Message Sequence Charts

Blaise Genest[1], Anca Muscholl[1], and Doron Peled[3]

[1] LIAFA, Université Paris VII & CNRS
2, pl. Jussieu, case 7014 75251 Paris cedex 05, France
[2] Department of Computer Science The University of Warwick
Coventry, CV4 7AL United Kingdom

Abstract. *Message sequence charts* (MSC) are a graphical notation for the description of communication scenarios between asynchronous processes.

1 Introduction

Specifying the behavior of software systems in such a way that formal methods can be applied and validation tasks can be automated, is a challenging goal. A lot of research has been devoted to the automated verification of hardware, for which quite efficient tools have been developped. The state-of-the-art for software verification is less successful, and this is due to the presence of complex features such as recursion, data, concurrency, etc. However, the problems we encounter depend heavily on the specific application we have in mind. For instance, one of the main issues for telecommunication applications is how to model the concurrency such that it is comprehensive and can be conveniently used by a designer. Further aspects that influence the choice is how a given model behaves w.r.t. automated verification, and how it combines with other models.

2 Message Sequence Graphs and Communicating Finite-State Machines

We present in this section two specification formalisms for communication protocols, namely *Message Sequence Charts* and *Communicating Finite-State Machines*.

Message Sequence Charts (MSC for short) is a scenario language standardized by the ITU ([**?**]). They represent simple diagrams depicting the activity and communications in a distributed system. The entities participating in the interactions are called instances (or processes). They are represented by vertical lines, on which the behavior of each single process is described by a sequence of events. Message exchanges are depicted by arrows from the sender to the receiver. In addition to messages, atomic events, timers, local/global conditions can also be represented.

Definition 1 *A Message Sequence Chart (MSC for short) is a tuple* $M = \langle \mathcal{P}, E, \mathcal{C}, \ell, m, <, \rangle$ *where:*

– $\mathcal{P}$ *is a finite set of processes,*
– E *is a finite set of events,*
– $\mathcal{C}$ *is a finite set of names for messages and local actions,*
– $\ell : E \to \mathcal{T} = \{p!q(a), p?q(a), p(a) \mid p \neq q \in \mathcal{P}, a \in \mathcal{C}\}$ *labels an event with its* type*: in process* p*, either a send* $p!q(a)$ *of message* a *to process* q*, or a receive* $p?q(a)$ *of message* a *from process* q*, or a local event* $p(a)$*. The labeling* ℓ *partitions the set of events by type (send, receive, or local),* $E = S \dot\bigcup R \dot\bigcup L$*, and by process,* $E = \dot\bigcup_{p \in \mathcal{P}} E_p$*.*
– $m : S \to R$ *is a bijection matching each send to the corresponding receive. If* $m(s) = r$*, then* $\ell(s) = p!q(a)$ *and* $\ell(r) = p?q(a)$ *for some processes* $p, q \in \mathcal{P}$ *and some message name* $a \in \mathcal{C}$*.*
– $< \subseteq E \times E$ *is an acyclic relation between events consisting of:*
  1. *a total order on* E_p*, for every process* $p \in \mathcal{P}$*, and*
  2. $s < r$*, whenever* $m(s) = r$*.*

The event labeling ℓ implicitly defines the process $pr(e)$ for each event e as $pr(e) = p$ if $e \in E_p$ (equivalently, $\ell(e) \in \{p!q(a), p?q(a), p(a)\}$ for some $q \in \mathcal{P}, a \in \mathcal{C}$.) Since point-to-point communication is usually FIFO (first-in-first-out) we make in the following the same assumption for MSCs. That is, we assume that whenever $m(s_1) = r_1$, $m(s_2) = r_2$ holds with $pr(s_1) = pr(s_2)$, $pr(r_1) = pr(r_2)$ and $s_1 < s_2$, then we also have $r_1 < r_2$.

The left part of Figure 1 gives an example of an MSC M modeling messages sent between processes p_1 and p_2.

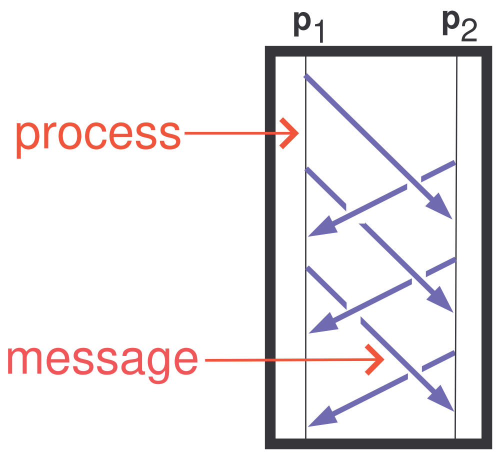

Fig. 1. MSC execution of alternated bit protocol

The relation $<$ is called the *visual* order on the MSC, since it corresponds to its graphical representation. It is comprised of the process ordering and the message ordering, pairwise between send and matching receive. Since $<$

is required to be acyclic, its reflexive-transitive closure $<^*$ is a partial order on the set E of events, which we will denote for simplicity also by $\leq$. Any extension of $\leq$ to a total order on E is called a linearization of M. We denote by $\mathrm{Lin}(M)$ the set of all *labeled linearizations* of an MSC M, $\mathrm{Lin}(M) = \{\ell(e_1)\cdots\ell(e_n) \mid e_1\cdots e_n \text{ is a linearization of } M\}$.

Since the specification of a communication protocol consists of many scenarios, either in positive or in negative form, a high-level description is needed for combining them together and defining infinite sets of (finite or infinite) scenarios. The standard description of the norm Z.120 introduces non-deterministic branching, concatenation and iteration for composing finite MSCs. Formally, a *Message Sequence Graph* (MSG for short) $G = \langle V, R, v^0, v^f, \lambda\rangle$ consists of a finite transition system (V, R, v^0, v^f) with transition set $R \subseteq V \times V$, initial node v^0 and terminal node v^f. Each node v is labeled by the finite MSC $\lambda(v)$. An *execution* of G is the labeling $\lambda(v_0)\lambda(v_1)\cdots\lambda(v_k)$ of some path $v^0 = v_0, v_1, \ldots, v_k = v^f$ of G, i.e., $(v_i, v_{i+1}) \in R$ for every $0 \leq i < k$. The set of executions of G is denoted by $\mathcal{L}(G)$, the set of linearizations of executions of G is denoted by $\mathrm{Lin}(G)$. The *size* of a MSG is the sum of the sizes of its nodes.

Of course, the semantics of MSGs depends on the definition of the MSC product. We consider the usual *weak product* of MSCs, that concatenates MSCs along the process lines. Let $M_1 = \langle E_1, <_1, \mathcal{P}, t_1, \mathcal{C}_1, m_1\rangle$ and $M_2 = \langle E_2, <_2, \mathcal{P}, t_2, \mathcal{C}_2, m_2\rangle$ be MSCs over the same set of processes $\mathcal{P}$. The product M_1M_2 is the MSC $\langle E_1 \biguplus E_2, <, \mathcal{P}, t_1 \cup t_2, \mathcal{C}_1 \cup \mathcal{C}_2, m_1 \cup m_2\rangle$ over the disjoint union of events $E_1 \biguplus E_2$, with the visual order given by:

$$< \,=\, <_1 \cup <_2 \cup \{(e,f) \in E_1 \times E_2 \mid pr(e) = pr(f)\}\,.$$

Note that there is no synchronization between different processes when moving from one node to the next one (*weak product*). Hence, it is possible that one process is still involved in some actions of M_1, while another process has advanced to an event of M_2.

Another standardized specification notation for telecommunication applications is SDL (Specification and Description Language, ITU norm Z.100). SDL is dedicated to the design of real-time, distributed systems and involves complex features as hierarchy, procedure calls and abstract data types. The basic theoretical model behind SDL are nested communicating finite-state machines. We recall the definition of (flat) *communicating finite-state machines* (CFM for short).

A CFM $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathcal{P}}$ consists of finite-state machines $\mathcal{A}_p$ associated with processes $p \in \mathcal{P}$, that communicate over unbounded, error-free, FIFO channels. The content of a channel is a word over a finite alphabet $\mathcal{C}$. With each pair $(p,q) \in \mathcal{P}^2$ of distinct processes we associate a channel $C_{p,q}$. Each finite state machine $\mathcal{A}_p$ is described by a tuple $\mathcal{A}_p = (S_p, A_p, \to_p, F_p)$ consisting of a set of local states S_p, a set of actions A_p, a set of local final states F_p and a transition relation $\to_p \subseteq S_p \times A_p \times S_p$. The computation begins in an initial state $s^0 \in \prod_{p\in\mathcal{P}} S_p$. The actions of $\mathcal{A}_p$ are either local actions or sending/receiving a message. We use the same notations as for MSCs. Sending message $a \in \mathcal{C}$ from process p to process q is denoted by $p!q(a)$ and it means that a is appended to the channel $C_{p,q}$. Receiving message a by p from q is denoted by $p?q(a)$ and it means that a must be the first message in $C_{q,p}$, which will be then removed from $C_{q,p}$. A local action a on process p is denoted by $l_p(a)$. We denote a run of the CFM as *successful*, if each process p finishes the execution in some final state F_p and all channels are empty. The set of successful runs of $\mathcal{A}$ is denoted $\mathcal{L}(\mathcal{A})$. The *size* of $\mathcal{A}$ is $\sum_p |\mathcal{A}_p|$.

Note that each successful run of a CFM defines an MSC. Conversely, with each MSC $M = \langle \mathcal{P}, E, \mathcal{C}, \ell, m, <, \rangle$ we can associate an equivalent CFM, by defining the behavior of process p as the (ordered) sequence of events E_p. However, the two formalisms MSG and CFM are incomparable in general, as discussed in the section below. Comparing the expressivity of these two models is interesting for at least two reasons. First, both formalisms are heavily used in protocol design, sometimes for specifying different parts of a system at different stages of the design process. Second, MSCs are usually intended as early requirements, for a rough description of the desired/undesired behavior. Thus, the question whether the described behavior can be turned into a protocol – *implementability/realizability problem*– is an important validation step in the design process.

2.1 Comparing MSG and CFM

Two main parameters are involved in a comparison between MSG and CFM, namely *control* and *channels*.

Control in a CFM is inherently local, since it corresponds to local transition functions. The control structure of an MSG is global, since the branching from a node concerns all processes occurring in the future execution. The global control mechanism of an MSG is actually a consequence of the visual character of the diagram graph, that imposes a sequential control mechanism. One problem arising from the global control is that an MSG G might be non-implementable, i.e., no CFM $\mathcal{A}$ exists with $\mathcal{L}(\mathcal{A}) = \mathcal{L}(G)$. For a simple example, consider the MSG G consisting of a single node v with a self-loop, labeled by a message from p_1 to p_2 and another message from p_3 to p_4. Since the MSCs in $\mathcal{L}(G)$ must contain equally many messages from p_1 to p_2 and from p_3 to p_4, there can be no equivalent CFM.

We turn now to the second parameter, namely channels. Although none of the models impose any bound on the channel capacity, validation tasks such as model-checking tend to be "more decidable" for MSGs than for CFMs. The reason is that MSGs have *existentially-bounded* channels, i.e., for each MSG G there exists an integer b such that every MSC in $\mathcal{L}(G)$ can be executed with channels of size at most b. Moreover, the bound b is linear in the maximal size of the MSCs labeling the nodes of G. A set X of MSCs is called *existentially-bounded* if there exists some b such that every MSC $M \in X$ has some linearization $w \in \mathrm{Lin}(M)$ satisfying the following property: for every pair of distinct processes p, q and every prefix v of w, it holds that $0 \leq \sum_{a \in \mathcal{C}} |v|_{p!q(a)} - \sum_{a \in \mathcal{C}} |v|_{q?p(a)} \leq b$. As an example of property that is undecidable for CFM (but not for MSG) one can consider the question whether a CFM generates at least one MSC, [8]. A less trivial example is *pattern-matching*: given an MSC M and an MSG G, we ask whether there is some execution $N \in \mathcal{L}(G)$ and a factorization $N = N_1 M N_2$, where N_1, N_2 are both MSCs. The pattern-matching algorithm described in [**?**] (see also [10]) uses heavily the fact that MSGs are existentially-bounded (with an priori known bound).

In a more general vein, CFMs cannot be transformed into MSGs since MSGs are *finitely generated*. That is, for any MSG G there exists a finite set X of finite MSCs such that any execution $M \in \mathcal{L}(G)$ can be written as a (finite or infinite) product $M = M_1 M_2 \cdots M_k$ of factors from X, $M_i \in X$ for all i. A typical example of CFM that is not finitely generated is a scenario from the well-known *alternating bit protocol* (imagine MSC in Figure 1 with crossing messages repeated). For this infinite MSC M there is no decomposition $M = M_1 M_2$ with M_1, M_2 non-empty MSCs, since including a send in M_1 forces to add another send on the other process (the one preceding the corresponding receive). More generally, an MSC M is called *atomic* (or *atom*), if for any decomposition $M = M_1 M_2$ where both M_1, M_2 are MSCs, at most one is non-empty. The set of atoms generating the MSC executions of an MSG G is denoted $\mathrm{At}(G)$. It is a finite set and it represents a canonic set of generators of $\mathcal{L}(G)$. Moreover, it can computed by a simple linear-time algorithm, see [13].

Returning to the alternating bit example, it is easily seen that the set of linearizations $\mathrm{Lin}(M)$ of the represented MSC M, is regular. Note that in this particular example *every* linearization of M has channel bound at most 3. We call a set X of MSCs *universally-bounded* if for every MSC $M \in X$, *every* linearization $w \in \mathrm{Lin}(M)$, every prefix v of w and every pair of distinct processes p, q we have $0 \leq \sum_{a \in \mathcal{C}} |v|_{p!q(a)} - \sum_{a \in \mathcal{C}} |v|_{q?p(a)} \leq b$. Notice also that such a universal channel bound for an MSG G does not suffice for $\mathrm{Lin}(G)$ being a regular set. Hence, even if the specification is given as finite state automaton $\mathcal{A}$, we cannot automatically transform $\mathcal{A}$ into an equivalent MSG G. This led to an extension of the MSG formalism, namely to *Compositional Message Sequence Graphs* (CMSG, for short), [12]. A compositional MSC (CMSC, for short) is defined as an MSC, except that the message function m is partially defined. A send that does not belong to the domain of the message function m, or a receive not belonging to the range of m, are called unmatched events. The product of two CMSC $M_1 M_2$ is defined as for MSC, but in addition the k-th unmatched send of M_1 is matched with the k-th unmatched receive of M_2 (if they exist) in such a way that the FIFO property is satisfied by matched events. Hence, the product of CMSCs is only partially defined. Moreover, it is not associative, hence we define a product $M_1 M_2 \cdots$ as parenthesized from left to right.

It is not very difficult to see that any CFM $\mathcal{A} = (\mathcal{A}_p)_{p \in \mathcal{P}}$ can be transformed into an equivalent CMSG G of exponential size. The nodes correspond to the global states $(s_p)_{p \in \mathcal{P}}$ ($s_p \in S_p$) of $\mathcal{A}$. From each global state $s = (s_p)_{p \in \mathcal{P}}$ there is a transition for each action $e \in A_p$ for which $\delta_p(s_p, e) \neq \emptyset$ for some process p, leading to the node $s' = (s'_p)_p$, where $s'_q = s_q$ for all $q \neq p$, and $s'_p \in \delta_p(s_p, e)$. It is easy to check that any CMSC execution of $\mathcal{A}$ with no unmatched receive is an execution of G, and vice-versa. For instance, the CFM generating our alternated bit protocol example can be transformed into the CMSG on figure 2.

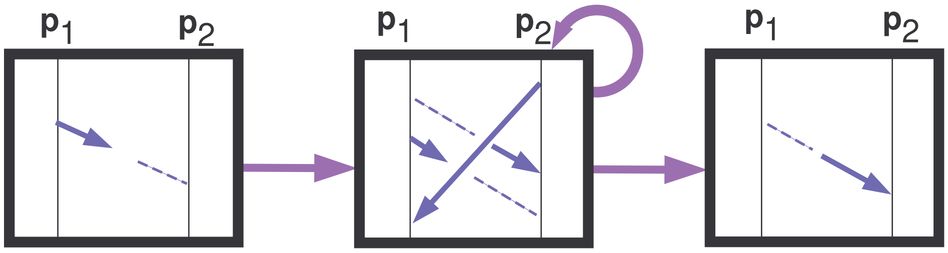

Fig. 2. CMSG depicting execution of alternated bit protocol

Theorem 1. *Any CFM can be transformed into an equivalent CMSG of exponential size.*

We end this section by some definitions concerning atoms. On the potentially infinite alphabet At of atomic MSCs, we can define an independence (commutation) relation $I \subseteq \mathrm{At} \times \mathrm{At}$ by letting $A\,I\,A'$ iff $pr(A) \cap pr(A') = \emptyset$. Notice that $A\,I\,A'$ implies that A, A' commute, $AA' = A'A$, and that the decomposition of any MSC into atoms is unique up to commuting adjacent atoms A, A', with $A\,I\,A'$. For a regular set $X \subseteq \mathrm{At}^*$ we denote as I-closure of X the set consisting of all strings that can be obtained from strings in X by repeatedly commuting adjacent symbols a, b with $a\,I\,b$.

3 Validating MSC Specifications: Model-Checking and Implementation

Since MSG specifications are used very early in the design process, revealing errors is of primary importance. This has motivated the design of algorithms that check specific properties such as *race conditions* [3, 19] and detecting *non-local choice* [6, 13]. *Model-checking* MSG specifications has been considered w.r.t. properties expressed as MSG [18], automata [5] and partial-order logics [20, 17]. Another test that may reveal the incompleteness of an MSG specification is the one for implementability. Here, we want to know whether the specification can be transformed into a state-based, distributed model. As discussed in section 3.2, the definition of implementability is not canonical, and the results strongly depend on the variant we consider.

3.1 Model-Checking

In the common model-checking approach (see for recent textbooks [9, 7]) we usually describe bad execution sequences using the same formalism as for specifying the system (e.g., finite automata over infinite words). Then we need to check the emptiness of the intersection between the bad sequences and the system, and counter-examples can be obtained if the intersection is non-empty. In the MSC setting we cannot use complementation as with finite automata, so we consider two variants of model-checking, *positive* and *negative* model-checking. In both cases we specify the property P we want to check, as well as the system S itself, by MSGs. For *negative* model-checking we view P as a set of bad MSC executions and we ask whether $\mathcal{L}(P) \cap \mathcal{L}(S) = \emptyset$. For *positive* model-checking we view P as a set of good MSC executions and we ask whether $\mathcal{L}(S) \subseteq \mathcal{L}(P)$.

In the general setting of MSG specifications, both model-checking variants are undecidable, [5, 18]. This holds even if the property P is given as a finite-state automaton, [5]:

Theorem 1 *Given a finite-state automaton P and an MSG graph G, it is undecidable whether $\mathcal{L}(P) \cap \mathcal{L}(G) = \emptyset$.*

The proof for theorem 1 is a straightforward reduction from Post's correspondence problem (PCP). Recall that an instance of PCP consists of pairs of words $(x_i, y_i)_{1 \le i \le k}$ over the alphabet $\{0, 1\}$. Then we ask for a non-empty sequence of indices $i_1, \ldots, i_n$ such that $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$.

The MSG G consists of $(k+2)$ nodes $v^0, v_1, \ldots, v_k, v^f$. Node v^0 (v^f, resp.) is initial (final, resp.), and labeled by the empty MSC. Node v_i is labeled by a sequence of messages from p_1 to p_2 labeled 0 or 1 such that the sequence of labels equals x_i, and a message from p_3 to p_4 labeled by i. There is a transition from v^0 to each of v_i, from each v_i to v^f, and one from v^f to v_0. The automaton P accepts precisely the set $(X_1 + \cdots + X_k)^+$, where each X_i is a finite word defined as follows: Let $y_i = a_1 \cdots a_m$, then $X_i = p_1!p_2(a_1)p_2?p_1(a_1) \cdots p_1!p_2(a_m)p_2?p_1(a_m)\, p_3!p_4(i)p_4?p_3(i)$.

Clearly, since there is no synchronization between the process pairs $\{p_1, p_2\}$ and $\{p_3, p_4\}$, both the MSG and the automaton describe MSCs with two parallel branches, one over the PCP words (x for G, y for P) and the other over the corresponding indices. The non-empty intersection between G and P reveals then a PCP solution.

Remark. Note that the undecidability proof above does not rely on the unboundedness of channels, since G is existentially-bounded. Actually the construction can be slightly modified such that G becomes universally-bounded. The true reason for undecidability is the concurrency, since G and P use different linearizations of the same partial orders (MSCs). $\square$

Several decidable variants of model-checking have been considered in subsequent papers. Some of them are obtained by restricting the properties we want to check, others are obtained by restricting the system specification. However, all variants are base on a similar idea. Suppose for instance that the property P is given by a *linearization-closed* finite-state automaton $\mathcal{A}$. That is, for every word $w \in \mathcal{L}(\mathcal{A})$, the automaton $\mathcal{A}$ also accepts every linearization $v \in \mathrm{Lin}(M)$ of the MSC M defined by w. In this case it suffices to consider *representative* *linearizations* of the system MSG G: We choose for every node v of G some linearization of the MSC labeling v, say l_v. Then we define a finite-state automaton $\mathcal{A}(G)$ from G by replacing the label of v by l_v. Thus, states of $\mathcal{A}(G)$ are labeled by words. It is easy to see now that $\mathcal{L}(G) \cap \mathcal{L}(\mathcal{A}) \neq \emptyset$ if and only if $\mathcal{L}(\mathcal{A}(G)) \cap \mathcal{L}(\mathcal{A}) \neq \emptyset$, and $\mathcal{L}(G) \subseteq \mathcal{L}(\mathcal{A})$ if and only if $\mathcal{L}(\mathcal{A}(G)) \subseteq \mathcal{L}(\mathcal{A})$.

Among the model-checking variants that led to algorithmic solutions we refer to the following ones:

- *Model-checking with gaps* [19]: The property P is given by an MSG, but its semantics differs from the semantics of the system G. An execution M of P is matched *with gaps* by an execution M' of G if there is an embedding ϕ of the events of M in the set of events of M' such that the visual order is preserved: whenever $e < f$ in M, we have $\phi(e) <' \phi(f)$ in M'. This problem has been shown to be NP-complete (even if P is an acyclic MSG).
  The reason for decidability of model-checking with gaps is twofold. First, gaps lead to very restricted languages for which we can compute a sort of linearization-closure. Second, we can stay in the realm of regular languages exploiting the fact that the system MSG has *existentially-bounded* channels.
- *Using partial-order specifications* [20, 17]: Here, the property P is given by a partial-order logic, which makes it linearization-closed. In [20] a logic derived from a fragment of TLC [4] is proposed for MSGs. Basically, this logic corresponds to CTL interpreted over partial-order graphs of MSCs, where the edge relation is the immediate successor relation (on each process, resp. for send/receive pairs). It is shown in [20] how to construct an exponential-size automaton from the specification, hence model-checking is PSPACE w.r.t. the specification (and only linear in the size of the system). In [17] the specification formalism is MSO, interpreted over partial-order graphs of MSCs. Here, the complexity is non-elementary, as it is already in the word case.

3.2 Implementability

As previously mentioned, the MSG formalism is useful as a specification notation, but it does not provide directly a protocol model. Such a model is usually state-based and distributed, whereas MSGs provide an implicit global control over the behavior of the processes. This allows for specifications that are not implementable because of global choices.

As protocol model we use a CFM over the same set of processes as the MSG specification. But we still have some choice for the semantics of the implementation. For instance, we could allow for an implementation with more (or less) behavior than the MSG. The most natural notion is that the implementation is equivalent to the MSG: An MSG G is *implementable*, if some CFM $\mathcal{A}$ exists such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(G)$. Furthermore, we allow the implementation to contain additional data in messages. That is, the message contents of the CFM come from a finite set $\mathcal{C}' = \mathcal{C} \times D$, where $\mathcal{C}$ is the set of message contents of the MSG and D is some finite set. Then, the equality $\mathcal{L}(\mathcal{A}) = \mathcal{L}(G)$ is required up to the additional data D. A further, even more relaxed notion of implementability, would also allow for additional messages. Notice that this would make every MSG implementable, since the additional messages can be used for synchronizing all processes after each node. We do not allow additional messages, since in many applications they are neither desired nor possible (e.g., satellite communication or applications where acknowledgements cannot be provided).

The first notion of implementation, which we denote as *standard implementation*, has been proposed in [1, 2]. The standard implementation of the MSG $G = \langle V, R, v^0, v^f, \lambda\rangle$ over the process set $\mathcal{P}$ does not add any data and it is fully determined by the MSG, being defined process by process: The automaton $\mathcal{A}_p$ for process p generates the projection of $\mathcal{L}(G)$ on the events of process p.

We call an MSG *standard-implementable* if it is implementable w.r.t. the standard version of implementability. Notice that this notion is actually too weak, since it captures just a small subset of implementable specifications. The simplest example is a set of two MSCs over the processes p_1, p_2 where the first MSC M_1 has a message from p_1 to p_2, followed by one from p_2 to p_1. In the second MSC M_2 we have first a message from p_2 to p_1, then one from p_1 to p_2. These two MSCs are not standard-implementable since we can combine the projection of M_1 on p_1 with the projection of M_2 on p_2 and we obtain an MSC (two crossing messages) that does not belong to $\{M_1, M_2\}$, see figure 3. Changing slightly this example we obtain one which is not standard implementable, but is implementable with additional data. For this, we just add at the beginning of both M_1, M_2 a first message (with no label) from p_1 to p_2, see figure 4. Then the non-implementability argument given previously still works. However, with additional data we use the initial message for letting p_1 decide on the outcome M_1 or M_2, and inform p_2.

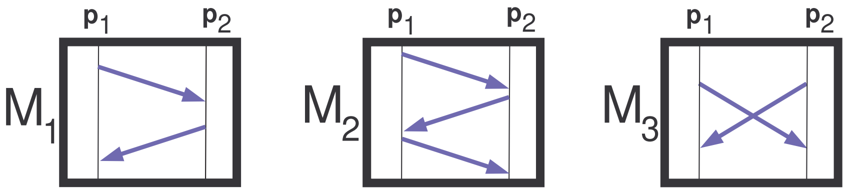

Fig. 3. The set of MSC $\{M_1, M_2, M_3\}$ is not implementable

Two striking weaknesses of the standard notion is that not even regular MSGs are standard-implementable, as seen from the example above, and that whether a regular MSG is standard-implementable or not, is undecidable (see section 4 for more details).

The results of [1, 2] show however that although the standard implementability is undecidable, it becomes decidable at least for regular MSGs if one disallows *deadlocks* (called *safe realizability* in [1, 2]), albeit with high algorithmic complexity. A CFM is called a *deadlock-free implementation* of an MSG G if

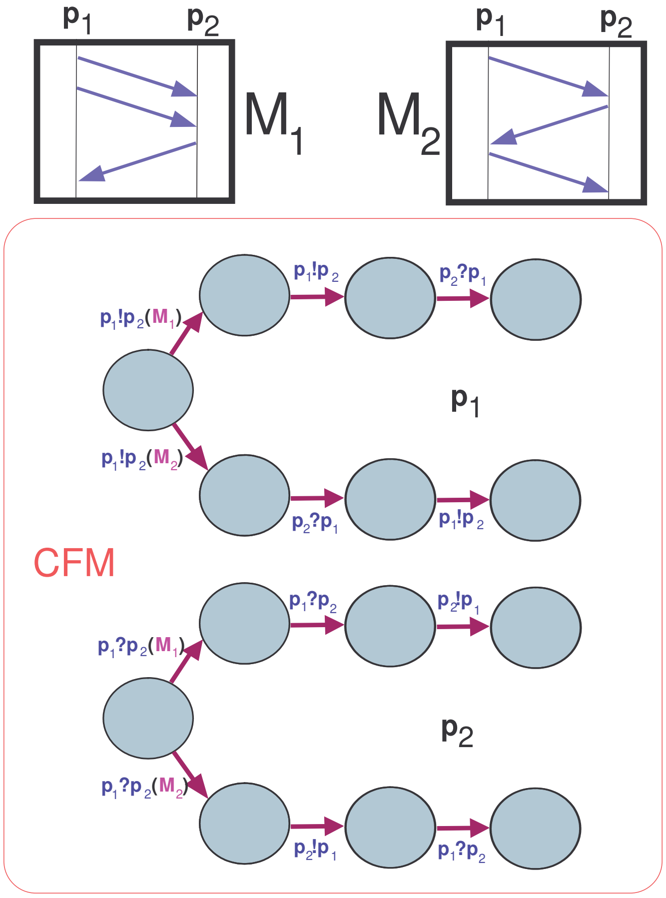

Fig. 4. CFM Implementating the set of MSC $\{M_1, M_2\}$ with additional data

$\mathcal{L}(\mathcal{A}) = \mathcal{L}(G)$ and every configuration of $\mathcal{A}$ that has no successor, is such that all processes have reached a final state and all channels are empty. Deadlock freeness is of course required in practice, since real-life protocols should not be aborted in some unclean state. We will recall the various results on the implementability problem in sections 4 and 5.

4 Regular MSC Specifications

Regular MSGs have been proposed in the context of model-checking, as a subclass for which both variants of model-checking are decidable, [5, 18]. It is a syntactic restriction that ensures that the set of all linearizations, i.e., the set $\mathrm{Lin}(G)$, is regular. Regular MSGs provided to be a robust class, in terms of logical and automata-theoretic characterizations. In particular, regular MSGs can be implemented with additional data by CFM with universally-bounded channels. However,the CFM implementation is not deadlock-free, in general.

A set X of finite MSCs is called *regular* [14] if $\mathrm{Lin}(X)$ is a regular string language over the alphabet $\mathcal{T}$ of event types. The following syntactic condition ensures that an MSG G generates a regular set $\mathcal{L}(G)$ of MSCs. We first need to define the *communication graph* of an MSC M: It is a directed graph over the set of communicating processes in M with an edge from process p to process q whenever M contains a message from p to q. An MSG G is called a *regular MSG* (locally-synchronized in [18], bounded in [5]) if any MSC labeling a loop of G has a strongly connected communication graph.

For instance, figure 5 represents an acces protocol throught a firewall by a regular MSG.

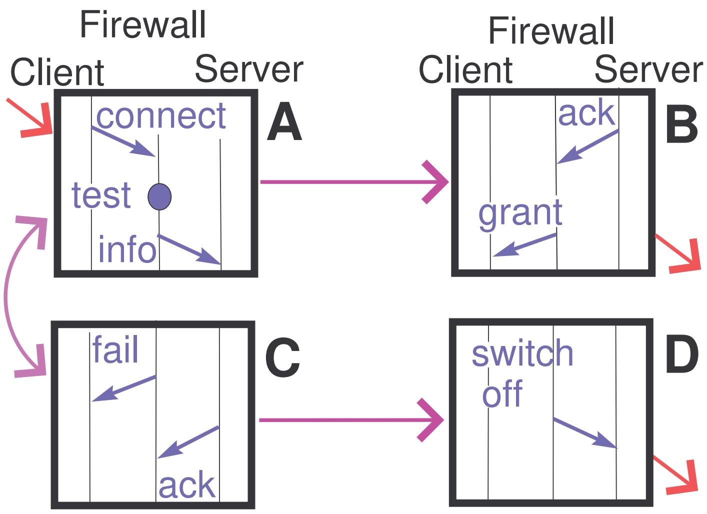

Fig. 5. protocol represented by a regular MSG

Putting together the results from [5, 18, 14] we have the following relationship between regular sets of MSCs and regular MSGs:

Theorem 2
For every regular MSG G the set $\mathcal{L}(G)$ of generated MSCs is regular, [5, 18]. For every regular and finitely generated set X of MSCs there exists a regular MSG G with $X = \mathcal{L}(G)$, [14].

The main interest in regular MSGs was to obtain a subclass of MSC specifications with a decidable model-checking problem. However, it turned out that model-checking is rather expensive, which is actually not very surprising when we deal with concurrent models:

Theorem 3 *[5, 18] The negative model-checking problem $\mathcal{L}(G) \cap \mathcal{L}(H) \neq \emptyset$ where G is a regular MSG, is PSPACE-complete. The positive model-checking problem $\mathcal{L}(G) \subseteq \mathcal{L}(H)$ where H is a regular MSG, is EXPSPACE-complete.*

The upper bounds in the theorem above are based on the fact that if G is a regular MSG then we can compute a finite automaton of exponential size generating $\mathrm{Lin}(G)$.

Regular MSC languages also have nice characterizations in the logical and communicating automata framework. The logic used in [15, 17, **?**] is MSO with atomic propositions $\ell(e) = t \in \mathcal{T}$, $e \leq f$ and $e \in E$ that have the usual interpretation, as type labeling, partial order of the MSC and membership in a second order variable E. The theorem below has been extended to infinite MSCs in [**?**]:

Theorem 4 *[15] Let X be a universally-bounded set of MSCs. The following assertions are equivalent:*

1. *X can be implemented by a (deterministic) CFM with additional data.*
2. *There exists an MSO formula ϕ such that X is the set of bounded MSCs satisfying ϕ.*

In the first part of the theorem above the implementation is not deadlock-free, since the constructed CFM uses global final states for accepting X. On the other hand, as we mentioned in section **??**, the standard implementation is not really helpful when applied to regular MSGs (the upper bound is due to [2], and the lower bound to [16]):

Theorem 2. *[2, 16] It is undecidable to know whether a regular MSG is standard-implementable. It is EXPSPACE-complete to know whether a regular MSG is standard-implementable without deadlocks.*

5 Globally-Cooperative MSGs

As seen in section 4, model-checking for regular MSGs is decidable and of tractable complexity (PSPACE). However, the situation is far from being ideal.

Notice first that some trivial protocols cannot be represented by regular MSGs. For instance, the protocol where process p_1 can send any number of messages to process p_2. The reason is that regular MSGs have universally bounded channels, which restricts severely their expressive power. Second, for real life communication protocols one can usually find a (sufficiently large) bound b so that any run of the protocol can be executed with channels each bounded by b. Similarly to MSGs, we call such protocols *existentially b-bounded*. Whereas an algorithm exists to check whether a CFM of a finite state machine is existentially b-bounded for a given b, its complexity depends severely (exponentially) on b. Hence, in practice we cannot hope to be able to fix a sufficiently large bound b that takes care of all executions. A further problem is that we cannot obtain in general an automaton generating all linearizations of executions for models that are strictly more expressive than regular MSGs. Instead, we can try to use representative linearizations rather than all linearizations, requiring that the set of representative linearizations is b-bounded, with b as small as possible.

The representative linearizations that we use are the ones that linearize atoms one by one. More precisely, for any $M \in \mathcal{L}(G)$ we consider only linearizations in $\mathrm{Lin}(M)$ of the form $w = w_1 \cdots w_n$, where $M = A_1 \cdots A_n$ is a decomposition of M into atoms A_i and $w_i \in \mathrm{Lin}(A_i)$ for all i. Let us denote by $\mathrm{Lin}^a(G) \subseteq \mathrm{Lin}(G)$ the set of such linearizations of MSCs of $\mathcal{L}(G)$. For an example, let G be the graph consisting of a single node with a self-loop, labeled by a message from p_1 to p_2. Let $s = p_1!p_2$ and $r = p_2?p_1$, then $\mathrm{Lin}^a(G) = (sr)^*$. Of course, $\mathrm{Lin}(G)$ is not regular, it corresponds to the Dyck language over one pair of brackets.

We can use representative linearizations for model-checking as follows. Let G, H be two MSGs. Then it is easy to see that $\mathcal{L}(G) \cap \mathcal{L}(H) = \emptyset$ if and only if $\mathrm{Lin}^a(G) \cap \mathrm{Lin}^a(H) = \emptyset$ (respectively, $\mathcal{L}(G) \subseteq \mathcal{L}(H)$ if and only if $\mathrm{Lin}^a(G) \subseteq \mathrm{Lin}^a(H)$. Recall that for any MSG G, atoms of $\mathcal{L}(G)$ are finite and finitely many. Hence the set of representative linearizations $\mathrm{Lin}^a(G)$ is b-bounded, where b is such that G is existentially b-bounded. For getting a regular set of representative linearizations $\mathrm{Lin}^a(G)$ we impose a theoretically well-known restriction, that of *loop-connectedness*. An MSG G is called *globally-cooperative* (*gc-MSG* for short) if every loop of G is labeled by a connected MSC.

We can change slightly the graph of a gc-MSG G by replacing each node v labeled by some non-empty MSC M by a path of new nodes $v_1, \ldots, v_k$ with v_i is labeled by A_i and $M = A_1 \cdots A_k$ is some decomposition of M into atoms A_i. The new graph G' can be seen as an automaton with states labeled over the alphabet of atoms $\mathrm{At}(G)$. The property of G being a gc-MSG translates to G' being loop-connected, i.e., every loop of G' is labeled by a subalphabet of $\mathrm{At} = \mathrm{At}(G)$ that is connected w.r.t. $(\mathrm{At} \times \mathrm{At}) \setminus I$. With this restriction it is well-known that the I-closure of the regular set generated by G' is regular, and an automaton generating the I-closure can be effectively computed, [**?**,18]. From this automaton we obtain $\mathrm{Lin}^a(G)$ and an automaton generating it simply by replacing every atom $A \in \mathrm{At}(G)$ by some linearization of A. Since the size of the automaton generating $\mathrm{Lin}^a(G)$ is exponential in the size of G we obtain:

Theorem 5 *[11] Given a gc-MSG G and an arbitrary MSG H, it is PSPACE-complete to decide whether $\mathcal{L}(G) \cap \mathcal{L}(H) = \emptyset$. The positive model-checking problem $\mathcal{L}(G) \subseteq \mathcal{L}(H)$ where H is a gc-MSG, is EXPSPACE-complete.*

Notice that the complexity of model-checking gc-MSGs is not higher than for regular MSGs. Moreover, the situation for gc-MSGS is better, since we do not have to compute all linearizations, but a smaller subset that has the additional property of being b-bounded for a small b, yielding an algorithm that is faster in practice than the one given for regular MSGs. A regular MSG G is universally B-bounded with a B that can be exponential in the size of G.

We turn now to the implementation problem. The situation here enforces the idea that that universal channel bounds are not needed. For the safe variant of the standard implementability problem, i.e., where the implementation is not allowed any additional data but must be deadlock free, the complexity is the same for regular MSGs and for gc-MSGs:

Theorem 6 *[?,16] Given a gc-MSG G, it is EXPSPACE-complete to decide whether there exists a deadlock-free CFM $\mathcal{A}$ with $\mathcal{L}(G) = \mathcal{L}(\mathcal{A})$.*

Again, this result is not really practical, given the high complexity. Moreover, in practice it might be the case that the standard implementation does not work for some gc-MSG G, but that G is still implementable with a little more data. For instance...

In the case where one allows data to be added to messages but deadlocks are not allowed, there are gc-MSGs that cannot be implemented. For instance, consider the gc-MSG G with two nodes with self-loops, and two edges between them. Both nodes are labeled by MSCs with two messages, one from p_1 to p_2 and one from p_3 to p_2. The first node has its messages carrying the data a, while the second node carries the data b. Both nodes are initial and final. In any CFM implementation processes p_1 and p_3 should decide to send either both a or both b, but this is impossible with no additional syncronization (messages). Hence, this protocol cannot be implemented without deadlocks. It remains open whether every gc-MSG can be implemented with additional data and allowing deadlocks. The conjecture in [11] is that this is always possible.

Concerning an algorithm for testing whether an MSG is implementable with additional data, the complexity is really hard to figure out without an estimation on the amount of data to be added. And this amount could be pretty high.

6 Choice and Implementability

Deadlock-free implementability being a key feature required for communicating protocols, tractable algorithms that help implementing an MSG with additional data are needed. One reasonable way of doing this is first to exhibit a non-trivial subclass of MSGs that is always implementable with additional data and no deadlocks. Then we want to test whether an MSG can be represented inside our subclass, preserving the MSC language.

As mentioned before, the reason for non-implementability of an MSG is the global control, whereas the choice in a CFM must be done locally. The idea is then to define MSGs that have only local choices, that is any node is controlled by a single process, [6, 13].

Definition 1 *An MSG* $G = \langle V, R, v^0, v^f, \lambda\rangle$ *is called* local-choice *(*lc-MSG *for short) if each MSC labeling any node* v *of* G *is a* triangle, *that is it has a single minimal event* $\min(v)$ *(in the partial order* $\leq$*). Moreover,* $\min(w)$ *belongs to* $pr(v)$ *whenever* $(v, w) \in R$*.*

Figure 6 shows an lc-MSG. Not that it is equivalent to the non lc-MSG in figure 5.

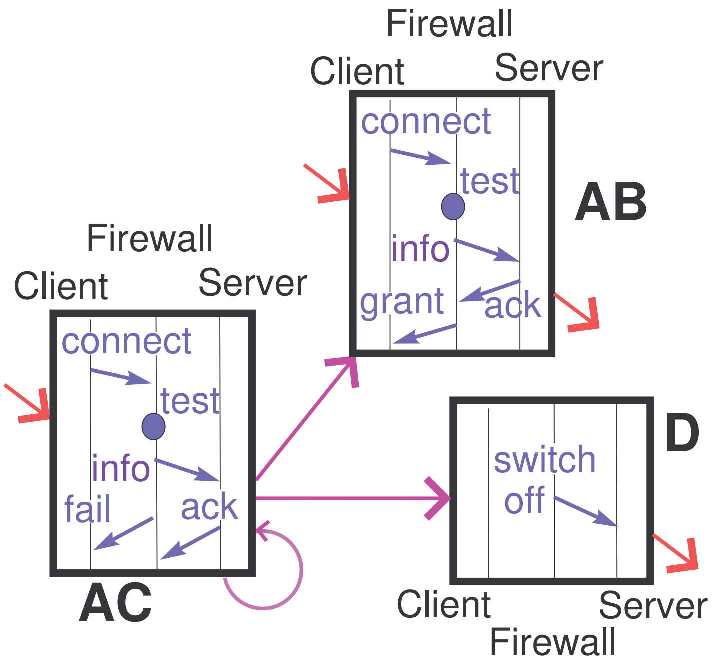

Fig. 6. protocol represented by local-choice MSG

It is not very hard to translate a lc-MSG into a deadlock-free CFM, with linear additional data:

The idea for implementating an lc-MSG is to use a *leader process* and to let the current leader choose the current_node to be executed and the next_leader. The node is chosen among the nodes that follow the node being executed, and that begin with a minimal event belonging to the leader. The next_leader should be chosen among the minimal processes of nodes that follow the chosen node.

In the procedure polling_state below process p waits for a message informing it about the next node to execute and the next leader:

```
void polling_state()
{ while (true) {
     if p receives a message (a,v,q) then
            { current_node=v; next_leader=q; return;} } }
```

Initially, the current node is initialized by letting next_leader $= pr(\min((v^0))$. Before executing its event from current_node, process p goes to a polling state, unless it is the leader process. Here is the algorithm for process p:

```
initialization();
 while (true)
 { if (p ≠ next_leader) polling_state();
   else {current_node=guess(current_node);
          next_leader=guessp(current_node);}
   execute_path(current_node); }
```

The algorithm execute_path(current_node) above makes that process p executes its events from current_node, if any. In this case each message sent by p contains the additional data (current_node, next_leader).

Theorem 7 *[11] Every lc-MSG is implementable by a deadlock-free CFM with linear additional data.*

Since each node is a triangle, every process but the minimal process begins by a receive. A process that is chosen to be the leader is always informed since it is involved in the node where it is chosen.

It is important to see that if additional data is forbidden, then there are lc-MSG that are not implementable, even when allowing deadlocks. Consider for example an lc-MSG with 3 nodes $1, 2, 3$. The initial node consists of a message from process p_1 to p_2. Then either node 2 is executed, with process p_1 sending to process p_3, or node 3 is executed with process p_2 sending to process p_4 (see figure 7. Since processes p_1 and p_2 do not know which one will be next (no additional data, same past), both can begin, thus both nodes 2 and 3 can start. The execution must stop, and there is no distributed way to know whether the protocol went fine or not. Hence without additional data this protocol is not implementable at all.

While local-choice is defined syntactically, we show that it corresponds actually to a semantic property. Moreover, one can test whether an MSG is transformable into a lc-MSG. Triangles are of huge importance here. We first define a generic lc-MSG over triangles of size bounded by n. The MSG H_n has, for each triangle T of size at most n, one node v_T labeled by T. There is an edge $v_T \to v_{T'}$ if $pr(\min(T')) \in pr(T')$.

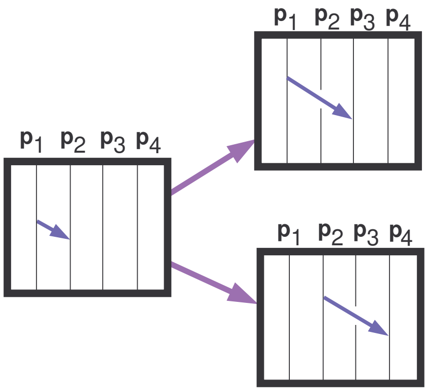

Fig. 7. Local-choice MSGs are not always implementable without additional data

Proposition 1 *An MSG G is equivalent to some lc-MSG iff there exists some n such that $\mathcal{L}(G) \subseteq \mathcal{L}(H_n)$. If this is the case, then we can obtain a lc-MSG equivalent to G, exponential in G and n.*

To obtain a lc-MSG from G, we build the synchronized product between G and H_n.

While it is PSPACE to test whether $\mathcal{L}(G) \subseteq \mathcal{L}(H_n)$ theorem 9, the number n is not bounded so far. For testing, we need to bound n. We use the following three structural properies of lc-MSG G.

First, any $M \in \mathcal{L}(G)$ is a triangle.

Secondly, there is a bound b s.t. for every MSC in $\mathcal{L}(G)$ of the form $U(R//S)V$ with R, S MSCs, either $|R| < b$ or $|S| < b$. Actually, one can take $b = 2\wp|G|$. It implies that for an MSG to be equivalent to some lc-MSG, it is necessary to be gc-MSG.

Third, there is a bound b s.t. for every MSC in $\mathcal{L}(G)$ of the form URV with a MSC $|V| > b$, there exists a triangle T suffix of RV, such that $min(T) \in R$.

Obviously, an MSG G equivalent to some lc-MSG follows these 3 properties. More important, the converse holds. It allows us to state

Theorem 8 *Testing whether an MSG G is equivalent to some lc-MSG is PSPACE. Moreover, if true, one can compute a double exponential lc-MSG equivalent to G.*

Proof. We can check whether G is a gc-MSG in co-NP.

Checking the first property is polynomial Time.

Checking the second property for gc-MSG is co-NP. If true, the algorithm provides a bound b polynomial in G.

Checking the third property for gc-MSG is PSPACE. If true, the algorithm provides a bound b exponential in G.

We can then compute an lc-MSG with the product $H_b \times G$. As b is exponential in G and H exponential in b, the result is at most doubly exponential in G.

□

One important question is whether this class is expressive enough, else the test to know whether a MSG is equivalent to some lc-MSG would almost certainly lead to a negative answer. Comparing lc-MSGs to regular MSGs, lc-MSGs tend to be more useful in practice. The restriction of universally-bounded channels of regular MSGs holds no more for lc-MSGs, allowing us to express a wider class of protocols. Moreover, we can implement without deadlock while it is not the case with regular MSGs. The real drawback of lc-MSG is the fact that too long MSCs in parrallel are not allowed, while it is the case in regular MSG, albeit not in the same loop of the graph. Actually, it would not be difficult to cut the protocol into parallel ones, and implement each one using lc-MSGs.

Since lc-MSG is a subclass of gc-MSG, the former can be strictly easier to model check than the latter. In order to improve the results, triangles can be used as generators instead of atoms. For a given lc MSG, each node v labeled by T is sliced into two nodes labeled by triangles R, S, as long as $T = RS$ with $pr(\min(w)) \in S$ for every $v \to w$. Notice that by definition of triangle, $pr(\min(S)) \in R$. Let $T_1 \cdots T_n, T'_1 \cdots T'_{n'}$ be sequences of triangles labeling 2 paths ρ, ρ' of lc-MSGs G, H sliced in this way. Then there exists k, X s.t. $T_i = T'_i$ for $i < k$, $T'_k = XT_{k+1} \cdots T_n$ and $T_k = XT'_{k+1} \cdots T'_{n'}$. Hence, $T_{k+1} \cdots T_n$ should be of size smaller than the biggest node. The same applies for $T'_{k+1} \cdots T'_{n'}$. We do not have commutation problems anymore, and we can model check using an algorithm very similar to the one for words automata.

Theorem 9 *[11] Given 2 lc-MSG G, H, it is quadratic time to decide whether $\mathcal{L}(G) \cap \mathcal{L}(H) = \emptyset$. The positive model-checking problem $\mathcal{L}(G) \subseteq \mathcal{L}(H)$ where H is a lc-MSG and G an arbitrary MSG, is PSPACE-complete.*

References

1. R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proceedings of the 22nd International Conference on Software Engineering, Limerick (Ireland)*, pages 304–313. ACM, 2000.
2. R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In F. e. a. Orejas, editor, *Proceedings of the 28th International colloquium on Automata, Languages and Programming (ICALP'01), Crete (Greece) 2001*, number 2076 in Lecture Notes in Computer Science, pages 797–808. Springer, 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, D. Peled, and W. Penczek. Model-checking of causality properties. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS '95)*, pages 90–100. IEEE, 1995.
5. R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proceedings of the 10th International Conference on Concurrency Theory CONCUR'99, Eindhoven (The Netherlands)*, number 1664 in Lecture Notes in Computer Science, pages 114–129. Springer, 1999.

6. H. Ben-Abdallah and S. Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In E. Brinksma, editor, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, Third International Workshop, TACAS'97*, number 1217 in Lecture Notes in Computer Science, pages 259–274, Enschede, The Netherlands, 1997. Springer.
7. B. Berard, M. Bidoit, A. Finkel, A. Laroussinie, F.and Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.
8. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.
9. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
10. B. Genest and A. Muscholl. Pattern matching and membership for hierarchical message sequence charts. In S. Rajsbaum, editor, *Proceedings of the LATIN 2002: Theoretical Informatics, 5th Latin American Symposium, Cancun, Mexico*, number 2286 in Lecture Notes in Computer Science, pages 326–340. Springer, 2002.
11. B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level mscs: Model-checking and realizability. In P. Widmayer, editor, *Proceedings of the 29thInternational colloquium on Automata, Languages and Programming (ICALP'02), Malaga (Spain), 2002*, number 2380 in Lecture Notes in Computer Science, pages 657–668. Springer, 2002.
12. E. Gunter, A. Muscholl, and D. Peled. Compositional message sequence charts. In T. Margaria, editor, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, 7th International Workshop (TACAS'01)*, number 2031 in Lecture Notes in Computer Science, pages 496–511. Springer, 2001. Journal version to appear in the International Journal on Software Tools for Technology Transfer.
13. L. Hélouët and P. Le Maigat. Decomposition of Message Sequence Charts. In S. Graf, C. Jard, and Y. Lahav, editors, *Proceedings of the 2nd Workshop on SDL and MSC (SAM2000), Col de Porte, Grenoble*, pages 46–60, 2000.
14. J. G. Henriksen, M. Mukund, K. Narayan Kumar, and P. Thiagarajan. On message sequence graphs and finitely generated regular msc languages. In U. Montanari, J. Rolim, and E. Welzl, editors, *Proceedings of the 27th International colloquium on Automata, Languages and Programming (ICALP'00), Geneva (Switzerland), 2000*, number 1853 in Lecture Notes in Computer Science, pages 675–686. Springer, 2000.
15. J. G. Henriksen, M. Mukund, K. Narayan Kumar, and P. Thiagarajan. Regular collections of message sequence charts. In M. e. a. Nielson, editor, *Proceedings of the 25th Symposium on Mathematical Foundations of Computer Science (MFCS'00), Bratislava (Slovakia), 2000*, number 1893 in Lecture Notes in Computer Science, pages 405–414. Springer, 2000.
16. M. Lohrey. Safe realizability of high-level message sequence charts. In L. e. a. Brim, editor, *Proceedings of the Concurrency Theory, 13th International Conference (CONCUR'02)*, number 2421 in Lecture Notes in Computer Science, pages 177–192. Springer, 2002.
17. P. Madhusudan. Reasoning about sequential and branching behaviours of message sequence graphs. In O. F., P. Spirakis, and J. van Leeuwen, editors, *Proceedings of the 28th International colloquium on Automata, Languages and Programming (ICALP'01), Crete (Greece) 2001*, number 2076 in Lecture Notes in Computer Science, pages 809–820. Springer, 2001.
18. A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz traces. In M. Kutylowski and L. Pacholski, editors, *Proceedings of the*

24thSymposium on Mathematical Foundations of Computer Science (MFCS'99), Szklarska Poreba (Poland) 1999, number 1672 in Lecture Notes in Computer Science, pages 81–91. Springer, 1999.

19. A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In M. Nivat, editor, *Proceedings of the 1st International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'98), Lisbon, Portugal, 1998*, number 1378 in Lecture Notes in Computer Science, pages 226–242. Springer, 1998.
20. D. Peled. Specification and verification of message sequence charts. In *Proceedings of Formal Techniques for Distributed System Development, FORTE/PSTV 2000, Pisa, Italy*, pages 139–154, 2000.