Realizability of Dynamic MSC Languages*

Benedikt Bollig[1] and Loïc Hélouët[2]

[1] LSV, ENS Cachan, CNRS, INRIA, France
[2] IRISA, INRIA, Rennes, France

Abstract. We introduce dynamic communicating automata (DCA), an extension of communicating finite-state machines that allows for dynamic creation of processes. Their behavior can be described as sets of message sequence charts (MSCs). We consider the realizability problem for DCA: given a dynamic MSC grammar (a high-level MSC specification), is there a DCA defining the same set of MSCs? We show that this problem is decidable in doubly exponential time, and identify a class of realizable grammars that can be implemented by *finite* DCA.

1 Introduction

Requirements engineering with scenario-based visual languages such as message sequence charts (MSCs) is a well established practice in industry. However, the requirements phase usually stops when a sufficiently large *finite* base of scenarios covering expected situations of the modeled system has been created. Although more elaborated formalisms have been proposed, such as HMSCs [13], compositional MSCs [8], or causal MSCs [6], requirements frequently consist in a finite set of finite behaviors over a finite set of processes. The existing higher-level constructs are often neglected. A possible reason might be that, in view of their huge expressive power, MSC specifications are not always implementable. As a part of the effort spent in the requirements design is lost when designers start implementing a system, scenarios remain confined to expressions of finite examples, and the higher-level constructs are rarely used. Another reason that may prevent designers from using high-level scenarios is that most models depict the interactions of an a priori *fixed* set of processes. Nowadays, many applications rely on threads, and most protocols are designed for an open world, where all the participating actors are not known in advance. A first step towards MSCs over an evolving set of processes was made by Leucker, Madhusudan, and Mukhopadhyay [11]. Their *fork-and-join MSC grammars* allow for dynamic creation of processes and have good properties, such as decidability of MSO model checking. However, it remains unclear how to implement fork-and-join MSC grammars. In particular, a corresponding automata model with a clear behavioral semantics based on MSCs is missing. Dynamic process creation and its realizability are then two important issues that must be considered jointly.

This paper introduces dynamic communicating automata (DCA) as a model of programs with process creation. In a DCA, there are three types of actions: (1) a new process can be created, (2) a message can be sent to an already existing process, and

* Partly supported by the projects ANR-06-SETI-003 DOTS and ARCUS Île de France-Inde.

(3) a message can be received from an existing process. Processes are identified by means of process variables, whose values can change dynamically during an execution of an automaton and be updated when a message is received. A message is sent through bidirectional unbounded FIFO channels, which are available to any pair of existing processes. Our model extends classical communicating finite-state machines [5], which allow only for actions of the form (2) and (3) and serve as an implementation model for existing specification languages such as HMSCs or compositional MSCs.

In a second step, we propose dynamic MSC grammars (DMG for short) as a specification language. They are inspired by the fork-and-join grammars from [11] but closer to an implementation. We keep the main idea of [11]: when unfolding a grammar, MSCs are concatenated on the basis of finitely many process identifiers. While, in [11], the location of identifiers can be changed by means of a very general and powerful split-operator, our grammars consider an identifier as a pebble, which can be moved *locally* within one single MSC. In addition to process identifiers, we introduce a new means of process identification that allows for a more concise description of some protocols.

Given an implementation model and a specification formalism, the *realizability* problem consists in asking whether a given specification comes with a corresponding implementation. Realizability for MSC languages has been extensively studied in the setting of a fixed number of processes [2, 12, 1]. In a dynamic framework where DMGs are seen as specifications and DCA as distributed implementations, we have to consider a new aspect of realizability, which we call *proximity realizability*. This notion requires that two processes know each other at the time of (asynchronous) communication. We show that proximity realizability can be checked in doubly exponential time. Note that the representation of the behavior of each process may require infinitely many states (due to the nature of languages generated by the grammar), and that the notion of proximity realizability does not take into account the structure of processes. The next step is then to identify a class of DMGs that is realizable in terms of *finite* DCA.

The paper is organized as follows: Section 2 introduces MSCs. Sections 3 and 4 present dynamic communicating automata and dynamic MSC grammars, respectively. In Section 5, we define proximity realizability and show that the corresponding decision problem can be solved in doubly exponential time. Moreover, we present an implementation of local-choice MSC grammars in terms of finite DCA. Section 6 concludes with some directions for future work. Missing proofs can be found in [3].

2 Message Sequence Charts

A message sequence chart (MSC) consists of a number of processes (or threads). Each process p is represented by a totally ordered set of events E_p. The total order is given by a direct successor relation $<_{\mathsf{p}}$. An event is labeled by its type. The minimal element of each thread is labeled with start. Subsequent events can then execute send (!), receive (?), or spawn (spawn) actions. The relation $<_{\mathsf{m}}$ associates each send event with a corresponding receive event on a different thread. The exchange of messages between two threads has to conform with a FIFO policy. Similarly, $<_{\mathsf{s}}$ relates a spawn event $e \in E_p$ with the (unique) start action of a different thread $q \neq p$, meaning that p has created q.

Definition 1. *An* MSC *is a tuple* $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, <_{\mathsf{p}}, <_{\mathsf{s}}, <_{\mathsf{m}}, \lambda)$ *where*

(a) $\mathcal{P} \subseteq \mathbb{N} = \{0, 1, \ldots\}$ *is a nonempty finite set of* processes,
(b) the E_p *are disjoint nonempty finite sets of* events *(we let* $E := \bigcup_{p \in \mathcal{P}} E_p$*),*
(c) $\lambda : E \to \{!, ?, \mathsf{spawn}, \mathsf{start}\}$ *assigns a* type *to each event, and*
(d) $<_\mathsf{p}$*,* $<_\mathsf{s}$*, and* $<_\mathsf{m}$ *are binary relations on* E*.*

There are further requirements: $\leq := (<_\mathsf{p} \cup <_\mathsf{s} \cup <_\mathsf{m})^*$ *is a partial order;* $\lambda^{-1}(\mathsf{start}) = \{e \in E \mid$ *there is no* $e' \in E$ *such that* $e' <_\mathsf{p} e\}$*;* $<_\mathsf{p} \subseteq \bigcup_{p \in \mathcal{P}}(E_p \times E_p)$ *and, for every* $p \in \mathcal{P}$*,* $<_\mathsf{p} \cap (E_p \times E_p)$ *is the direct-successor relation of some total order on* E_p*;* $(E, \leq)$ *has a unique minimal element, denoted by* $start(M)$*;* $<_\mathsf{s}$ *induces a bijection between* $\lambda^{-1}(\mathsf{spawn})$ *and* $\lambda^{-1}(\mathsf{start}) \setminus \{start(M)\}$*;* $<_\mathsf{m}$ *induces a bijection between* $\lambda^{-1}(!)$ *and* $\lambda^{-1}(?)$ *satisfying the following: for* $e_1, e_2 \in E_p$ *and* $e_1', e_2' \in E_q$ *with* $e_1 <_\mathsf{m} e_1'$ *and* $e_2 <_\mathsf{m} e_2'$*, we have both* $p \neq q$ *and* $e_1 \leq e_2$ *iff* $e_1' \leq e_2'$ *(FIFO).*

In Figure 1, M is an MSC with set of processes $\mathcal{P} = \{1, 2, 3, 4\}$. An MSC can be seen as one single execution of a distributed system. To generate infinite collections of MSCs, specification formalisms usually provide a concatenation operator. It will allow us to append to an MSC a partial MSC, which is a kind of suffix that does not necessarily have start events on each process. Let $M = (\mathcal{P}, (E_p)_{p \in \mathcal{P}}, <_\mathsf{p}, <_\mathsf{s}, <_\mathsf{m}, \lambda)$ be an MSC and let $E' \subseteq E$ be a nonempty set satisfying $E' = \{e \in E \mid (e, e') \in <_\mathsf{m} \cup <_\mathsf{s} \cup \leq^{-1}$ for some $e' \in E'\}$ (i.e., E' is an upward-closed set containing only *complete* messages and spawning pairs). Then, the restriction of M to E' is called a *partial MSC* (PMSC). In particular, the new process set is $\{p \in \mathcal{P} \mid E' \cap E_p \neq \emptyset\}$. The set of PMSCs is denoted by $\mathbb{P}$, the set of MSCs by $\mathbb{M}$. Consider Figure 1. It depicts the simple MSC I_p, with one event on process $p \in \mathbb{N}$. Moreover, $M_1, M_2 \in \mathbb{P} \setminus \mathbb{M}$.

Let $M = (\mathcal{P}, (E_p)_{p \in \mathcal{P}}, <_\mathsf{p}, <_\mathsf{s}, <_\mathsf{m}, \lambda)$ be a PMSC. For $e \in E$, we denote by $loc(e)$ the unique process $p \in \mathcal{P}$ such that $e \in E_p$. For every $p \in \mathcal{P}$, there are a unique minimal and a unique maximal event in $(E_p, \leq \cap (E_p \times E_p))$, which we denote by $\min_p(M)$ and $\max_p(M)$, respectively. We let $Proc(M) = \mathcal{P}$. By $Free(M)$, we denote the set of processes $p \in \mathcal{P}$ such that $\lambda^{-1}(\mathsf{start}) \cap E_p = \emptyset$. Finally, $Bound(M) = \mathcal{P} \setminus Free(M)$. Intuitively, free processes of a PMSC M are processes that are not initiated in M. In Figure 1, $Bound(\mathsf{I}_p) = \{p\}$, $Free(M_1) = \{1\}$, and $Free(M_2) = \{1, 2\}$.

Visually, concatenation of PMSCs corresponds to drawing identical processes one below the other. For $i = 1, 2$, let $M^i = (\mathcal{P}^i, (E^i_p)_{p \in \mathcal{P}^i}, <^i_\mathsf{p}, <^i_\mathsf{s}, <^i_\mathsf{m}, \lambda^i)$ be PMSCs. Consider the structure $M = (\mathcal{P}, (E_p)_{p \in \mathcal{P}}, <_\mathsf{p}, <_\mathsf{s}, <_\mathsf{m}, \lambda)$ where $E_p = E^1_p \uplus E^2_p$ for all $p \in \mathcal{P} = \mathcal{P}^1 \cup \mathcal{P}^2$ (assuming $E^i_p = \emptyset$ if $p \notin \mathcal{P}^i$) and $<_\mathsf{p} = <^1_\mathsf{p} \cup <^2_\mathsf{p} \cup \{(\max_p(M^1), \min_p(M^2)) \mid p \in \mathcal{P}$ with $E^1_p \neq \emptyset$ and $E^2_p \neq \emptyset\}$. In addition, $<_\mathsf{m}$ and λ arise as simple unions. If M is a PMSC, then we set $M^1 \circ M^2 := M$. Otherwise, $M^1 \circ M^2$ is undefined (e.g., if some $p \in \mathcal{P}^2$ has a start event and $E^1_p \neq \emptyset$).

In the context of partial orders, it is natural to consider linearizations. We fix the infinite alphabet $\Sigma = \{!(p, q), ?(p, q), \mathsf{spawn}(p, q) \mid p, q \in \mathbb{N}$ with $p \neq q\}$. For a PMSC $M = (\mathcal{P}, (E_p)_{p \in \mathcal{P}}, <_\mathsf{p}, <_\mathsf{s}, <_\mathsf{m}, \lambda)$, we let $poset(M) := (E', \leq', \lambda')$ where $E' = E \setminus \lambda^{-1}(\mathsf{start})$, $\leq' = \leq \cap (E' \times E')$, and $\lambda' : E' \to \Sigma$ such that, for all $(e, \hat{e}) \in <_\mathsf{s}$, we have $\lambda'(e) = \mathsf{spawn}(loc(e), loc(\hat{e}))$, and, for all $(e, \hat{e}) \in <_\mathsf{m}$, both $\lambda'(e) = !(loc(e), loc(\hat{e}))$ and $\lambda'(\hat{e}) = ?(loc(e), loc(\hat{e}))$. The set $Lin(poset(M))$ of *linearizations* of $poset(M)$ is defined as usual as a subset of Σ^*.

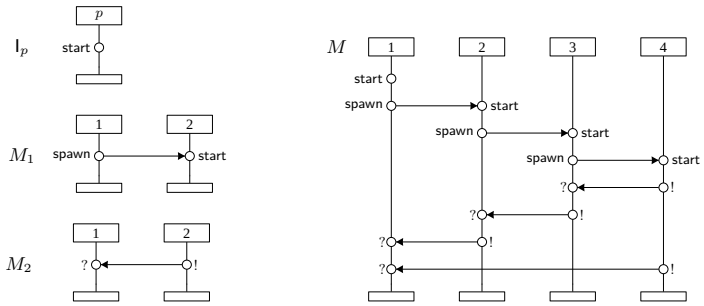

Fig. 1. (Partial) message sequence charts

3 Dynamic Communicating Automata

Dynamic communicating automata (DCA) extend classical communicating finite-state machines [5]. They allow for the dynamic creation of processes, and *asynchronous* FIFO communication between them. Note that most of existing dynamic models lack such asynchronous communication (see [4] for some references). Each process p holds a set of process variables. Their values represent process identities that p remembers at a given time, and they allow p to communicate with them. This model is close to the threading mechanism in programming languages such as JAVA and Erlang, but also borrows elements of the routing mechanisms in protocols implemented over partially connected mesh topologies. Threads will be represented by dynamically created copies of the same automaton. At creation time, the creating thread will pass known process identities to the created thread. A thread can communicate with another one if both threads know each other, i.e., they have kept their identities in memory. This mechanism is chosen to preserve the partial-order concurrency of MSCs, which provide the semantics of DCA.

We introduce DCA with an example. The DCA in Figure 2 comes with sets of process variables $X = \{x_1, x_2, x_3\}$, messages $Msg = \{m\}$, states $Q = \{s_0, \ldots, s_6\}$ where s_0 is the initial state, final states $F = \{s_3, s_4, s_6\}$, and transitions, which are labeled with actions. Each process associates with every variable in X the identity of an existing process. At the beginning, there is one process, say 1. Moreover, all process variables have value 1, i.e., $(x_1, x_2, x_3) = (1, 1, 1)$. When process 1 moves from s_0 to s_1, it executes $x_1 \leftarrow \mathsf{spawn}(s_0, (\mathsf{self}, \mathsf{self}, x_3))$, which creates a new process, say 2, starting in s_0. In the creating process, we obtain $x_1 = 2$. In process 2, on the other hand, we initially have $(x_1, x_2, x_3) = (1, 1, 1)$. So far, this scenario is captured by the first three events in the MSC M of Figure 1. Process 2 itself might now spawn a new process 3, which, in turn, can create a process 4 in which we initially have $(x_1, x_2, x_3) = (3, 3, 1)$. Now assume that, instead of spawning a new process, 4 moves to state s_5 so that it sends the message $(m, (4, 3, 1))$ to process 3. Recall that process 3 is in state s_1 with $(x_1, x_2, x_3) = (4, 2, 1)$. Thus, 3 can execute $x_1\,?\,(m, \{x_1\})$, i.e., receive $(m, (4, 3, 1))$ and set x_1 to 4. We then have $(x_1, x_2, x_3) = (4, 2, 1)$ on process 3. The DCA accepts, e.g., the behavior M depicted in Figure 1.

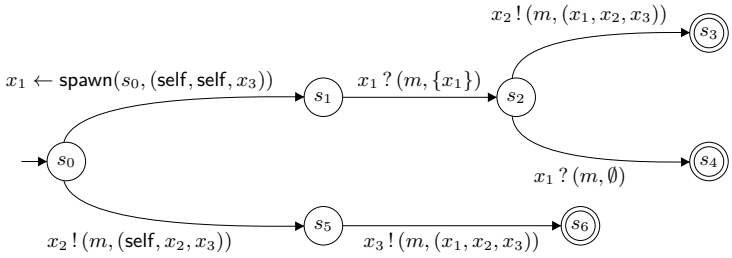

Fig. 2. A dynamic communicating automaton

Definition 2. *A* dynamic communicating automaton *(or simply DCA) is a tuple* $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ *where* X *is a set of* process variables*,* Msg *is a set of* messages*,* Q *is a set of* states*,* $\iota \in Q$ *is the* initial state*,* $F \subseteq Q$ *is the set of* final states*, and* $\Delta \subseteq Q \times Act_{\mathcal{A}} \times Q$ *is the set of* transitions*. Here,* $Act_{\mathcal{A}}$ *is a set of* actions *of the form* $x \leftarrow \mathsf{spawn}(s, \eta)$ *(*spawn action*),* $x\,!\,(m, \eta)$ *(*send action*),* $x\,?\,(m, Y)$ *(*receive action*), and* $\mathsf{rn}(\sigma)$ *(*variable renaming*) where* $x \in X$*,* $s \in Q$*,* $\eta : (X \uplus \{\mathsf{self}\})^X$*,* $\sigma : X \to X$*,* $Y \subseteq X$*, and* $m \in Msg$*. We say that* $\mathcal{A}$ *is* finite *if* X*,* Msg*, and* Q *are finite.*

We define the semantics of a DCA as a word language over Σ. This language is the set of linearizations of some set of MSCs and therefore yields a natural semantics in terms of MSCs. Let $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ be some DCA.

A *configuration* of $\mathcal{A}$ is a quadruple $(\mathcal{P}, state, proc, ch)$ where $\mathcal{P} \subseteq \mathbb{N}$ is a non-empty finite set of active processes (or identities), $state : \mathcal{P} \to Q$ maps each active process to its current state, $proc : \mathcal{P} \to \mathcal{P}^X$ contains the identities that are known to some process, and $ch : (\mathcal{P} \times \mathcal{P}) \to (Msg \times \mathcal{P}^X)^*$ keeps track of the channels contents. The configurations of $\mathcal{A}$ are collected in $Conf_{\mathcal{A}}$. We define a global transition relation $\Longrightarrow_{\mathcal{A}} \subseteq Conf_{\mathcal{A}} \times (\Sigma \cup \{\varepsilon\}) \times Conf_{\mathcal{A}}$ as follows: For $a \in \Sigma \cup \{\varepsilon\}$, $c = (\mathcal{P}, state, proc, ch) \in Conf_{\mathcal{A}}$, and $c' = (\mathcal{P}', state', proc', ch') \in Conf_{\mathcal{A}}$, we let $(c, a, c') \in \Longrightarrow_{\mathcal{A}}$ if there are $p \in \mathcal{P}$ and $\hat{p} \in \mathbb{N}$ with $p \neq \hat{p}$ (the process executing a and the communication partner or spawned process), $x \in X$, $s_0 \in Q$, $\eta : (X \uplus \{\mathsf{self}\})^X$, $Y \subseteq X$, $\sigma : X \to X$, and $(s, b, s') \in \Delta$ such that $state(p) = s$, and one of the cases in Figure 3 holds (c and c' coincide for all values that are not specified below a line).

An *initial configuration* is of the form $(\{p\}, p \mapsto \iota, proc, (p,p) \mapsto \varepsilon) \in Conf_{\mathcal{A}}$ for some $p \in \mathbb{N}$ where $proc(p)[x] = p$ for all $x \in X$. A configuration $(\mathcal{P}, state, proc, ch)$ is *final* if $state(p) \in F$ for all $p \in \mathcal{P}$, and $ch(p,q) = \varepsilon$ for all $(p,q) \in \mathcal{P} \times \mathcal{P}$. A *run* of DCA $\mathcal{A}$ on a word $w \in \Sigma^*$ is an alternating sequence $c_0, a_1, c_1, \ldots, a_n, c_n$ of configurations $c_i \in Conf_{\mathcal{A}}$ and letters $a_i \in \Sigma \cup \{\varepsilon\}$ such that $w = a_1.a_2 \ldots a_n$, c_0 is an initial configuration and, for every $i \in \{1, \ldots, n\}$, $(c_{i-1}, a_i, c_i) \in \Longrightarrow_{\mathcal{A}}$.[3] The run is *accepting* if c_n is a final configuration. The *word language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of words $w \in \Sigma^*$ such that there is an accepting run of $\mathcal{A}$ on w. Finally, the *(MSC) language* of $\mathcal{A}$ is $L(\mathcal{A}) := \{M \in \mathbb{M} \mid Lin(poset(M)) \subseteq \mathcal{L}(\mathcal{A})\}$. Figure 2 shows a *finite* DCA. It accepts the MSCs that look like M in Figure 1.

[3] Here and elsewhere, $u.w$ denotes the concatenation of words u and v. In particular, $a.\varepsilon = a$.

$$\textsf{spawn}\ \frac{a = \textsf{spawn}(p,\hat{p}) \qquad b = x \leftarrow \textsf{spawn}(s_0,\eta)}{\begin{array}{ll} \mathcal{P}' = \mathcal{P} \uplus \{\hat{p}\} & ch'(q,q') = \varepsilon \\ state'(p) = s' & \text{if } \hat{p} \in \{q,q'\} \\ state'(\hat{p}) = s_0 & \end{array} \qquad \begin{array}{l} proc'(p)[x] = \hat{p} \\ proc'(\hat{p})[y] = \begin{cases} proc(p)[\eta[y]] & \text{if } \eta[y] \neq \textsf{self} \\ p & \text{if } \eta[y] = \textsf{self} \end{cases} \\ \qquad \text{for all } y \in X \end{array}}$$

$$\textsf{send}\ \frac{a = !(p,\hat{p}) \qquad b = x\,!\,(m,\eta) \qquad \hat{p} = proc(p)[x]}{state'(p) = s' \qquad ch'(p,\hat{p}) = (m,\gamma).ch(p,\hat{p})}$$
$$\text{where } \gamma \in \mathcal{P}^X \text{ with}$$
$$\gamma[y] = \begin{cases} proc(p)[\eta[y]] & \text{if } \eta[y] \neq \textsf{self} \\ p & \text{if } \eta[y] = \textsf{self} \end{cases}$$

$$\textsf{receive}\ \frac{a = ?(\hat{p},p) \qquad b = x\,?\,(m,Y) \qquad \hat{p} = proc(p)[x]}{state'(p) = s' \qquad \begin{array}{c} \text{there is } \gamma \in \mathcal{P}^X \text{ such that} \\ \left[\begin{array}{c} ch(\hat{p},p) = ch'(\hat{p},p).(m,\gamma) \\ \wedge\ \text{for all } y \in Y, proc'(p)[y] = \gamma[y] \end{array}\right] \end{array}}$$

$$\textsf{renaming}\ \frac{a = \varepsilon \qquad b = \textsf{rn}(\sigma)}{state'(p) = s' \qquad \begin{array}{c} proc'(p)[y] = proc(p)[\sigma(y)] \\ \text{for all } y \in X \end{array}}$$

Fig. 3. Global transition relation of a DCA

DCA actually generalize the classical setting of communicating finite-state machines [5]. To simulate them, the starting process spawns the required number of processes and broadcasts their identities to any other process.

4 Dynamic MSC Grammars

In this section, we introduce *dynamic MSC grammars* (DMGs). They are inspired by the grammars from [11], but take into account that we want to implement them in terms of DCA. We keep the main idea of [11] and use process identifiers to tag active processes in a given context. Their concrete usage is different, though, and allows us to define protocols such as the language of the DCA from Figure 2 in a more compact way.

Let us start with an example. Figure 4 depicts a DMG with non-terminals $\mathcal{N} = \{S, A, B\}$, start symbol S, process identifiers $\Pi = \{\pi_1, \pi_2\}$, and five rules. Any rule has a left-hand side (a non-terminal), and a right-hand side (a sequence of non-terminals and PMSCs). In a derivation, the left-hand side can be replaced with the right-hand side. This replacement, however, depends on a more subtle structure of a rule. The bottom left one, for example, is actually of the form $A \longrightarrow_f \alpha$ with $\alpha = \mathcal{M}_1.A.\mathcal{M}_2$, where f is a function that maps the first process of α, which is considered *free*, to the process identifier π_2. This indicates where α has to be inserted when replacing A in a configuration. To illustrate this, consider a derivation as depicted in Figure 5, which is a sequence of configurations, each consisting of an upper and a lower part. The upper

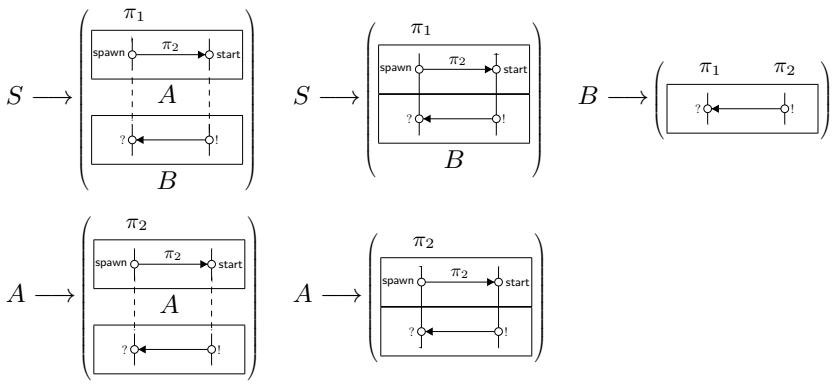

Fig. 4. A dynamic MSC grammar

part is a *named* MSC [11], an MSC where some processes are tagged with process identifiers. The lower part, a sequence of PMSCs and non-terminals, is subject to further evaluation. In the second configuration, which is of the form $(\mathfrak{M}, A.\beta)$ (with named MSC $\mathfrak{M}$), replacing A with α requires a renaming σ of processes: the first process of α, tagged with π_2, takes the identity of the second process of $\mathfrak{M}$, which also carries π_2. The other process of α is considered newly created and obtains a fresh identity. Thereafter, A can be replaced with $\alpha\sigma$ so that we obtain a configuration of the form $(\mathfrak{M}, \mathcal{M}.\gamma)$, $\mathcal{M}$ being a PMSC. The next configuration is $(\mathfrak{M} \circ \mathcal{M}, \gamma)$ where the concatenation $\mathfrak{M} \circ \mathcal{M}$ is simply performed on the basis of process names and does not include any further renaming. Process identifiers might migrate, though. Actually, $\mathcal{M}$ is a pair (M, μ) where M is a PMSC and μ partially maps process identifiers π to process pairs (p, q), allowing π to change its location from p to q during concatenation (cf. the third configuration in Figure 5, where π_2 has moved from the second to the third process).

Let us formalize the components of a DMG. Let Π be a nonempty and finite set of *process identifiers*. A *named MSC* over Π is a pair (M, ν) where M is an MSC and $\nu : \Pi \to \mathit{Proc}(M)$. An *in-out PMSC* over Π is a pair (M, μ) where M is a PMSC and $\mu : \Pi \rightharpoonup \mathit{Free}(M) \times \mathit{Proc}(M)$ is a partial mapping. We denote by $\mathrm{n}\mathbb{M}$ the set of named MSCs and by $\mathrm{m}\mathbb{P}$ the set of in-out PMSCs over Π (we assume that Π is clear from the context). We let $\mathfrak{M}$ range over named MSCs and $\mathcal{M}$ over in-out PMSCs.

A derivation of a DMG is a sequence of configurations $(\mathfrak{M}, \beta)$. The named MSC $\mathfrak{M}$ represents the scenario that has been executed so far, and β is a sequence of non-terminals and in-out PMSCs that will be evaluated later, proceeding from left to right. If $\beta = \mathcal{M}.\gamma$ for some in-out PMSC $\mathcal{M}$, then the next configuration is $(\mathfrak{M} \circ \mathcal{M}, \gamma)$. However, the concatenation $\mathfrak{M} \circ \mathcal{M}$ is defined only if $\mathfrak{M}$ and $\mathcal{M}$ are compatible. Formally, we define a partial operation $\_ \circ \_ : \mathrm{n}\mathbb{M} \times \mathrm{m}\mathbb{P} \rightharpoonup \mathrm{n}\mathbb{M}$ as follows: Let $(M_1, \nu_1) \in \mathrm{n}\mathbb{M}$ and $(M_2, \mu_2) \in \mathrm{m}\mathbb{P}$. Then, $(M_1, \nu_1) \circ (M_2, \mu_2)$ is defined if $M_1 \circ M_2$ is defined and contained in $\mathbb{M}$, and, for all $\pi \in \Pi$ such that $\mu_2(\pi) = (p, q)$ is defined, we have $\nu_1(\pi) = p$. If defined, we set $(M_1, \nu_1) \circ (M_2, \mu_2) := (M, \nu)$ where $M = M_1 \circ M_2$, $\nu(\pi) = \nu_1(\pi)$ if $\mu_2(\pi)$ is undefined, and $\nu(\pi) = q$ if $\mu_2(\pi) = (p, q)$ is defined.

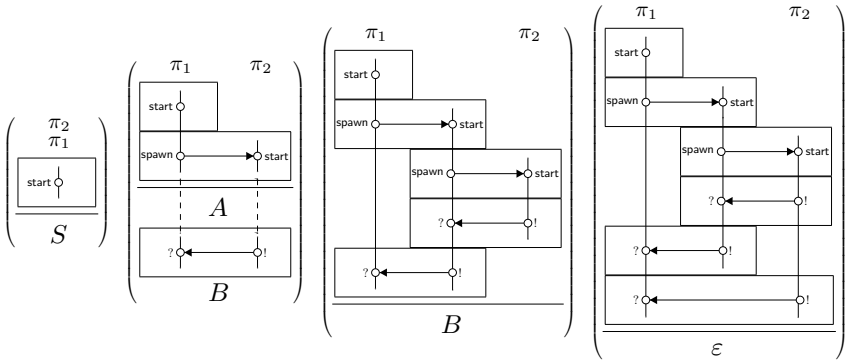

Fig. 5. A derivation

Consider a configuration $(\mathfrak{M}, A.\gamma)$. Replacing non-terminal A with a sequence α includes a renaming of processes to make sure that those that are *free* in α and carry identifier π have the same name as an existing process of $\mathfrak{M}$ carrying π. I.e., processes that occur free in α take identities of processes from $\mathfrak{M}$. To be able to distinguish between free and bound processes in α, we introduce the notion of an expression. Let $\mathcal{N}$ be a set of non-terminals, and Π be a set of process identifiers. An *expression* over $\mathcal{N}$ and Π is a sequence $\alpha \in (\mathrm{m}\mathbb{P} \cup \mathcal{N})^*$ of the form $u_0.(M_1, \mu_1).u_1 \ldots (M_k, \mu_k).u_k$, $k \geq 1$ and $u_i \in \mathcal{N}^*$, such that $M(\alpha) := M_1 \circ \ldots \circ M_k \in \mathbb{P}$. We let $\mathit{Proc}(\alpha) := \mathit{Proc}(M(\alpha))$, $\mathit{Free}(\alpha) := \mathit{Free}(M(\alpha))$, and $\mathit{Bound}(\alpha) := \mathit{Bound}(M(\alpha))$.

Definition 3. *A* dynamic MSC grammar *(DMG) is a quadruple $G = (\Pi, \mathcal{N}, S, \longrightarrow)$ where Π and $\mathcal{N}$ and are nonempty finite sets of* process identifiers *and* non-terminals*, $S \in \mathcal{N}$ is the* start non-terminal*, and $\longrightarrow$ is a finite set of* rules*. A rule is a triple $r = (A, \alpha, f)$ where $A \in \mathcal{N}$ is a non-terminal, α is an expression over $\mathcal{N}$ and Π with $\mathit{Free}(\alpha) \neq \emptyset$, and $f : \mathit{Free}(\alpha) \to \Pi$ is injective. We may write r as $A \longrightarrow_f \alpha$.*

In the sequel, let $|G| := |\Pi| + \sum_{A \longrightarrow_f \alpha} (|\alpha| + |M(\alpha)|)$ be the *size* of G ($|\alpha|$ denoting the length of α as a word and $|M(\alpha)|$ the number of events of $M(\alpha)$). Moreover, we set $\mathit{Proc}(G) := \bigcup_{A \longrightarrow_f \alpha} \mathit{Proc}(\alpha)$.

A *renaming* is a bijective mapping $\sigma : \mathbb{N} \to \mathbb{N}$. For an in-out PMSC $\mathcal{M} = (M, \mu)$ with $M = (\mathcal{P}, (E_p)_{p \in \mathcal{P}}, <_{\mathsf{p}}, <_{\mathsf{s}}, <_{\mathsf{m}}, \lambda)$, we let $\mathcal{M}\sigma = (M\sigma, \mu\sigma)$ where $M\sigma = (\sigma(\mathcal{P}), (E_{\sigma^{-1}(p)})_{p \in \sigma(\mathcal{P})}, <_{\mathsf{p}}, <_{\mathsf{s}}, <_{\mathsf{m}}, \lambda)$ and $\mu\sigma(\pi) = (\sigma(p), \sigma(q))$ if $\mu(\pi) = (p, q)$ is defined; otherwise, $\mu\sigma(\pi)$ is undefined. For a rule $r = (A, \alpha, f)$ with $\alpha = u_0.\mathcal{M}_1.u_1 \ldots \mathcal{M}_k.u_k$, we set $r\sigma := (A, \alpha\sigma, f\sigma)$ where $\alpha\sigma = u_0.\mathcal{M}_1\sigma.u_1 \ldots \mathcal{M}_k\sigma.u_k$ and $f\sigma(q) = f(\sigma^{-1}(q))$ for $q \in \mathit{Free}(\alpha\sigma)$.

A *configuration* of DMG $G = (\Pi, \mathcal{N}, S, \longrightarrow)$ is a pair $(\mathfrak{M}, \beta)$ where $\mathfrak{M} \in \mathrm{n}\mathbb{M}$ and $\beta \in (\mathrm{m}\mathbb{P} \cup \mathcal{N})^*$. If $\beta = \varepsilon$, then the configuration is said to be *final*. Let Conf_G be the set of configurations of G. A configuration is *initial* if it is of the form $((\mathsf{I}_p, \nu), S)$ for some $p \in \mathbb{N}$, where I_p is depicted in Figure 1 and $\nu(\pi) = p$ for all $\pi \in \Pi$. The semantics of G is given as the set of (named) MSCs appearing in final configurations that can be derived from an initial configuration by means of relations $\stackrel{r}{\Longrightarrow}_G \subseteq \mathit{Conf}_G \times \mathit{Conf}_G$ (for every rule r) and $\stackrel{\mathrm{e}}{\Longrightarrow}_G \subseteq \mathit{Conf}_G \times \mathit{Conf}_G$.

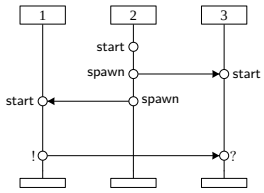

Fig. 6. not realizable

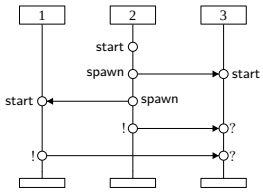

Fig. 7. 2-realizable

– For configurations $\mathcal{C} = (\mathfrak{M}, A.\gamma)$ and $\mathcal{C}' = (\mathfrak{M}, \alpha.\gamma)$, $\mathfrak{M} = (M, \nu)$, and $r \in \longrightarrow$, we let $\mathcal{C} \overset{r}{\Longrightarrow}_G \mathcal{C}'$ if there is a renaming σ such that $r\sigma = (A, \alpha, f)$, $\nu(f(p)) = p$ for all $p \in \mathit{Free}(\alpha)$, and $\mathit{Proc}(M) \cap \mathit{Bound}(\alpha) = \emptyset$.
– For configurations $\mathcal{C} = (\mathfrak{M}, \mathcal{M}.\gamma)$ and $\mathcal{C}' = (\mathfrak{M}', \gamma)$, we let $\mathcal{C} \overset{\mathrm{e}}{\Longrightarrow}_G \mathcal{C}'$ if $\mathfrak{M}' = \mathfrak{M} \circ \mathcal{M}$ (in particular, $\mathfrak{M} \circ \mathcal{M}$ must be defined).

We define $\Longrightarrow_G$ to be $\overset{\mathrm{e}}{\Longrightarrow}_G \cup \bigcup_{r \in \longrightarrow} \overset{r}{\Longrightarrow}_G$. The *language* of G is the set $L(G) := \{M \in \mathbb{M} \mid \mathcal{C}_0 \Longrightarrow_G^* ((M, \nu), \varepsilon)$ for some initial configuration $\mathcal{C}_0$ and $\nu\}$.

Let us formalize $G = (\Pi, \mathcal{N}, S, \longrightarrow)$ from Figure 4. Given the PMSCs M_1 and M_2 from Figure 1, we let $\mathcal{M}_1 = (M_1, \mu_1)$, $\mathcal{M}_2 = (M_2, \mu_2)$, and $\mathcal{M}_{12} = (M_1 \circ M_2, \mu_1)$ be in-out PMSCs with $\mu_1(\pi_1), \mu_2(\pi_1), \mu_2(\pi_2)$ undefined and $\mu_1(\pi_2) = (1, 2)$. We have

$$S \longrightarrow_{f_S} \mathcal{M}_1.A.\mathcal{M}_2.B \qquad S \longrightarrow_{f_S} \mathcal{M}_{12}.B \qquad B \longrightarrow_{f_B} \mathcal{M}_2$$
$$A \longrightarrow_{f_A} \mathcal{M}_1.A.\mathcal{M}_2 \qquad A \longrightarrow_{f_A} \mathcal{M}_{12}$$

where $f_S(1) = f_B(1) = \pi_1$ and $f_A(1) = f_B(2) = \pi_2$. Recall that $\Longrightarrow_G^*$ is illustrated in Figure 5. In a configuration, the part above a first non-terminal (if there is any) illustrates a named MSC. Note that $L(G) = L(\mathcal{A})$ for the DCA $\mathcal{A}$ from Figure 2.

5 Realizability of Dynamic MSC Grammars

Definition 4. *Let $L \subseteq \mathbb{M}$ be an MSC language. We call L* (proximity) realizable *if there is a DCA $\mathcal{A}$ such that $L = L(\mathcal{A})$. For $B \in \mathbb{N}$, we say that L is B-*realizable *if there is a DCA $\mathcal{A} = (X, \mathit{Msg}, Q, \Delta, \iota, F)$ such that $L = L(\mathcal{A})$ and $|X| \leq B$.*

The MSC M from Figure 1, considered as a singleton set, is 3-realizable. It is not 2-realizable. The singleton set from Figure 6 is not realizable, as process 3 receives a message from an unknown process. Adding a message makes it 2-realizable (Figure 7).

Theorem 5. *For a DMG G, one can decide in exponential time (wrt. $|G|$) if $L(G)$ is empty, and in doubly exponential time if $L(G)$ is realizable.*

Proof (sketch). Let $G = (\Pi, \mathcal{N}, S, \longrightarrow)$ be a DMG. To answer the first question, we build a tree automaton $\mathcal{A}_G$ that accepts all parse trees that correspond to successful derivations of G. Thus, we have $L(\mathcal{A}_G) = \emptyset$ iff $L(G) = \emptyset$. To answer the second question, we build a tree automaton $\mathcal{B}_G$ for those parse trees that give rise to realizable

MSCs (considering an MSC as a singleton set). One can show that $L(G)$ is realizable iff all MSCs in $L(G)$ are realizable. Thus, $L(G)$ is realizable iff $L(\mathcal{A}_G) \setminus L(\mathcal{B}_G) = \emptyset$.

We restrict here to the more involved construction of the tree automaton $\mathcal{B}_G$. To illustrate the idea of $\mathcal{B}_G$, we use the DMG G from Figure 4. The left-hand side of Figure 8 depicts the parse tree t of G that corresponds to the derivation from Figure 5. We, therefore, call t legal. Note that, for technical reasons, the function f from a rule $A \longrightarrow_f \alpha$ is located at its non-terminal A. The crucial point of the construction is to record, during a derivation, only a bounded amount of information on the current communication structure of the system. A communication structure is a partition of the set of process identifiers together with a binary relation that provides information on what processes know of other processes. The right-hand side of Figure 8 depicts a run of $\mathcal{B}_G$ on t. States, which are assigned to nodes, are framed by a rectangle. A state is hence either a pair of communication structures (together with a non-terminal, which is omitted), or an element from $\mathrm{m}\mathbb{P}$ that occurs in G. Our automaton works bottom-up. Consider the upper right leaf of the run tree, which is labeled with its state $\mathcal{M}_{12}$. Suppose that, when it comes to executing $\mathcal{M}_{12}$, the current communication structure C_0 of the system contains two processes carrying π_1 and π_2, respectively, that know each other (represented by the two edges). When we apply $\mathcal{M}_{12}$, the outcome will be a new structure, C_1, with a newly created process that collects process identifier π_2. Henceforth, the process carrying π_1 is known to that carrying π_2, but the converse does not hold. Names of nodes are omitted; instead, identical nodes are combined by a dotted line. We conclude that applying $A \longrightarrow_{f_A} \mathcal{M}_{12}$ has the effect of transforming C_0 into C_1. Therefore, (C_0, A, C_1) is a state that can be assigned to the (A, f_A)-labeled node, as actually done in our example run. It is important here that the first structure C_0 of a state (C_0, A, C_1) is *reduced* meaning that it restricts to nodes carrying process identifiers. The structure C_1, however, might keep some unlabeled nodes, but only those that stem from previously labeled ones. Hence, the set of states of $\mathcal{B}_G$ will be finite, though exponential in $|G|$. Like elements of $\mathrm{m}\mathbb{P}$, a triple (C_0, A, C_1) can be applied to a communication structure. E.g., the states that label the successors of the root transform D_0 into D_1. It is crucial here that, at any time, communicating processes know each other in the current communication structure. Now, we can reduce D_1 to D_2 by removing the middle node, as it does not carry a process identifier nor does it arise from an identifier-carrying node. Thus, (D_0, S, D_2) is the state assigned to the root. It is final, as D_0 consists of only one process, which carries all the process identifiers. A final state at the root ensures that the run tree represents a derivation that starts in the initial configuration gathering all process identifiers, and ends in a realizable MSC. □

Corollary 6. *For every DMG $G = (\Pi, \mathcal{N}, S, \longrightarrow)$, $L(G)$ is realizable iff $L(G)$ is $(|Proc(G)| + a \cdot |\Pi|)$-realizable where $a = \max\{|\alpha| \mid A \longrightarrow_f \alpha\}$.*

A realizable DMG is not necessarily implementable as a finite DCA, as the behavior of a single process need not be finite-state. We will determine a simple (but non-trivial) class of DMGs that are finitely realizable. To guarantee finiteness, we restrict to right-linear rules: a rule $A \longrightarrow_f \alpha$ is *right-linear* if α is of the form $\mathcal{M}$ or $\mathcal{M}.B$. Our class is inspired by *local-choice HMSCs* as introduced in [9]. Local-choice HMSCs are scenario descriptions over a fixed number of processes in which every choice of the specification

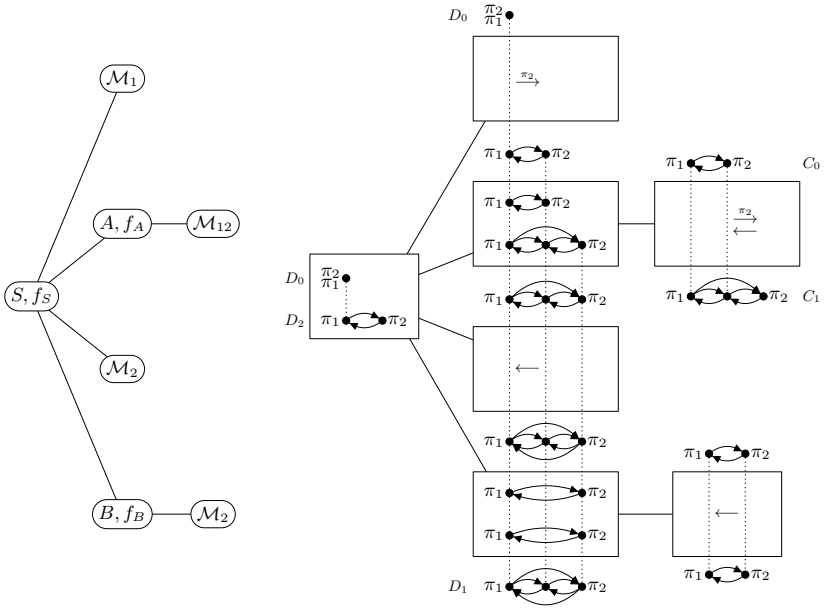

Fig. 8. A legal parse tree of G and a run of $\mathcal{B}_G$

is taken by a root process for that choice. This root is in charge of executing the minimal event of every scenario, and the subsequent messages can then be tagged to inform other processes about the choice. Note that locality allows for a deadlock-free implementation if the number of processes is fixed [7]. This is not guaranteed in our setting.

To adapt the notion of local-choice to DMGs, we essentially replace "process" in HMSCs by "process identifier". I.e., the root process that chooses the next rule to be applied must come with a process identifier π that is *active* in the current rule. So, for a right-linear rule $r = A \longrightarrow_f (M, \mu).\alpha$, we set $Active(r) = f(Free(M)) \cup \mathrm{dom}(\mu)$.

Definition 7. *A DMG $(\Pi, \mathcal{N}, S, \longrightarrow)$ is* local *if, for every rule $r = A \longrightarrow_f \alpha$, r is right-linear and $M(\alpha)$ has a unique minimal element. Moreover, if $\alpha = \mathcal{M}.B$, then there is $\pi \in Active(r)$ such that, for all B-rules $B \longrightarrow_g \beta$, $M(\beta)$ has a unique minimal element e satisfying $g(loc(e)) = \pi$.*

Theorem 8. *Let G be a local DMG such that $L(G)$ is realizable. There is a finite DCA $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ such that $L(\mathcal{A}) = L(G)$. Hereby, $|X|$ and $|Msg|$ are polynomial in $|G|$. Moreover, $|Q|$ and $|Act_{\mathcal{A}}|$ are exponential in $|G|$.*

Proof (sketch). A state of $\mathcal{A}$ will locally keep track of the progress that has been made to implement a rule. The root process may choose the next rule and inform its communication partners about this choice. The main difficulty in the implementation is the correct identification of process identities in terms of process variables. We introduce a variable x_π for each $\pi \in \Pi$ and a variable x_p for each $p \in Proc(G)$. As G is right-linear, $L(G)$ is indeed $|\Pi| + |Proc(G)|$-realizable. We pursue the following strategy of transmitting

process identities: When a process p is about to instantiate a non-terminal with a new rule r, an arbitrary renaming σ is applied. We assume hereby, that the "free processes" of r are known to p, though it is not clear to which variables they belong. Thus, σ is a simple guess, which has to be verified in the following. Indeed, the subsequent execution can pass through only if that guess is correct. The reason is that identifiers of free processes are held in local states and are sent in messages (in terms of events) so that the receiving process can be sure to receive from the correct process. Yet, we need to make sure that bound processes q are correctly identified. The idea is to enforce an update of x_q whenever a message is received from a process that "knows" q. □

6 Future Work

A nice theory of regular sets of MSCs over a fixed number of processes has been established [10] (a set of MSCs is regular if its linearization language is regular). We would like to extend this notion to our setting. Preferably, any regular set of MSCs should have an implementation in terms of a DCA. Note that, however, the linearizations of a set of (dynamic) MSCs are words over an infinite alphabet. Another challenge is to extend the class of DMGs that can be implemented by finite DCA beyond that of right-linear specifications (and preferably without deadlock). Last, we think that logics (e.g., MSO logic) may serve as an alternative specification language for DCA implementations.

References

1. B. Adsul, M. Mukund, K. Narayan Kumar, and Vasumat Narayanan. Causal closure for MSC languages. In *FSTTCS'05*, volume 3821 of *LNCS*, pages 335–347. Springer, 2005.
2. R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.
3. B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. Research report, LSV, ENS Cachan, 2010. Available at `http://www.lsv.ens-cachan.fr/Publis/`.
4. L. Bozzelli, S. La Torre, and A. Peron. Verification of well-formed communicating recursive state machines. *Theoretical Computer Science*, 403(2-3):382–405, 2008.
5. D. Brand and P. Zafiropulo. On communicating finite-state machines. *JACM*, 30(2), 1983.
6. T. Gazagnaire, B. Genest, L. Hélouët, P. S. Thiagarajan, and S. Yang. Causal message sequence charts. *Theor. Comput. Sci.*, 410(41):4094–4110, 2009.
7. B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level MSCs: Model-checking and realizability. *Journal on Comp. and System Sciences*, 72(4):617–647, 2006.
8. E.L. Gunter, A. Muscholl, and D. Peled. Compositional message sequence charts. *STTT*, 5(1):78–89, 2003.
9. L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from HMSCs. In *FMICS'00*, pages 203–224. Springer, 2000.
10. J. G. Henriksen, M. Mukund, K. Narayan Kumar, M. Sohoni, and P. S. Thiagarajan. A theory of regular MSC languages. *Information and Computation*, 202(1):1–38, 2005.
11. M. Leucker, P. Madhusudan, and S. Mukhopadhyay. Dynamic message sequence charts. In *FSTTCS'02*, volume 2556 of *LNCS*, pages 253–264. Springer, 2002.
12. M. Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theoretical Computer Science*, 309(1-3):529–554, 2003.
13. E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on message sequence charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.