We discuss a problem of HCMSCs, and show that the definition is, in some sense, too general. Specifically, it allows scenarios where some unmatched *receive* does not correspond to any previous unmatched *send*. For that, we propose here the use of a restricted class of HCMSCs, called *realizable* HCMSCs. We show how to test whether an HCMSC is realizable in an efficient way. The notion of realizable HCMSC is quite natural, as our algorithm for the HCMSC generation already yields HCMSCs of this kind.

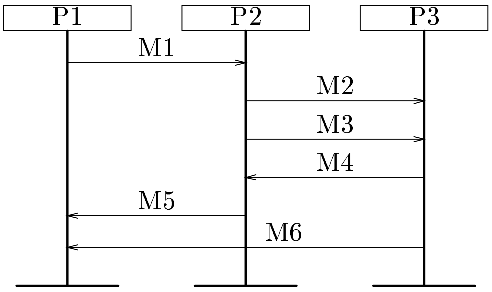
The deficiency of the original MSCs was also recognized in [16]. The solution suggested there is a different extension to HMSCs. According to this extension, one can use parallel components of MSCs, and allow intercommunication between them, using a mechanism called 'gates'. Our solution differs from that of [16], as we study the effect of allowing communication between sequentially composed CMSCs. That is, a communication that starts in one CMSC and ends in a subsequent one. Notice that our solution avoids the need for special message names for the purpose of binding, as in [16]. Further papers considering this issue are [12,19]. These papers look at the problem of checking whether a finite state protocol can be translated into an HMSC. In the first of these papers, it is shown that this question is decidable, whereas the second paper shows that for a natural class of finite state protocols it can be efficiently checked whether the translation into an equivalent HMSC is possible. The present paper completes this picture, by showing how to express any finite state communication protocol in the framework of Compositional HMSCs.

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, while a message is represented by a horizontal or slanted arrow from the sending process to the receiving one, as appears in the upper half of Figure 1. The corresponding ITU Z120 textual representation of the MSC appears in the lower half of Figure 1. The semantics of MSCs is defined as follows.

Definition 1. An MSC M is given as a tuple $\langle V, <, \mathcal{P}, \mathcal{N}, L, T, N, m\rangle$, where

- V is a finite set of events,
- $< \subseteq V \times V$ is an acyclic relation,
- $\mathcal{P}$ is a set of processes,
- $\mathcal{N}$ is a set of *message names*,




```
msc MSC;
inst P1: process Root,
P2: process Root,
P3: process Root;
instance P1;
out M1 to P2;
in M5 from P2;
in M6 from P3;
endinstance;
instance P2;
in M1 from P1;
out M2 to P3;
out M3 to P3;
in M4 from P3;
out M5 to P1;
endinstance;
instance P3;
in M2 from P2;
in M3 from P2;
out M4 to P2;
out M6 to P1;
endinstance;
endmsc;
```

Fig. 1. Visual and textual representation of an MSC

- $L : V \to \mathcal{P}$ is a mapping that associates each event with a process,
- $T : V \to \{\mathrm{s}, \mathrm{r}, \mathrm{l}\}$ is a mapping that describes 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 partial function called *matching* that pairs up *send* and *receive* events. Each *send* is paired up with exactly one *receive* and vice versa. Events v_1 and v_2 can be paired up with each other, only if $N(v_1) = N(v_2)$.

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. 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.,

$$\begin{aligned}&(T(v_1) = T(v_2) = \mathrm{s} \wedge T(v_1') = T(v_2') = \mathrm{r}\\ &\wedge m(v_1, v_1') \wedge m(v_2, v_2') \wedge\\ &L(v_1) = L(v_2) \wedge L(v_1') = L(v_2') \wedge v_1 < v_2) \Rightarrow v_1' < v_2'\end{aligned}$$

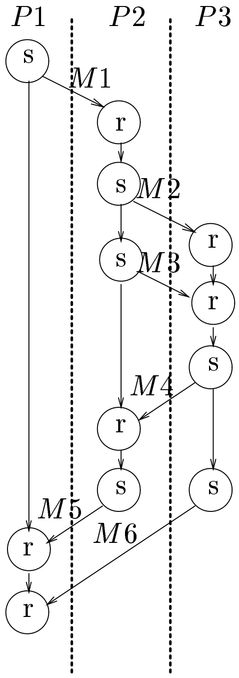


Fig. 2. The partial order between the events of the MSC in Figure 1.

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. That is, u immediately precedes v. 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 $\longrightarrow$. 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. One of the earliest use of such a notation for partial order between events executed in an asynchronous message passing system is by Lamport [14]. A *linearization* of an MSC $M = \langle V, <, \mathcal{P}, \mathcal{N}, L, T, N, m\rangle$ is a total order on V, which extends the relation $(V, <)$.

Example 1. Consider the example MSC given in Figure 1. Denote by v_i the *send* event and by w_i the receive event of message Mi, $1 \leq i \leq 6$. Then we have $V = \{v_1, \ldots, v_6, w_1, \ldots, w_6\}$, $\mathcal{P} = \{P1, P2, P3\}$, $\mathcal{N} = \{M1, \ldots, M6\}$ and $N(v_i) = N(w_i) = Mi$ for all i. The events located on $P1$ are $L^{-1}(P1) = \{v_1, w_5, w_6\}$, with $T(v_1) = \mathrm{s}$, $T(w_5) = T(w_6) = \mathrm{r}$, and $v_1 < w_5 < w_6$. This ordering is the time ordering of events on $P1$. We also have $m(v_i, w_i)$ and $v_i < w_i$ for all i (message ordering). In particular, $v_1 < w_1 < v_2 < w_2$.

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, $M5$ and $M6$, respectively, are unordered.

Definition 2. The *concatenation* $M_1 M_2$ of two MSCs $M_k = \langle V_k, <_k, \mathcal{P}, \mathcal{N}_k, L_k, T_k, N_k, m_k\rangle$ over the same 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 $\langle V_1 \cup V_2, <, \mathcal{P}, \mathcal{N}_1 \cup \mathcal{N}_2, L_1 \cup L_2, T_1 \cup T_2, N_1 \cup N_2, m_1 \cup m_2\rangle$, 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 (but some events in M_1 of one process may be unordered with respect to some events in M_2 of another process). 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 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 $M1$, while another process has advanced to events from $M2$. The infinite concatenation of finite MSCs is defined in a similar way, and it allows defining infinite MSCs as well.

Definition 3. Let M_1, M_2, ... be an infinite sequence of finite MSCs. Define a sequence $M_1{}'$, $M_2{}'$, ... as follows: Let $M_1{}' = M_1$, and for $i > 1$, $M_i{}' = M'_{i-1} M_i$. (Thus, if $i < j$, $M_i{}' \sqsubseteq M_j{}'$.)

Let $M'_i = \langle V_i, <_i, \mathcal{P}, \mathcal{N}_i, L_i, T_i, N_i, m_i\rangle$. Then the infinite concatenation $M_1 M_2 \ldots$ is defined as the infinite MSC $M = \langle V, <, \mathcal{P}, \mathcal{N}, L, T, N, m\rangle$ where $V = \cup_{i \geq 1} V_i$ is the disjoint union of the V_i, $\mathcal{N} = \cup_{i\geq 1}\mathcal{N}_i$, $L|_{V_i} = L_i$, $T|_{V_i} = T_i$, $N|_{V_i} = N_i$ ($T|_{V_i}$ and $N|_{V_i}$ denote the functions T and N, respectively, restricted to the domain V_i), $m = \cup_{i\geq 1} m_i$ and

$$\begin{aligned} < \, = \, & \cup_{i\geq 1} <_i \; \cup \\ & \{(u,v) \mid L_i(u) = L_j(v) \wedge u \in V_i \wedge v \in V_j \wedge i < j\} . \end{aligned}$$

(Note that in the last line, the first use of '<' refers to the relation between events that is defined here, while the last '<' is the usual 'smaller than' relation between natural numbers.)

Since a communication system usually includes many (or even infinitely many) such scenarios, a high level description is needed for combining them together. The standard description consists of a graph called HMSC (high-level MSC), where each node contains one 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*. Such an execution can be used to denote the communication structure of a typical (aka 'sunny day') or an exceptional (aka 'rainy day') behavior of a system, or a counterexample found during testing or model checking.

Definition 4. An HMSC N 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 $c : \mathcal{S} \to \mathcal{M}$ associates the node s with an MSC $c(s)$. 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 N is a (finite or infinite) MSC $c(g_0)\, c(g_1)\, c(g_2) \cdots$ associated with a maximal path $g_0, g_1, \ldots$ of N that starts with some initial node $g_0 \in \mathcal{S}_0$.

Figure 3 shows an example of an HMSC where the node in the upper left corner is the starting node. Initially, process P_1 sends a message to P_2, requesting a connection (e.g., to an internet service), according to the top left box. This can result in either an approval message from P_2, according to the top right box, or a failure message, according to the bottom left box. 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 the bottom left box. We can decide to try and connect again, by choosing the up arrow, or to give up and send a service request (from process P_1 to process P_3), by choosing the left-to-right arrow. Notice 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. Note that according to HMSC semantics, process $P2$ in Figure 3 does not necessarily have to send its Report message before the execution of process $P1$ has progressed into the next node and has sent its Req_service message. However, process P3 must receive the Report message before the Req_service message.

According to the ITU standard [13], 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 paper.

3 MSC Decomposition

The HMSC model combines the visual notation of message sequence charts with the ability to describe repetitions and alternative computations. In this section we will show that this, seemingly powerful model, cannot describe some basic finite state communication protocols. The main problem lies within the requirement that the *send* and *receive* events in each node must be matched.

We want to exemplify that there are finite state protocols that do not allow a finite HMSC representation. To do that, we show an infinite execution ξ of a finite state protocol with the following property: there is no way to write ξ as an infinite concatenation of finite MSCs. Given the above property, it is not possible to construct an HMSC such that ξ would correspond to a traversal of one of the HMSC paths. Thus, we cannot represent such a system using HMSCs.

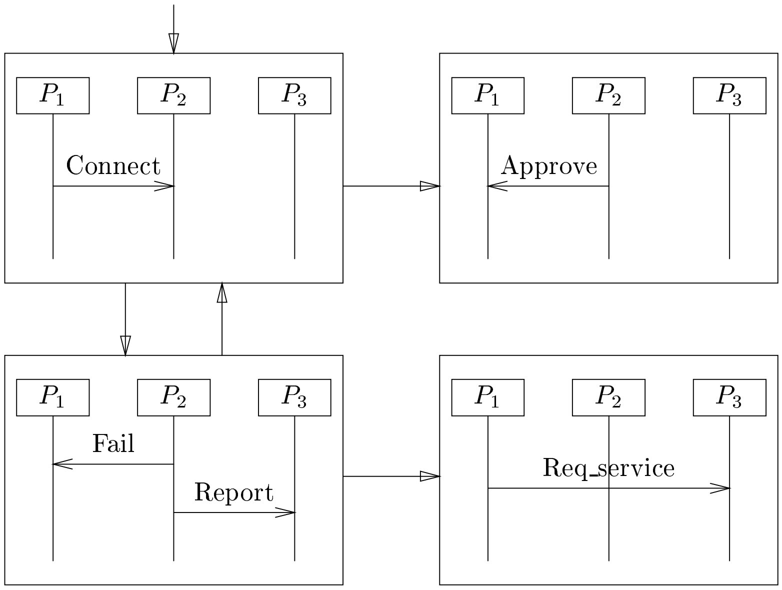


Fig. 3. An HMSC graph

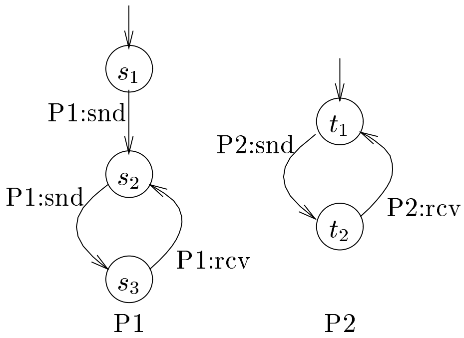


Fig. 4. A simple two process protocol

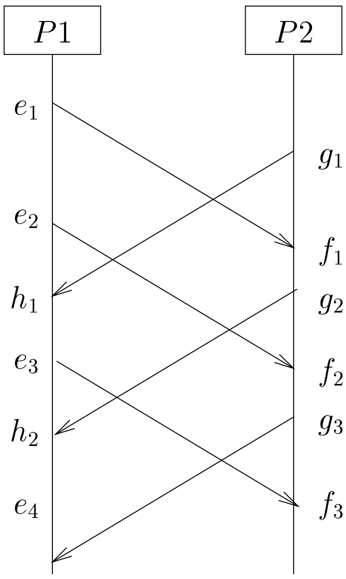


Fig. 5. A prefix of an MSC execution that cannot be decomposed

As an example, consider the infinite MSC that is generated from the simple protocol in Figure 4. A finite prefix of the MSC behavior of this protocol appears in Figure 5. 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 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 must be in the same HMSC node with e_1 and f_1. 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.

While the repeated crossing of message edges seems to be atypical for MSCs, the above behavior ξ describes a possible execution of an actual protocol (alternating bit, [22]), where messages and acknowledgments are being sent between two processes, with (bounded) buffering.

4 Compositional MSC

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 5. A CMSC M is defined as in Definition 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.
- Fix an ordered pair of processes P_i and P_j for the CMSC. Let S_m (S_u, respectively) be the matched (unmatched, respectively) *send* events from P_i to P_j, and R_m (R_u, respectively) be the set of matched (unmatched, respectively) *receive* events from P_i to P_j (some of these sets can be empty). Then we have that the events of R_u precede those of R_m (according to the order '$<$' associated with the CMSC), and the events S_m precede those of S_u.

The reason for the restrictions imposed in the definition of a CMSC is that unmatched *send* events must 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 6, we can see an HCMSC that represents the execution that is approximated in Figure 5.

Definition 6. A CMSC is *left-closed*, if it does not contain unmatched *receive* events, or any *send* events that are not yet matched and precede another matched *send* of the same type.

Note that the without the latter restriction of this definition, we could have *send* events that could never be matched without violating the FIFO order.

Definition 7. Consider two CMSCs $M_1 = \langle V_1, <_1, \mathcal{P}, \mathcal{N}_1, L_1, T_1, N_1, m_1\rangle$ and $M_2 = \langle V_2, <_2, \mathcal{P}, \mathcal{N}_2, L_2, T_2, N_2, m_2\rangle$ 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_1M_2 is then defined as $\langle V_1 \cup V_2, <, \mathcal{P}, \mathcal{N}_1 \cup \mathcal{N}_2, L_1 \cup L_2, T_1 \cup T_2, N_1 \cup N_2, m_1 \cup m_2 \cup m'\rangle$, 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

1. M_1 is left-closed, and
2. $M_1\, M_2$ is a CMSC satisfying the FIFO property when restricting the events to the matched pairs of events.

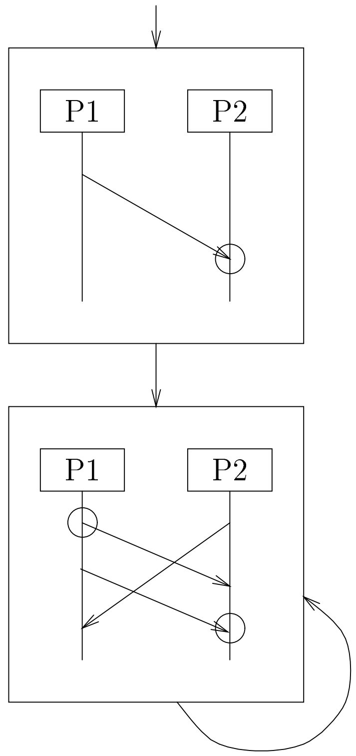


Fig. 6. A decomposition of the execution in Figure 5.

Notice that the concatenation $M_1 M_2$ can be defined, but not left-closed as required in Definition 6. Clearly, the concatenation of CMSCs is not associative anymore. Hence, when we write $M_1 \cdots M_k$ we mean the concatenation $(\cdots(M_1M_2)M_3)\cdots M_k)$.

Again, we can define $M_1 \sqsubseteq M$ 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 3. Note that in an infinite concatenation, there can be infinitely many unmatched messages sent from one process to another.

An HCMSC is a graph whose nodes are CMSCs, similarly to Definition 4. 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 4, with the additional constraint that every finite prefix of this path results in a left-closed CMSC.

It is important to note that at this point, we allow paths in the HCMSC graph that start with an initial node and have prefixes that, when concatenated, do not result in a left-closed CMSC. Such paths are simply not counted as executions. It is important to observe that an empty arrow head or tail does not correspond to an actual *send* or *receive* event, and the actual event may exist in a preceding or subsequent CMSC.

5 Undecidability

Extending the MSC standard allows representing the execution of a bigger class of protocols than what is allowed by the ITU standard. However, with the added expressiveness we loose some of the power of analyzing such systems.

Unlike simple HMSCs, where some simple properties can be checked, see e.g., [18], in HCMSCs one cannot decide even the trivial property of whether a particular message can be received in at least one execution.

Theorem 1. *The problem of checking whether a particular event appears in at least one execution of an HCMSC is undecidable.*

Proof. The undecidability proof will be a reduction from Post Correspondence Problem (PCP). An instance of PCP is a set of pairs of words

$$C = \langle (v_1, w_1),\ (v_2, w_2),\ \ldots,\ (v_m, w_m) \rangle$$

over some mutual alphabet Σ. We want to find out if there is some integer $n > 0$ and some sequence of indexes $i_1, i_2, \ldots, i_n$ such that $v_{i_1} v_{i_2} \cdots v_{i_n} = w_{i_1} w_{i_2} \cdots w_{i_n}$. We will construct an HCMSC with four processes P_1 to P_4, and with CMSC nodes E_1, E_2, ..., E_m, E'_1, E'_2, ..., E'_m, F.

- Messages from P_1 to P_2 correspond to the letters of Σ. Each CMSC E_i contains a sequence of unmatched *send* events from P_1 to P_2, representing the sequence of messages of v_i. Each CMSC $E_i{}'$ contains a sequence of unmatched *receive* events from P_1 to P_2, representing the sequence of messages of w_i.
- Messages from P_3 to P_4 correspond to the index of the PCP word being sent. Each CMSC E_i contains also a single unmatched *send* from P_3 to P_4 labeled by the current index i. Each CMSC $E_i{}'$ contains the corresponding unmatched *receive* event.

The node F contains a message from P_1 to P_2, one from P_3 to P_4, and one from P_2 to P_4. The HCMSC has the form $(E_1 + \cdots + E_m)^+ (E'_1 + \cdots + E'_m)^+ F$. Thus, we repeatedly take nodes of the form E_i. Then we take nodes of the form E'_i, followed by the node F. Note that from Definition 6, if any of the a matched messages of node F appear in an execution, it cannot follow a *receive* or *send* that were not yet matched. Thus, the *receive* event for the message from P_2 to P_4 appears in a CMSC execution only if there is a nonempty solution to the PCP instance. □

It is important to note that in the construction for the above proof, we obtain an HCMSC that has some paths that do not correspond to executions. That is, paths with some finite prefix which is not left-closed.

6 Realizable HCMSC

The definition of HCMSCs suffers from several deficiencies. One is the undecidability of even the simplest problems, as demonstrated in the previous section. The other problem is that we can obtain some "unreasonable" paths in HCMSCs, 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, which was taken previous (see last paragraph of Section 4) is to disregard them as executions of the HCMSC system. Another approach, which will be taken in this section, is to forbid HCMSCs with such paths.

Definition 8. *Realizable HCMSC* are a subclass of HCMSCs where all maximal executions starting in an initial node define left-closed CMSC.

Note that we explicitly allow executions with unmatched *send* events. For example, the HCMSC of Figure 6 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. Moreover, Definition 6 of left-closedness guarantees that an unmatched *send* cannot be executed in a way that prevents the system from matching it later in a FIFO order. The decision problem of Section 5 becomes now trivial for HCMSCs: all the maximal paths starting from an initial node are executions, and thus, every event occurs in at least one execution.

Definition 9. An HCMSC is *realizable* if the execution of every finite path starting with the initial state is a left-closed CMSC.

We will show how to test whether an HCMSC is realizable. From the definition of realizable HCMSCs, we can focus on messages sent from each P_i to another process P_j separately. There are three situations that violate the realizability 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 realizable, we construct a 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 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 realizable 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* realizable. Using the reachability algorithm for pushdown systems given in [6] we obtain the following:

Theorem 2. *Let M be an HCMSC with n events and let $n_{i,j}$ be the number of events (send and receive) from process P_i to P_j. Then it can checked whether M is realizable in time $O(\Sigma_{i,j} n_{i,j}^2) = O(n^2)$.*

7 An HCMSC Representation for Finite State Systems

The HCMSC formalism suggested in this paper broadens the scope of HMSCs and allows us to capture many more protocols. In particular, as we see below, we can represent now any finite state protocol. We present an automatic translation from finite state systems with asynchronous message passing to (realizable) HCMSC.

Definition 10. A finite state system $\mathcal{G} = (S, S_0, E, \Sigma)$, includes a finite set of states S, initial states $S_0 \subseteq S$, and finitely many edges $E \subseteq S \times \Sigma \times S$, labeled over a finite set of actions Σ. The actions in Σ are *send*, *receive* and *local* actions. The states in S contain information about the system, including the contents of the various inter-process message queues. A *run* of a finite state system is a maximal path that starts with an initial state.

We start with a trivial translation, which establishes the theoretic possibility of performing such a translation for a class of finite state systems with asynchronous message passing. We later proceed to suggest a more informative translation. The trivial translation is performed by constructing the dual graph $\mathcal{H} = (N, N_0, F)$ of $\mathcal{G}$ as follows:

- The *nodes* N of $\mathcal{H}$ correspond to the *edges* of $\mathcal{G}$. That is, $N = E$. The label of a node e is the label of e in $\mathcal{G}$.
- The *initial nodes* $N_0 \subseteq N$ of $\mathcal{H}$ correspond to the edges of $\mathcal{G}$ that exit from the initial states S_0.
- The edges F of $\mathcal{H}$ correspond to pairs of edges of $\mathcal{G}$ such that the target of the first edge is the source of the second.

The above trivial construction does not provide any new insight, since the HCMSC graph follows closely the state space and each CMSC node includes a single local or unmatched event. We thus look into a translation that would construct more reasonable HCMSCs. The translation aims at the following goals:

1. Minimize the number of unmatched events appearing in the individual CMSC nodes. However, recall from Section 3 that this is not always achievable.

2. Present relatively long scenarios with the CMSCs, in order to obtain an intuitive understanding of the interprocess interaction.
3. Minimize the number of individual CMSC blocks, so that the HCMSC would not become too big.

Notice that the second and third goal may contradict each other in some systems. The above 'trivial' translation gives a rather reasonable solution to the third goal, since the number of HCMSC nodes generated is the same as the number of states, while providing unacceptable solution for the first and second goals. Notice further that the size of an HCMSC graph can easily get prohibitively large. Thus, in practice, the HCMSC construction algorithm should be applied only to small parts of communication protocols, rather than to complete protocols.

As we will see later, different execution paths in the state space may correspond to a single CMSC. We would like to eliminate this duplicity. The *partial order reduction* algorithms were constructed for this particular reason. The *sleep set* method [8], adapted to our case, is in particular appropriate.

Converting Finite State Communication to HCMSC

1. Calculate a set $Z \subseteq S$ of 'cutpoints'. These are global states of the system, such that every cycle must pass through one of these nodes. One heuristics is to perform simple DFS on the state space and include in Z every node in the target of a back edge. Notice that this is not optimal (finding a minimal set of such nodes is an NP-complete problem). Since every cycle must pass at least one of these points, paths from Z to Z cannot contain cycles. We may also want to include in this set nodes in which all or most of the queues are empty. This will help in reducing the number of unmatched events in the generated CMSC nodes.
2. Start a *reduced Depth First Search (DFS)* from nodes in Z or initial global states. The search stops at nodes in Z (after progressing at least one step) or to a terminating node. The reduced search algorithm, related to Godefroid's sleep set algorithm [8], and to the variant of that algorithm presented in [20] is shown in Figure 7. During the search, we pair each state s with a set *sleep* of actions, called its *sleep set*. These actions do not need to be explored from the state s, although they can be taken from that state. We start a search from a global state s with $expand\_node(s, \emptyset)$. The variant presented in [20] of the reduction algorithm allows removing nodes that have an empty number of successors under the reduction.[1] Since the number of paths can be quite large, one can split the reduced graph further by adding more nodes in the the set of cut points Z. In this way, we generate shorter paths, but possibly more of them.
3. Construct CMSCs nodes for each paths generated by the search in the above step. In particular, each edge on such a path is labeled as a *send*, *receive* or *local* event of some process. To create the corresponding CMSC node, project the events on the participating process lines.
4. Connect the matching *send* and *receive* events in each CMSC node to each other with arrows. This can be done as follows. The events along each CMSC node are generated in some order that is a linearization of the order '<' between events. Each event starts from a global state of the system (since we are given a finite state space of the system). Each global state includes, for each ordered pair of processes, the number of *send* events that have not been received yet. Consider a *send* event from P_i to P_j that appears immediately after a global state in which n such messages are in transit. Then, this *send* event matches the $(n + 1)$st *receive* event from P_i to P_j, if indeed there are at least $n + 1$ such *receive* events in this CMSC node.
5. Connect the separate CMSCs in the following way: If one CMSC node N ends at some global state $s \in Z$ and another CMSC node N' starts with that global state, make an edge from N to N'.

```
function expand_node(s, sleep);
local explored, working_set, new_sleep, fixed;
    explored := ∅;
    fixed := false;
    Let en(s) be the actions that can be applied from s.
    if en(s) = ∅ then return true fi;
    working_set := en(s) \ sleep ;
    while working_set ≠ ∅ do
      Let α be some action from working_set;
      working_set:=working_set\{α};
      s' := α(s);
      Let dep(α) be actions of the same process as α.
      new_sleep:=(sleep ∪ explored) \ dep(α);
      explored := explored ∪ {α};
      if s' ∈ Z orelse s' is terminal orelse exists_node(s', new_sleep)
            orelse expand_node(s', new_sleep) then
        fixed := true;
        create_edge((s, sleep), α, (s', new_sleep)) fi;
      fi
    end while;
    if fixed then store_node_in_hash(s, sleep);
    return fixed;
end expand_node.
```

Fig. 7. A reduced state space generation algorithm

The use of the reduced search algorithm (in Figure 7) is intended for reducing the number of paths, and hence the number of CMSCs in the representation. The proof of

[1] Another change from the original sleep sets algorithm is that the new nodes are pairs of a global state and a sleep set, and two global states that are paired with different sleep sets are considered different nodes.

correctness and the properties of the sleep set algorithm can be found in [8,20] or in [7].

To demonstrate the algorithm, consider first the state space of the simple protocol in Figure 8. Each state contains the following items:

1. The state of the left process P_1.
2. The state of the right process P_2.
3. The number of messages sent from P_1 to P_2 and not yet received.
4. The number of messages sent from P_2 to P_1 and not yet received.

Each edge is labeled by the type of operation executed (whether it is a *send* or a *receive*, and the target process). We need first to find a set of cut points Z. One solution is the set that includes two points: perform a reduced DFS from the initial node $x = \langle s_1, t_1, 0, 0\rangle$ and the node $y = \langle s_2, t_2, 1, 1\rangle$. Now notice that from x to y there are 2 sequences:

- $\langle s_1, t_1, 0, 0\rangle - \mathrm{P1} : \mathrm{snd} \rightarrow \langle s_2, t_1, 1, 0\rangle - \mathrm{P1} : \mathrm{snd} \rightarrow \langle s_2, t_2, 1, 1\rangle$, and
- $\langle s_1, t_1, 0, 0\rangle - \mathrm{P1} : \mathrm{snd} \rightarrow \langle s_1, t_2, 0, 1\rangle - \mathrm{P1} : \mathrm{snd} \rightarrow \langle s_2, t_2, 1, 1\rangle$.

However, they are just different linearizations of a single CMSC (or partial order) execution. The reduced DFS will use the information about dependency between the executed events to find one representative, which can be converted into a CMSC. The generated CMSC appears in the left upper part of Figure 9. Similarly, from y back to itself, we have 2 sequences, which also correspond to a single $CMSC$ execution. Again, the reduced DFS will find one such sequence. Finally, from y to x there are additional 2 sequences, with again one representative. The generated HCMSC appears in Figure 9.

Advanced comment. Our algorithm has the following additional property: Let ξ be a path from an initial state of the state space $\mathcal{G}$ to one of its global states s. Consider the CMSC M_ξ that includes exactly the events that occurred in ξ. Then there is a path $M_0, M_1, \ldots, M_{n-1}, M_n$, with M_0 an initial node, such that $M_0\, M_1 \ldots M_{n-1} \sqsubseteq M_\xi \sqsubseteq M_0\, M_1 \ldots M_n$. This CMSC represents in some sense the history of the global state s, along the path ξ. Intuitively, this means that every global state in $\mathcal{G}$ is represented by the HCMSC. It is interesting to note that the HCMSC generated from the state space in Figure 8, which appears in Figure 9, the right CMSC together with all the arrows connected to it are redundant. However, if we remove them, the above property would not hold. For example, with respect to paths in $\mathcal{G}$ that end with the global states $\langle s_2, t_2, 2, 0\rangle$ and $\langle s_2, t_2, 0, 2\rangle$.

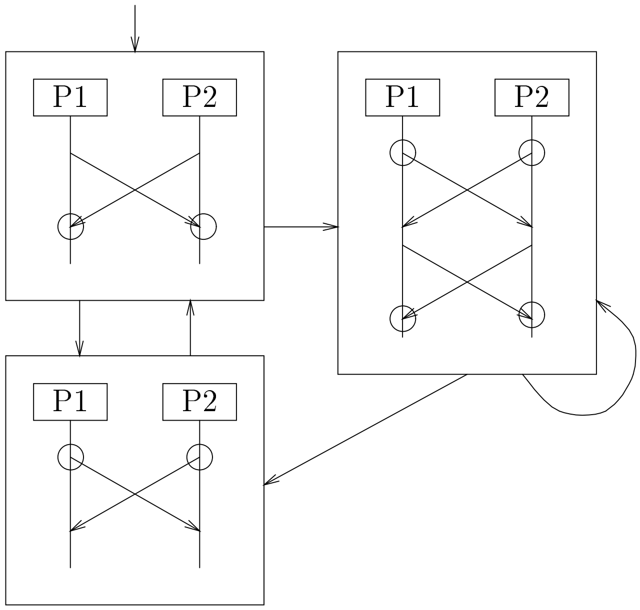


Fig. 9. A generated HCMSC representation for the protocol in Figure 8

8 Conclusion and Implementation

HMSCs are a useful and standard notation for describing executions of communication protocols. We showed that the requirement of pairing up *send* and *receive* events in each MSC node prohibits the representation of a simple finite state protocol. We presented an extension of the HMSC notation, which we call High-level *Compositional* Message Sequence Chart (HCMSC). This notation circumvents this problem. We also showed a 'realizable' version of HCMSCs, which disallows anomalities such as *receive* events that are not matched to any previous *send*.

With this extension, we presented an algorithm for automatically generating the HCMSC structure for finite state communication protocols. We have implemented this algorithm as an extension of the PET system [9]. This system helps with designing test suites for unit testing, and is visually oriented. In particular, it automatically translates programs to the notation of flow charts. With the addition of the HCMSC view, we believe that the user obtains an additional useful visual representation of the checked system.

The implementation is written using 800 lines of SML/NJ code, and in addition exploits the C code of the MSC/POGA [2] system for generating the HCMSC visual structure. The algorithm presented in this paper does not generate an 'optimal' HCMSC. Similar to graph drawing, it is difficult to define precisely optimality for such a representation. In particular, we observed that the quality of the generated representation depends on the ability to find a small set of cutpoints Z, which break the state space cycles (a problem that is known to be in NP-hard, and therefore a good heuristics is called for). We expect that a good strategy will be an interactive approach, where the user will help the system by selecting

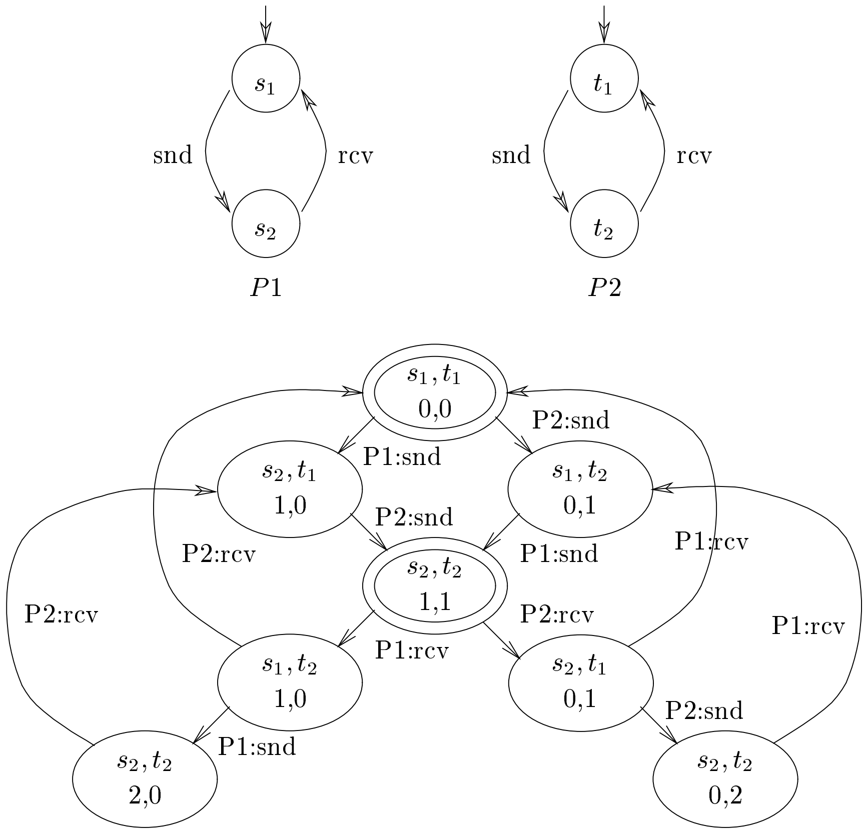


Fig. 8. A simple protocol and its state space

only a part of the checked software for which an HCMSC representation is required.

Acknowledgment. We would like to thank Mihalis Yannakakis, who suggested the counterexample in Figure 5, and to Dietrich Kuske, whose comments helped simplifying the algorithm in Section 6. The detailed comments and improvements suggested by the referees are gratefully acknowledged.

References

1. Alur R, Etessami K, and Yannakakis, M (2000). Inference of message sequence charts. In *22nd International Conference on Software Engineering*, Limerick, Ireland, 2000. ACM, pp 304–313.
2. Alur R, Holzmann G, and Peled D (1996). An analyzer for message sequence charts. In *Software Concepts and Tools*, 17(2):70–77.
3. Alur R, Peled D and Penczek W (1995). Model Checking of Causality Properties. In 10th Annual IEEE Symposium on Logic in Computer Science, LICS'95, San Diego, CA, 1995, pp 90-100.
4. Alur R and Yannakakis Y (1999). Model checking of message sequence charts. In Concurrency Theory, 10th International Conference, CONCUR'99, Eindhoven, The Netherlands, 1999. LNCS no. 1664, pp 114–129.
5. Bolognesi T and Brinksma E (1987). Introduction to the ISO specification language LOTOS. Computer Networks, 14:25–59.
6. Esparza E, Hansel D, Rossmanith P and Schwoon S (2000). Efficient algorithms for model checking pushdown systems. In Computer Aided Verification, 12th International Conference, CAV 2000, Chicago (IL), USA. LNCS no. 1855, 232-247, 2000.
7. Godefroid P (1996) Partial Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem. Springer, LNCS no. 1035.
8. Godefroid P and Wolper P (1994). A partial approach to model checking. In *Information and Computation*, 110(2):305–326.
9. Gunter E and Peled D (1999). Path exploration tool. In Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Amsterdam, The Netherlands. LNCS no. 1579, pp 405–419.
10. Haugen O (1996). Special Issue of Computer Networks and ISDN Systems on SDL and MSC.
11. Hélouët L and Le Maigat P (2000). Decomposition of Message Sequence Charts. In 2nd Workshop on SDL and MSC, SAM 2000, Grenoble, France, pp 46–60.
12. Henriksen J, Mukund M, Narayan Kumar K, and Thiagarajan P (2000). On message sequence graphs and finitely generated regular MSC languages. In Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland. LNCS no. 1853, pp 675-686.

13. ITU-T Recommendation Z.120, Message Sequence Chart (MSC), Geneva, 1996.
14. Lamport L (1978). Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21:558–565.
15. Madhusudan P (2001). Reasoning about sequential and branching behaviours of message sequence graphs. In Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece. LNCS no. 2076, 809–820.
16. Mauw S and Reniers M (1997). High-level message sequence charts. In SDL'97: Time for Testing - SDL, MSC and Trends. Proc. of the SDL Forum'97, 291–306.
17. Muscholl A and Peled D (1999). Message sequence graphs and decision problems on Mazurkiewicz traces. In Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland. LNCS no. 1672, 81–91.
18. Muscholl A, Peled D, and Su Z (1998). Deciding properties of Message Sequence Charts. In Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Lisbon, Portugal. LNCS no. 1378, 226–242.
19. Muscholl A and Peled D (2001). High-level message sequence charts and finite-state communication protocols. In Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece. LNCS no. 2076, pp 720–731.
20. Peled D (1993). All from One, One for All: on Model Checking Using Representatives. In Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece. LNCS no. 697, 409–423.
21. Peled D (2000). Specification and verification of Message Sequence Charts. In Formal Techniques for Distributed System Development, FORTE/PSTV 2000, Pisa, Italy, Kluwer, pp 139-154.
22. Tanenbaum A (1998). Computer Networks, Prentice Hall.