Realizability of Dynamic MSC Languages

Benedikt Bollig, Loïc Hélouët

Realizability of Dynamic MSC Languages*

Benedikt Bollig[1] and Loïc Hélouët[2]

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

Version: August 4, 2010

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 EXPTIME-complete. Moreover, we 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 [RGG96], compositional MSCs [GMP03], or causal MSCs [GGH+09], requirements frequently consist in a finite set of scenarios over a fixed 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 realizable in terms of an 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, however, many applications rely on threads, and most protocols are designed for an open world, where all the participating actors are not known in advance. Example domains include mobile computing and ad-hoc networks. A first step towards MSCs over an evolving set of processes was made by Leucker, Madhusudan, and Mukhopadhyay [LMM02]. 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. However, dynamic process creation and its realizability are two important issues that must be considered jointly.

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

This paper introduces dynamic communicating automata (DCA) as a model of programs with process creation. It is very much inspired by the message-passing programming language Erlang [Arm07]. Like Erlang, a DCA has three communication primitives:

(1) a new process can be spawned,
(2) a message can be sent to an already existing process, and
(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 [BZ83], which allow only for actions of the form (2) and (3) and serve as an implementation model for HMSCs or compositional MSCs.

In a second step, we propose dynamic MSC grammars as a specification language. They are inspired by the fork-and-join grammars from [LMM02] but closer to an implementation. We keep the main idea of [LMM02]: when unfolding a grammar, MSCs are concatenated on the basis of finitely many process identifiers. While, in [LMM02], 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 [AEY05,Loh03,AMKN05]. In a dynamic framework where dynamic MSC grammars 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 checking proximity realizability is EXPTIME-complete. 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 local structure of a process. The next step is then to identify a class of dynamic MSC grammars that is realizable in terms of *finite* DCA.

The paper is organized as follows: In Section 2, we recall some basic concepts needed throughout the paper such as trees and tree automata. Section 3 introduces MSCs, which constitute the semantics of dynamic communicating automata and dynamic MSC grammars. The latter two are presented in Sections 4 and 5, respectively. In Section 6, we define realizability formally and show that the corresponding decision problem (as well as emptiness of grammars) is EXPTIME-complete. In the proofs, we make use of the tree-automata framework that we settled in Section 2. Section 7 presents a subclass of our grammars, local MSC grammars, and shows how they can be implemented automatically in terms of finite DCA. Finally, in Sections 9 and 8, we give an overview of related work and some directions for future work.

This article is a revised and extended version of [BH10].

2 Preliminaries

In this section, we settle some notation and basic concepts such as trees and tree automata.

A (finite) alphabet is a (finite, respectively) nonempty set. For an alphabet Γ, the set of finite words over Γ is denoted by Γ^*. It includes the empty word ε. For words $u, v \in \Gamma^*$, we let $u.v$ denote the concatenation uv of u and v. In particular, $u.\varepsilon = \varepsilon.u = u$.

For a set P, we let id_P denote $\{(p,p) \mid p \in P\}$ or, if required, the identity mapping on P. If R is a binary relation, we write $R \restriction P$ to denote $R \cap (P \times P)$. Moreover, $\restriction$ serves as the usual restriction operator for functions and structures. We set $R^{-1} = \{(p',p) \mid (p,p') \in R\}$ and, for any p, $R(p) = \{p' \mid (p,p') \in R\}$. Given binary relations $R_1, R_2 \subseteq P \times P$, we denote their product, as usual, by $R_1 \circ R_2 \subseteq P \times P$. Similarly, the composition of two mappings $\sigma_1, \sigma_2 : P \to P$ is denoted $\sigma_1 \circ \sigma_2 : P \to P$ and defined by $(\sigma_1 \circ \sigma_2)(p) = \sigma_2(\sigma_1(p))$ for all $p \in P$.

Trees and tree automata A *ranked alphabet* is a pair $\Omega = (\Gamma, \mathit{arity})$ where Γ is a finite alphabet and $\mathit{arity} : \Gamma \to 2^{\mathbb{N}}$ assigns to $a \in \Gamma$ a nonempty finite set $\mathit{arity}(a) \subseteq \mathbb{N}$ of *arities*. We call Ω *binary* if $\mathit{arity}(a) \subseteq \{0,2\}$ for all $a \in \Gamma$. It is *unary* if $\mathit{arity}(a) \subseteq \{0,1\}$ for all $a \in \Gamma$.

A *tree* over the ranked alphabet $\Omega = (\Gamma, \mathit{arity})$ is a pair $t = (\mathit{dom}_t, \mathit{val}_t)$. Its domain, $\mathit{dom}_t \subseteq \{1,2,\ldots\}^*$, is a nonempty finite prefix-closed set of *nodes* where, for all $u \in \mathit{dom}_t$ and $0 < i < j$, $u.j \in \mathit{dom}_t$ implies $u.i \in \mathit{dom}_t$. The node ε is called *root*, and a node u is a *successor* of a node v if $u = v.i$ for some i. Moreover, $\mathit{val}_t : \mathit{dom}_t \to \Gamma$ such that, for all $u \in \mathit{dom}_t$, the number of successors of u is in $\mathit{arity}(\mathit{val}_t(u))$. A node without any successor is a *leaf*. Otherwise, it is an *inner node*.

A *tree automaton* over Ω is a triple $\mathcal{A} = (Q, \leftarrow, F)$ where Q is a nonempty finite set of *states*, $F \subseteq Q$ is the set of *final states*, and $\leftarrow \subseteq Q \times \Gamma \times \bigcup_{i \in \mathbb{N}} Q^i$ is a finite *transition relation*. A run of $\mathcal{A}$ on a tree $t = (\mathit{dom}_t, \mathit{val}_t)$ over Ω is a mapping $\mathit{run} : \mathit{dom}_t \to Q$ such that, for all nodes $u \in \mathit{dom}_t$, say, with n successors, we have

$$\mathit{run}(u) \overset{\mathit{val}_t(u)}{\leftarrow} (\mathit{run}(u.1), \ldots, \mathit{run}(u.n)) \,.$$

The run is accepting if $\mathit{run}(\varepsilon) \in F$. The language of $\mathcal{A}$, denoted by $L(\mathcal{A})$, is the set of trees t over Ω such that there is an accepting run of $\mathcal{A}$ on t. It is well known that checking emptiness of $L(\mathcal{A})$ can be done in time $\mathcal{O}(|Q| \cdot (|Q| \cdot |{\leftarrow}|))$. Moreover, determinization and complementation generally cause an exponential blow up.

We will later consider decision problems for dynamic MSC grammars and establish lower bounds by reductions from the intersection problem for tree automata.

Problem 1. Intersection for tree automata:
INSTANCE: Binary ranked alphabet Ω; $n \geq 1$; tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ over Ω.
QUESTION: Is $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ empty?

When we later move to so-called local grammars, a lower bound can be established using a variant of the former problem where a unary ranked alphabet is used. Then, tree automata actually reduce to finite automata.

Problem 2. Intersection for finite automata:
INSTANCE: Unary ranked alphabet Ω; $n \geq 1$; tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ over Ω.
QUESTION: Is $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n)$ empty?

The complexity of both problems is well-known:

Theorem 3 ([Sei94]). *Intersection for tree automata is EXPTIME-complete.*

Theorem 4 ([Koz77]). *Intersection for finite automata is PSPACE-complete.*

3 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 $\prec_{\mathsf{proc}}$. 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 $\prec_{\mathsf{msg}}$ 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, $\prec_{\mathsf{spawn}}$ 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 5 (MSC). *An* MSC *is a tuple* $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda)$ *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) $\prec = (\prec_{\mathsf{proc}}, \prec_{\mathsf{spawn}}, \prec_{\mathsf{msg}})$ *where* $\prec_{\mathsf{proc}}, \prec_{\mathsf{spawn}}, \prec_{\mathsf{msg}}$ *are binary relations on* E.

There are further requirements:

(1) $\leq := (\prec_{\mathsf{proc}} \cup \prec_{\mathsf{spawn}} \cup \prec_{\mathsf{msg}})^*$ *is a partial order,*
(2) $(E, \leq)$ *has a unique minimal element, denoted by* $start(M)$*,*
(3) $\lambda^{-1}(\mathsf{start}) = \{e \in E \mid$ *there is no* $e' \in E$ *such that* $e' \prec_{\mathsf{proc}} e\}$*,*
(4) $\prec_{\mathsf{proc}} \subseteq \bigcup_{p\in\mathcal{P}}(E_p \times E_p)$*,*
(5) for each $p \in \mathcal{P}$*,* $\prec_{\mathsf{proc}} \cap (E_p \times E_p)$ *is the direct-successor relation of some total order on* E_p*,*
(6) $\prec_{\mathsf{spawn}}$ *induces a bijection between* $\lambda^{-1}(\mathsf{spawn})$ *and* $\lambda^{-1}(\mathsf{start}) \setminus \{start(M)\}$*,*
(7) $\prec_{\mathsf{msg}}$ *induces a bijection between* $\lambda^{-1}(!)$ *and* $\lambda^{-1}(?)$ *satisfying the following (FIFO): for* $e_1, e_2 \in E_p$ *and* $\hat{e}_1, \hat{e}_2 \in E_q$ *with* $e_1 \prec_{\mathsf{msg}} \hat{e}_1$ *and* $e_2 \prec_{\mathsf{msg}} \hat{e}_2$*, both* $p \neq q$ *and* $e_1 \leq e_2$ *iff* $\hat{e}_1 \leq \hat{e}_2$*.*

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 does not necessarily have start events on each process.

Definition 6 (partial MSC). *Let $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda)$ be an MSC and let $E' \subseteq E$ be a nonempty set satisfying $E' = \{e \in E \mid (e, e') \in \prec_{\mathsf{msg}} \cup \prec_{\mathsf{spawn}} \cup \leq^{-1} \text{ for some } e' \in E'\}$ (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 and M_2 are both PMSCs, but not MSCs. Thus, we have $M_1, M_2 \in \mathbb{P} \setminus \mathbb{M}$.

Let $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda)$ be a PMSC. For $e \in E$, we denote by $loc(e)$ the unique process $p \in \mathcal{P}$ such that $e \in E_p$. If M is not clear from the context, we rather write $loc_M(e)$. In other words, event e is located on process $loc(e)$. For every $p \in \mathcal{P}$, there are a unique minimal and a unique maximal event in the total order $(E_p, \leq \cap (E_p \times E_p))$, which we denote by $\min_p(M)$ and $\max_p(M)$, respectively. We also set $\min(M) = \{\min_p(M) \mid p \in \mathcal{P}\}$ and $\max(M) = \{\max_p(M) \mid p \in \mathcal{P}\}$. For $e \in E \setminus \min(M)$, we set $pred(e)$ to be the *predecessor* of e, i.e., the unique event e' such that $e' \prec_{\mathsf{proc}} e$. For $e \in \min(M)$, $pred(e)$ remains undefined. 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, $Bound(\mathsf{I}_p) = \{p\}$, $Free(M_1) = \{1\}$, $Free(M_2) = \{1, 2\}$, and $Bound(M) = \{1, 2, 3, 4\}$. In particular, $Free(N) = \emptyset$ for every MSC N.

Visually, concatenation of PMSCs corresponds to drawing identical processes one below the other. For $i = 1, 2$, let $M^i = (\mathcal{P}^i, (E^i_p)_{p\in\mathcal{P}^i}, \prec^i, \lambda^i)$ be PMSCs. Consider the structure $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda)$ where

- $\mathcal{P} = \mathcal{P}^1 \cup \mathcal{P}^2$,
- $E_p = E^1_p \uplus E^2_p$ for all $p \in \mathcal{P}$ (assuming $E^i_p = \emptyset$ if $p \notin \mathcal{P}^i$),
- $\prec_{\mathsf{proc}} = \prec^1_{\mathsf{proc}} \cup \prec^2_{\mathsf{proc}} \cup \{(\max_p(M^1), \min_p(M^2)) \mid p \in \mathcal{P}_1 \cap \mathcal{P}_2\}$,
- $\prec_{\mathsf{msg}} = \prec^1_{\mathsf{msg}} \cup \prec^2_{\mathsf{msg}}$, and
- $\lambda = \lambda^1 \cup \lambda^2$ (with the obvious meaning).

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 $E^1_p \neq \emptyset$). Concatenation is illustrated in Figure 5.

In the context of partial orders, it is natural to consider linearizations, which extend a partial order to a total order. Actually, a linearization can be considered as a word over an infinite alphabet

$$\Sigma = \{!(p, q)\,,\ ?(p, q)\,,\ \mathsf{spawn}(p, q) \mid p, q \in \mathbb{N} \text{ with } p \neq q\}\,.$$

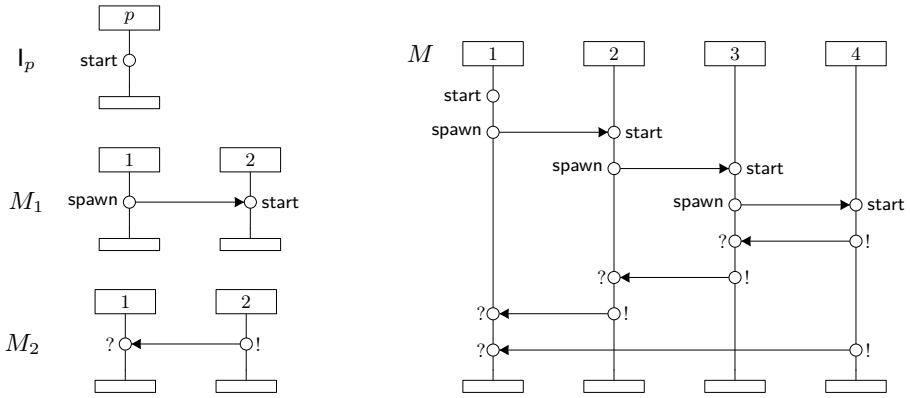

Fig. 1. (Partial) message sequence charts

For a PMSC $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda)$, we let $poset(M)$ be the Σ-labeled partial order $(E', \leq', \lambda')$ where $E' = E \setminus \lambda^{-1}(\mathsf{start})$, $\leq' = \leq \cap (E' \times E')$, and $\lambda' : E' \to \Sigma$ such that, for all $(e, \hat{e}) \in \prec_{\mathsf{spawn}}$, we have $\lambda'(e) = \mathsf{spawn}(loc(e), loc(\hat{e}))$, and, for all $(e, \hat{e}) \in \prec_{\mathsf{msg}}$, 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^*. For example,

$$\mathsf{spawn}(1,2)\ \mathsf{spawn}(2,3)\ \mathsf{spawn}(3,4)\ !(4,3)\ !(4,1)\ ?(4,3)\ !(3,2)\ ?(3,2)\ !(2,1)\ ?(2,1)\ ?(4,1)$$

$$\mathsf{spawn}(1,2)\ \mathsf{spawn}(2,3)\ \mathsf{spawn}(3,4)\ !(4,3)\ ?(4,3)\ !(3,2)\ ?(3,2)\ !(2,1)\ ?(2,1)\ !(4,1)\ ?(4,1)$$

are two linearization of the MSC M from Figure 1. I.e., they are both contained in $Lin(poset(M))$.

In the following, we do not distinguish PMSCs that differ only in their event names.[3] Moreover, we say that two PMSCs are isomorphic if the one can be obtained from the other via renaming of processes. A *renaming* is a bijective mapping $\sigma : \mathbb{N} \to \mathbb{N}$. Applied to a PMSC $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda)$, we obtain $M\sigma = (\sigma(\mathcal{P}), (E_{\sigma^{-1}(p)})_{p\in\sigma(\mathcal{P})}, \prec, \lambda)$. With this, PMSCs M and N are said to be *isomorphic* if $N = M\sigma$ for some renaming σ. The isomorphism class of M is denoted $[M]$. For a set L of PMSCs, we let $[L] = \bigcup_{M\in L}[M]$. We call L *closed* if $L = [L]$.

4 Dynamic Communicating Automata

Dynamic communicating automata (DCA) extend classical communicating finite-state machines [BZ83]. 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 Section 8 for some references). Each process p holds a set of process variables. Their values represent process identities that p remembers

[3] Alternatively, we could require that the events of a process p are named $(p, 1), (p, 2), \ldots, (p, n)$, starting with the minimal and ending up in the maximal one.

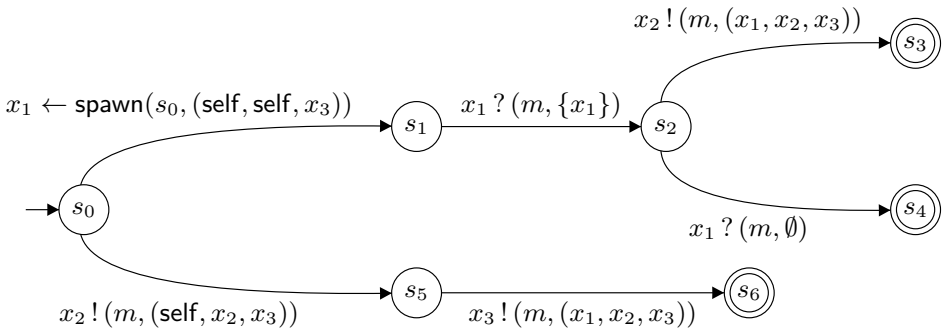

Fig. 2. A dynamic communicating automaton

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. Thus, channels are private and, in particular, a receiving process has to know the sending process.

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 $x_1\,?\,(m, \{x_1\})$: it receives $(m, (4, 3, 1))$ and sets x_1 to 4, whereas variables x_2 and x_2 are not touched. 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.

Let us formally define syntax and semantics of DCA. Note that, apart from spawn, send, and receive actions, a DCA will also provide a renaming primitive, which allows a process to reorder its variables.

Definition 7 (dynamic communicating automaton). *A* dynamic communicating automaton *(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 the set of* actions*, which are 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.*

Operational semantics We first define the semantics of a DCA as a word language over Σ. It is actually the language of an infinite transition system. We will later argue that 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 nonempty 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 $i \in \{1, \ldots, n\}$, $(c_{i-1}, a_i, c_i) \in \Longrightarrow_{\mathcal{A}}$. 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.

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

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

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

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

Fig. 3. Global transition relation of a DCA

Partial-order semantics Alternatively, the semantics of DCA $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ can be given directly for MSCs. This semantics abstracts from channel contents and will be easier to cope with in proofs that deal with MSCs.

For an MSC $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda)$, an *MSC run* of $\mathcal{A}$ on M is a pair $(state, proc)$ where $state : E \to Q$ and $proc : E \to \mathcal{P}^X$. Intuitively, $state(e)$ is the state of process $loc(e)$ reached after executing e, and $state(e)[x]$ is the value of $x \in X$ after the execution of e. We require that

- $state(start(M)) = \iota$,
- $proc(start(M))[x] = loc(start(M))$ for all $x \in X$,
- for all $e_0, e, \hat{e} \in E$ such that $e_0 \prec_{\textsf{proc}} e \prec_{\textsf{spawn}} \hat{e}$, there are $n \geq 0$ and $n+1$ transitions

$$(s_0, \textsf{rn}(\sigma_1), s_1), \ldots, (s_{n-1}, \textsf{rn}(\sigma_n), s_n), (s_n, \hat{x} \leftarrow \textsf{spawn}(\hat{s},\eta), s) \in \Delta$$

  such that $state(e_0) = s_0$, $state(e) = s$, $state(\hat{e}) = \hat{s}$, $proc(e)[y] = proc(e_0)[(\sigma_1 \circ \ldots \circ \sigma_n)(y)]$ for all $y \in X \setminus \{\hat{x}\}$, $proc(e)[\hat{x}] = loc(\hat{e})$, $proc(\hat{e})[y] = proc(e)[\eta(y)]$ if $\eta(y) \neq \textsf{self}$, and $proc(\hat{e})[y] = loc(e)$ if $\eta(y) = \textsf{self}$,

– for all $e_0, \hat{e}_0, e, \hat{e} \in E$ such that $e_0 \prec_{\mathsf{proc}} e \prec_{\mathsf{msg}} \hat{e}$ and $\hat{e}_0 \prec_{\mathsf{proc}} \hat{e}$, there are transitions

$$(s_0, \mathsf{rn}(\sigma_1), s_1), \ldots, (s_{n-1}, \mathsf{rn}(\sigma_n), s_n), (s_n, \hat{x}\,!\,(m, \eta), s) \in \Delta$$
$$(\hat{s}_0, \mathsf{rn}(\hat{\sigma}_1), \hat{s}_1), \ldots, (\hat{s}_{\ell-1}, \mathsf{rn}(\hat{\sigma}_\ell), \hat{s}_\ell), (\hat{s}_\ell, x\,?\,(m, Y), \hat{s}) \in \Delta$$

such that $\mathit{state}(e_0) = s_0$, $\mathit{state}(\hat{e}_0) = \hat{s}_0$, $\mathit{state}(e) = s$, $\mathit{state}(\hat{e}) = \hat{s}$, $\mathit{proc}(e)[\hat{x}] = \mathit{loc}(\hat{e})$, $\mathit{proc}(\hat{e})[x] = \mathit{loc}(e)$, $\mathit{proc}(e)[y] = \mathit{proc}(e_0)[(\sigma_1 \circ \ldots \circ \sigma_n)(y)]$ for all $y \in X \setminus \{\hat{x}\}$, and, for all $y \in X$,

$$\mathit{proc}(\hat{e})[y] = \begin{cases} \mathit{loc}(e) & \text{if } y \in Y \text{ and } \eta(y) = \mathsf{self} \\ \mathit{proc}(e)[\eta(y)] & \text{if } y \in Y \text{ and } \eta(y) \neq \mathsf{self} \\ \mathit{proc}(\hat{e}_0)[(\hat{\sigma}_1 \circ \ldots \circ \hat{\sigma}_\ell)(y)] & \text{if } y \notin Y \end{cases}$$

The MSC run is *accepting* if $\{\mathit{state}(\max_p(M)) \mid p \in \mathcal{P}\} \subseteq F$. The *(MSC) language* of $\mathcal{A}$ is $L(\mathcal{A}) := \{M \in \mathbb{M} \mid \text{there is an accepting MSC run of } \mathcal{A} \text{ on } M\}$. It is easy to see that $[L(\mathcal{A})] = L(\mathcal{A})$.

Consider the *finite* DCA $\mathcal{A}$ from Figure 2. For $n \geq 2$, suppose that $M(n)$ denotes the MSC where an initial phase creates processes $1, \ldots, n$, whereupon n sends a message to $n-1$, which is relayed back to process 1, followed by a message from n to 1. Thus, $M(4)$ is precisely the MSC M from Figure 1. Then, we have $L(\mathcal{A}) = [\{M(n) \mid n \geq 2\}]$.

It is standard to show that the operational and the partial-order semantics of DCA coincide:

Proposition 8. *For a DCA $\mathcal{A}$, we have*

– $L(\mathcal{A}) = \{M \in \mathbb{M} \mid \mathit{Lin}(\mathit{poset}(M)) \subseteq \mathcal{L}(\mathcal{A})\}$ *and, equivalently,*
– $\mathcal{L}(\mathcal{A}) = \bigcup_{M \in L(\mathcal{A})} \mathit{Lin}(\mathit{poset}(M))$.

DCA actually generalize the classical setting of communicating finite-state machines [BZ83]. To simulate them, the starting process spawns the required number of processes and broadcasts their identities to any other process.

5 Dynamic MSC Grammars

In this section, we introduce *dynamic MSC grammars*. They are inspired by the grammars from [LMM02], but take into account that we want to implement them in terms of DCA. We keep the main idea of [LMM02] 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 dynamic MSC grammar 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.

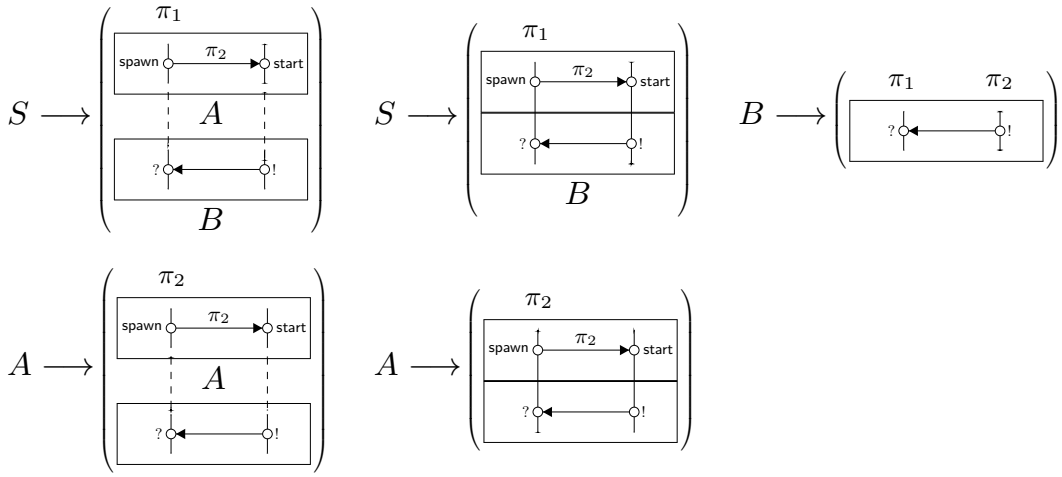

Fig. 4. A dynamic MSC grammar

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 6, which is a sequence of configurations, each consisting of an upper and a lower part. The upper part is a *named* MSC [LMM02], 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 6, where π_2 has moved from the second to the third process).

We now formalize the components of a dynamic MSC grammar. Let Π be a nonempty and finite set of *process identifiers*.

Definition 9 (named MSC). *A* named MSC *over Π is a pair (M, ν) where M is an MSC and $\nu : \Pi \to Proc(M)$.*

Definition 10 (migration PMSC). *A* migration PMSC *over Π is a pair (M, μ) where M is a* PMSC *with $Free(M) \neq \emptyset$, and $\mu : \Pi \rightharpoonup Free(M) \times Proc(M)$ is a partial mapping.*

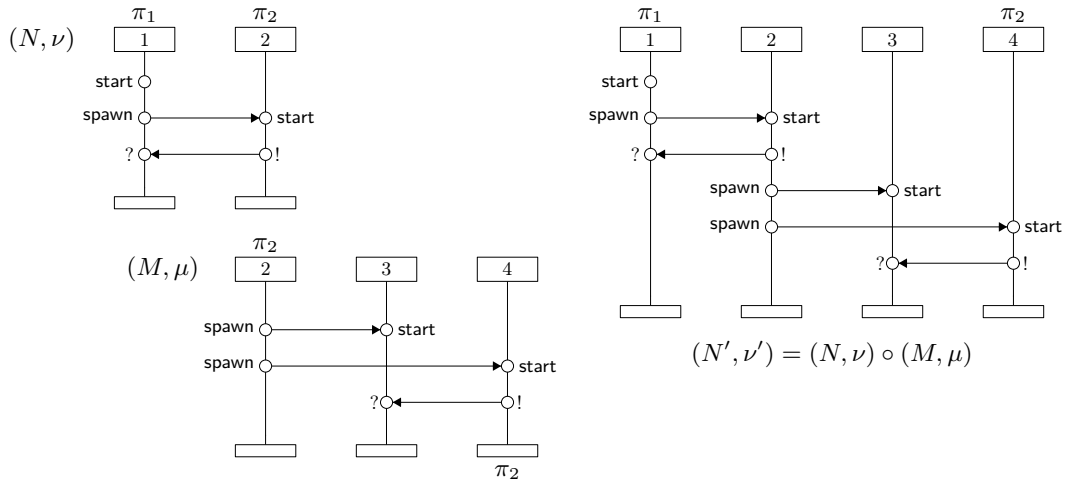

Fig. 5. Concatenation

We denote by $\mathrm{n}\mathbb{M}$ the set of named MSCs and by $\mathrm{m}\mathbb{P}$ the set of migration PMSCs over Π (we assume that Π is clear from the context). We let $\mathfrak{M}$ range over named MSCs and $\mathcal{M}$ over migration PMSCs. For $\mathfrak{M} = (N, \nu) \in \mathrm{n}\mathbb{M}$ and $\mathcal{M} = (M, \mu) \in \mathrm{m}\mathbb{P}$, we may write $\mathit{Proc}(\mathfrak{M})$ and $\mathit{Proc}(\mathcal{M})$ to denote, respectively, $\mathit{Proc}(N)$ and $\mathit{Proc}(M)$.

A derivation of a dynamic MSC grammar 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 migration PMSCs that will be evaluated later, proceeding from left to right. If $\beta = \mathcal{M}.\gamma$ for some migration 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 $(N, \nu) \in \mathrm{n}\mathbb{M}$ and $(M, \mu) \in \mathrm{m}\mathbb{P}$. Then, $(N, \nu) \circ (M, \mu)$ is defined if $N \circ M$ is defined and contained in $\mathbb{M}$, and, for all $\pi \in \Pi$ such that $\mu(\pi) = (p, q)$ is defined, we have $\nu(\pi) = p$. If defined, we set $(N, \nu) \circ (M, \mu) := (N', \nu')$ where $N' = N \circ M$, $\nu'(\pi) = \nu(\pi)$ if $\mu(\pi)$ is undefined, and $\nu'(\pi) = q$ if $\mu(\pi) = (p, q)$ is defined.

Concatenation is illustrated in Figure 5. The set of process identifiers is $\{\pi_1, \pi_2\}$. We have $\nu(\pi_1) = 1$ and $\nu(\pi_2) = 2$. Moreover, $\mu(\pi_2) = (2, 4)$ whereas $\mu(\pi_1)$ is undefined. In the resulting named MSC, we have $\nu'(\pi_1) = 1$ and $\nu'(\pi_2) = 4$.

We already mentioned that, in 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. In other words, 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 $\mathit{Bound}(\alpha) := \mathit{Bound}(M(\alpha))$.

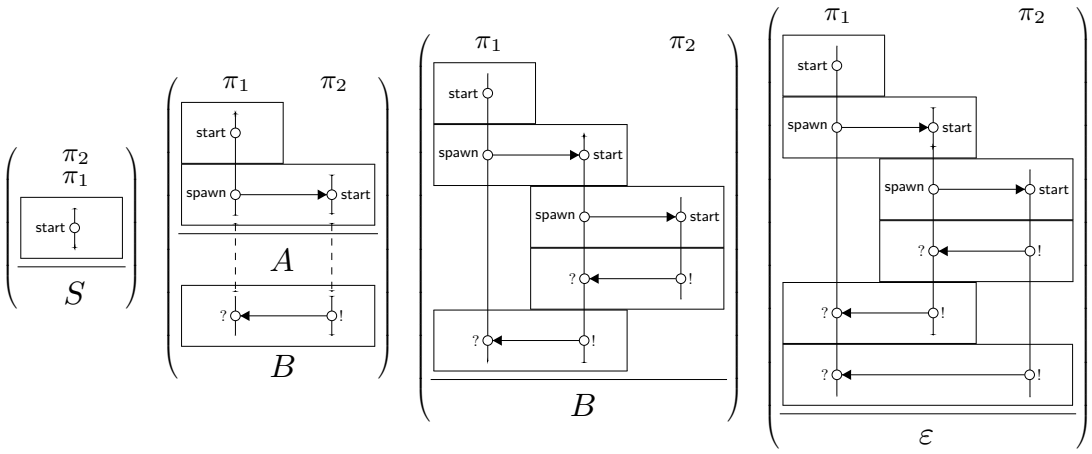

Fig. 6. A derivation

Definition 11 (dynamic MSC grammar). *A* dynamic MSC grammar *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* $Free(\alpha) \neq \emptyset$*, and* $f : Free(\alpha) \to \Pi$ *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 $Proc(G) := \bigcup_{A \longrightarrow_f \alpha} Proc(\alpha)$.

A renaming σ (recall that σ is a bijective mapping from $\mathbb{N}$ to $\mathbb{N}$) can be applied to migration PMSCs and expressions as follows. For a migration PMSC $\mathcal{M} = (M, \mu)$ with $M = (\mathcal{P}, (E_p)_{p \in \mathcal{P}}, \prec, \lambda)$, we let $\mathcal{M}\sigma = (M\sigma, \mu\sigma)$ where $\mu\sigma(\pi) = (\sigma(p), \sigma(q))$ 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 $\alpha\sigma = u_0.\mathcal{M}_1\sigma.u_1 \ldots \mathcal{M}_k\sigma.u_k$ is a new expression and $f\sigma(q) = f(\sigma^{-1}(q))$ for $q \in Free(\alpha\sigma)$.

A *configuration* of a dynamic MSC grammar $G = (\Pi, \mathcal{N}, S, \longrightarrow)$ 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 $\overset{r}{\Longrightarrow}_G \subseteq Conf_G \times Conf_G$ (for every rule r) and $\overset{\mathrm{e}}{\Longrightarrow}_G \subseteq Conf_G \times Conf_G$ (where e stands for "evaluate"):

- For configurations $\mathcal{C} = (\mathfrak{M}, A.\gamma)$ and $\mathcal{C}' = (\mathfrak{M}, \alpha.\gamma)$ with $\mathfrak{M} = (M, \nu)$, $r \in \longrightarrow$, and a renaming σ, we let $\mathcal{C} \overset{r}{\underset{\sigma}{\Longrightarrow}}_G \mathcal{C}'$ if $r\sigma = (A, \alpha, f)$, $\nu(f(p)) = p$ for all $p \in Free(\alpha)$, and $Proc(M) \cap Bound(\alpha) = \emptyset$. Moreover, we let $\overset{r}{\Longrightarrow}_G = \bigcup_{\sigma \text{ renaming}} \overset{r}{\underset{\sigma}{\Longrightarrow}}_G$.

– 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

$$L(G) := \{M \in \mathbb{M} \mid \mathcal{C}_0 \Longrightarrow_G^* ((M, \nu), \varepsilon) \text{ for some initial configuration } \mathcal{C}_0 \text{ and } \nu\}.$$

Let us formalize the grammar $G = (\Pi, \mathcal{N}, S, \longrightarrow)$ as depicted in 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 migration 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 6. 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.

6 Realizability of Dynamic MSC Grammars

In this section, we introduce our notion of realizability. An MSC language L is realizable if it is recognized by some DCA. If that DCA can do with at most $B \geq 1$ process variables, then L is called B-realizable.

Definition 12 (realizable). *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 $B \geq 1$, we say that L is B-*realizable *if there is a DCA $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ such that $L = L(\mathcal{A})$ and $|X| \leq B$.*

A dynamic MSC grammar G is (B-)*realizable* if so is $L(G)$. Moreover, we call an MSC M *realizable* if so is $[M]$.

For example, the MSC M from Figure 1 is 3-realizable. To implement M, a process must keep in memory its left and its right neighbor. Moreover, in the spawning phase, it remembers process 1 (to communicate it eventually to process 4). In the second phase, it can forget about 1 and keep in mind process 4 (to communicate it to process 1). Two process variables are actually not sufficient so that M is not 2-realizable. The MSC from Figure 7 is not realizable, as process 1 sends a message to process 3, which is unknown to 1: there is no way to communicate the identity of 3 to 1. Adding a message from 2 (which knows 3) to 1 makes the MSC 2-realizable (Figure 8). The (closure of the) language depicted in Figure 9, n ranging over $\mathbb{N}$, is realizable, though not by a finite DCA. Indeed, our definition of realizability does not make any assumption on the set of states nor the set of messages in a DCA. As a consequence, realizability of a set of MSCs actually reduces to realizability of *each* MSC in that set (see Lemma 15 below). In turn, realizability of a single MSC can be reduced to the question if events can be consistently labeled with processes known after their execution such that subsequent

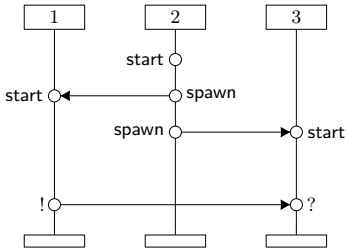

Fig. 7. not realizable

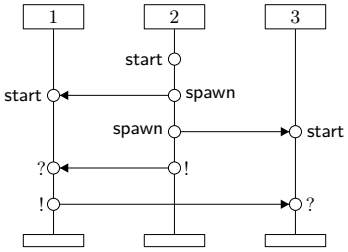

Fig. 8. 2-realizable

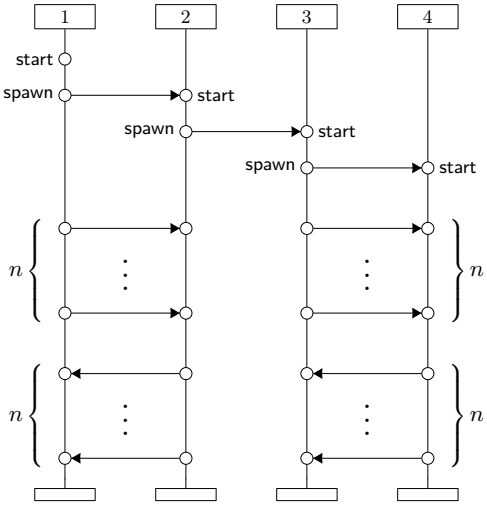

Fig. 9. A realizable MSC language

events communicate only with these known processes. This will be elaborated formally in the following.

Let $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda)$ be a PMSC and let $K \subseteq ((\mathbb{N} \setminus \mathit{Bound}(M)) \times (\mathbb{N} \setminus \mathit{Bound}(M))) \setminus \mathsf{id}_{\mathbb{N}}$ be a finite relation. Intuitively, K represents some "is-known-to" relation for processes that are already active, before it comes to the execution of M. Let $P = \{p \in \mathbb{N} \mid (p, \hat{p}) \in K \cup K^{-1}$ for some $\hat{p} \in \mathbb{N}\}$ be the set of those active processes. Let $\chi : E \to 2^{P\cup\mathcal{P}}$ be some mapping. We will define when χ correctly captures the transmission of process identities in M, i.e., $\chi(e)$ contains only processes that $\mathit{loc}(e)$ may know after the execution of e. For $e \in E$, let $\chi_{\mathit{pred}}(e)$ denote $\chi(\mathit{pred}(e))$ if $\mathit{pred}(e)$ is defined, and $K^{-1}(\mathit{loc}(e))$, otherwise. We call χ a *knowledge mapping* for M under K if the following hold:

(a) $M \in \mathbb{M}$ implies $\chi(\mathit{start}(M)) \subseteq \emptyset$

(b) for all $(e, \hat{e}) \in \prec_{\mathsf{spawn}}$, both $\begin{cases} \chi(e) \subseteq \chi_{\mathit{pred}}(e) \cup \{\mathit{loc}(\hat{e})\} & \text{and} \\ \chi(\hat{e}) \subseteq \chi_{\mathit{pred}}(e) \cup \{\mathit{loc}(e)\} \end{cases}$

(c) for all $(e, \hat{e}) \in \prec_{\mathsf{msg}}$, both $\begin{cases} \chi(e) \subseteq \chi_{\mathit{pred}}(e) & \text{and} \\ \chi(\hat{e}) \subseteq (\chi_{\mathit{pred}}(e) \cup \chi_{\mathit{pred}}(\hat{e}) \cup \{\mathit{loc}(e)\}) \setminus \{\mathit{loc}(\hat{e})\} \end{cases}$

Moreover, χ is called

- *B-bounded* (for $B \geq 1$) if $|\chi(e)| \leq B$ for all $e \in E$,
- *valid* (for K) if, for all $(e, \hat{e}) \in \prec_{\mathsf{msg}}$, both $\mathit{loc}(\hat{e}) \in \chi_{\mathit{pred}}(e)$ and $\mathit{loc}(e) \in \chi_{\mathit{pred}}(\hat{e})$.

When we replace, in the three items (a), (b), and (c), the subset relations $\subseteq$ by $:=$, we obtain the inductive definition of the *maximal knowledge mapping* for M under K, which we denote by χ_M^K. Moreover, χ_M will denote $\chi_M^\emptyset$.

Figure 10 exemplifies the mapping χ_M of an MSC M. Note that this mapping is valid and 2-bounded.

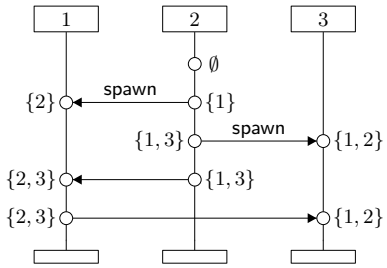

Fig. 10. The knowledge mapping χ_M of an MSC M

Lemma 13. *Let $M \in \mathbb{M}$ and $B \geq 1$. Then, M is B-realizable iff there is a knowledge mapping χ for M under $\emptyset$ such that χ is B-bounded and valid.*

Proof. Let $M \in \mathbb{M}$ be realizable. There are a DCA $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ and an accepting MSC run $(proc, state)$ of $\mathcal{A}$ on M. One can readily verify that the mapping $\chi : E \to \mathcal{P}$ given by $\chi(e) = \{proc(e)[x] \mid x \in X\} \setminus \{loc(e)\}$ is a valid knowledge mapping for M under $\emptyset$. Obviously, χ is $|X|$-bounded.

Conversely, let $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda) \in \mathbb{M}$ and χ be a B-bounded valid knowledge mapping for M under $\emptyset$. We construct a DCA $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ such that $|X| \leq B$ and $L(\mathcal{A}) = [M]$ as follows: $Q = E$, $\iota = start(M)$, $F = \max(M)$, $Msg = \prec_{\mathsf{msg}}$, and $X = \{x_i \mid i \in \{1, \ldots, B\}\}$. Let us turn to Δ. For all $i \in \{1, \ldots, B\}$, $Y \subseteq X$, $\eta \in (X \uplus \{\mathsf{self}\})^X$, and $e_0, e, \hat{e}_0, \hat{e} \in E$ with $e_0 \prec_{\mathsf{proc}} e \prec_{\mathsf{msg}} \hat{e}$ and $\hat{e}_0 \prec_{\mathsf{proc}} \hat{e}$, we introduce transitions $(e_0, x_i\,!\,((e, \hat{e}), \eta), e)$ and $(\hat{e}_0, x_i\,?\,((e, \hat{e}), \eta), \hat{e})$. Finally, for all $i \in \{1, \ldots, B\}$, $\eta \in (X \uplus \{\mathsf{self}\})^X$, and $e_0, e, \hat{e} \in E$ with $e_0 \prec_{\mathsf{proc}} e \prec_{\mathsf{spawn}} \hat{e}$, we introduce a transition $(e_0, x_i \leftarrow \mathsf{spawn}(\hat{e}, \eta), e)$. □

Note that the DCA that we construct in the proof of the previous lemma is nondeterministic and basically guesses a correct assignment of process variables. We point out that we could likewise construct a deterministic and deadlock-free automaton, which, however, is not needed for our result.

Lemma 14. *Let $M \in \mathbb{M}$. Then, M is realizable iff χ_M is a valid knowledge mapping.*

Proof. Let $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda) \in \mathbb{M}$ be realizable. Then, there are a DCA $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ and an accepting MSC run $(proc, state)$ of $\mathcal{A}$ on M. It can be easily checked that, for all $e \in E$, $\{proc(e)[x] \mid x \in X\} \setminus \{loc(e)\} \subseteq \chi_M(e)$. From the latter, we can deduce that χ_M is valid.

Let χ_M be a valid knowledge mapping for $M \in \mathbb{M}$. Then, χ_M is $|\mathcal{P}|$-bounded and, by Lemma 13, realizable. □

Lemma 15. *Let $L \subseteq \mathbb{M}$ be closed. Then, L is realizable iff every MSC in L is realizable.*

Proof. Suppose L is realizable and let $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda) \in L$. Then, there is a DCA $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ with $L(\mathcal{A}) = L$. In particular, there is an accepting MSC run $(proc, state)$ of $\mathcal{A}$ on M. Again, for all $e \in E$, $\{proc(e)[x] \mid x \in X\} \setminus \{loc(e)\} \subseteq \chi_M(e)$ so that χ_M is valid. By Lemma 14, M is realizable.

Then, there are a DCA $\mathcal{A}$ with $L(\mathcal{A}) = L$ and an accepting run of $\mathcal{A}$ on (some linearization of) M. Clearly, this run can be added to the state information of $\mathcal{A}$ to ensure that only M can be executed. Thus, M is realizable as well.

Now suppose that, for every $M \in L$, M is realizable, i.e., there is a DCA $\mathcal{A}_M = (X_M, Msg_M, Q_M, \Delta_M, \iota_M, F_M)$ such that $L(\mathcal{A}_M) = [M]$. We assume that $Msg_M \cap Msg_N = Q_M \cap Q_N = \emptyset$ for any two distinct MSCs $M, N \in L$. One can construct a DCA $\mathcal{A} = (X, Msg, Q, \Delta, \iota, F)$ with $L(\mathcal{A}) = L$ as follows: $X = \bigcup_{M\in L} X_M$, $Msg = \bigcup_{M\in L} Msg_M$, $Q = \bigcup_{M\in L} Q_M \uplus \{\iota\}$, $F = \bigcup_{M\in L} F_M \cup \{\iota \mid \mathsf{I}_0 \in L\}$, and $\Delta = \bigcup_{M\in L} \Delta_M \cup \{(\iota, \mathsf{rn}(\mathsf{id}_X), \iota_M) \mid M \in L\}$. The idea is that, starting in ι, the DCA $\mathcal{A}$ chooses an MSC $M \in L$ to be executed. As states and messages are disjoint, transitions cannot be "mixed" anymore during an execution so that M is indeed executed. □

Lemma 16. *Let $L \subseteq \mathbb{M}$ be closed and let $B \geq 1$. Then, L is B-realizable iff every MSC in L is B-realizable.*

Proof. Suppose L is B-realizable. There is a DCA $\mathcal{A}$ with set of process variables X such that $|X| \leq B$ and $L(\mathcal{A}) = L$. For $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda) \in L$, there is an accepting run $(proc, state)$ of $\mathcal{A}$ on M. One can check that the mapping $\chi : E \to \mathcal{P}$ given by $\chi(e) = \{proc(e)[x] \mid x \in X\} \setminus \{loc(e)\}$ is a B-bounded valid knowledge mapping for M under $\emptyset$. By Lemma 13, M is B-realizable.

For the reverse direction, we can simply use the construction of a DCA from Lemma 15, assuming $X_M = X_N$ and $|X_M| \leq B$ for every all $M, N \in L$. □

We are interested in two decision problems for dynamic MSC grammars, addressing emptiness and realizability:

Problem 17. Emptiness for dynamic MSC grammars:
INSTANCE: Dynamic MSC grammar G.
QUESTION: Is $L(G)$ empty?

Problem 18. Realizability for dynamic MSC grammars:
INSTANCE: Dynamic MSC grammar G.
QUESTION: Is G realizable?

In the following, we will proof the following two theorems, which establish the precise complexity of the above problems:

Theorem 19. *Emptiness for dynamic MSC grammars is EXPTIME-complete.*

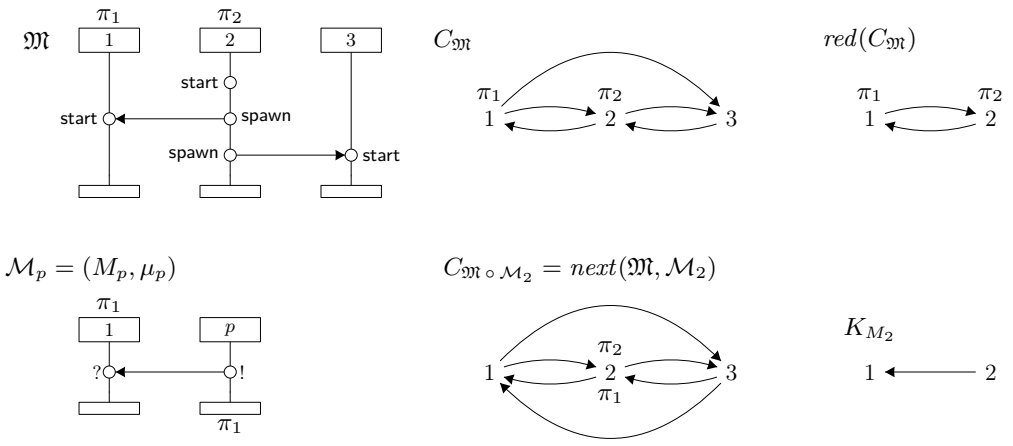

Fig. 11. Communication structures

Theorem 20. *Realizability for dynamic MSC grammars is EXPTIME-complete.*

We establish lower and upper bounds for these problems separately, in terms of several propositions.

In the proof of both theorems, we will make use of bounded abstractions of both named MSCs and migration PMSCs in terms of *communication structures*. Accordingly, a communication structure reflects either the current communication topology, comprising the currently active processes as well as a binary "is known to"-relation between processes, or the message flow that takes place within a PMSC.

Let Π be a nonempty finite set of process identifiers. A *communication structure* over Π is a tuple $C = (P, K, \ell)$ where $P \subseteq \mathbb{N}$ is a nonempty finite set of *nodes*, $K \subseteq (P \times P) \setminus \mathsf{id}_P$, and $\ell : \Pi \to P$. Intuitively, any node represents a process and $(p, q) \in K$ means that process p is known to process q, or p has communicated some information to q. The mapping ℓ associates with every process identifier a process. We call C *reduced* if $\ell(\Pi) = P$, i.e., every process carries at least one process identifier. By $red(C)$, we denote the restriction of C to $\ell(\Pi)$, which is thus a reduced communication structure. The set of communication structures over Π is denoted by $\mathcal{CS}(\Pi)$. Three communication structures are depicted in Figure 11. Hereby, the top right structure is the only reduced one.

Recall that the basic operation in a derivation of a grammar is concatenation of a named MSC and a migration PMSC. Fortunately, applicability and realizability of the PMSC only depend on (a restriction of) the communication structure associated with the named MSC.

For a PMSC $M = (\mathcal{P}, (E_p)_{p \in \mathcal{P}}, \prec, \lambda)$, we let

$$K_M = \{(loc(e), loc(\hat{e})) \mid (e, \hat{e}) \in (\prec_{\mathsf{spawn}} \cup \prec_{\mathsf{spawn}}^{-1} \cup \prec_{\mathsf{proc}} \cup \prec_{\mathsf{msg}})^+\} \setminus \mathsf{id}_{\mathcal{P}} \,.$$

Abusing notation, we may also write $K_{\mathfrak{M}}$ or $K_{\mathcal{M}}$ to denote K_M if $\mathfrak{M} = (M, \nu) \in \mathrm{n}\mathbb{M}$ or, respectively, $\mathcal{M} = (M, \mu) \in \mathrm{m}\mathbb{P}$. One can easily check that K_M can be described in terms of χ_M as follows:

Remark 21. We have $K_M = \{(p,q) \in (\mathcal{P} \times \mathcal{P}) \setminus \mathsf{id}_{\mathcal{P}} \mid p \in \chi_M(\max_q(M))\}$.

Now let $\mathfrak{M} = (M, \nu)$ be a named MSC over Π. We set $C_{\mathfrak{M}}$ to be the communication structure $(\mathit{Proc}(M), K_M, \nu)$. Again, we refer to Figure 11 for an example.

Suppose the system is in a configuration that is reflected by a communication structure $C = (P, K, \ell)$. Then, a migration PMSC $\mathcal{M} = (M, \mu)$ with $M = (\mathcal{P}, (E_p)_{p \in \mathcal{P}}, \prec, \lambda)$ is said to be *applicable* at C, written $C \vdash \mathcal{M}$, if

- $\mathit{Free}(M) \subseteq P$,
- $\mathit{Bound}(M) \cap P = \emptyset$, and
- for all $\pi \in \Pi$ such that $\mu(\pi) = (p, q)$ is defined, we have $\ell(\pi) = p$.

The intuition behind this definition is that concatenation is defined. This is reflected by the following lemma, which follows directly from the definitions.

Lemma 22. *Let $\mathfrak{M} = (N, \nu)$ be a named MSC and $\mathcal{M} = (M, \mu)$ be a migration* PMSC. *Furthermore, let $P \subseteq \mathit{Proc}(N)$ such that $\nu(\Pi) \cup (\mathit{Proc}(N) \cap \mathit{Proc}(M)) \subseteq P$. Then, $\mathfrak{M} \circ \mathcal{M}$ is defined iff $C_{\mathfrak{M}} \restriction P \vdash \mathcal{M}$.*

We refine the previous definition and say that PMSC $\mathcal{M} = (M, \mu)$ is *realizable* at $C = (P, K, \ell)$, written $C \Vdash \mathcal{M}$, if both $C \vdash \mathcal{M}$ and χ_M^K is valid. Consider Figure 11. We have $C_{\mathfrak{M}} \vdash \mathcal{M}_3$, but $C_{\mathfrak{M}} \nVdash \mathcal{M}_3$. Indeed, the MSC in $\mathfrak{M} \circ \mathcal{M}_3$ is not realizable:

Lemma 23. *Let $\mathfrak{M} = (N, \nu)$ be a named MSC and $\mathcal{M} = (M, \mu)$ be a migration* PMSC *such that $\mathfrak{M} \circ \mathcal{M}$ is defined. Moreover, let $P \subseteq \mathit{Proc}(N)$ such that $\nu(\Pi) \cup (\mathit{Proc}(N) \cap \mathit{Proc}(M)) \subseteq P$. Then, $N \circ M$ is realizable iff both N is realizable and $C_{\mathfrak{M}} \restriction P \Vdash \mathcal{M}$.*

Proof. Suppose $N \circ M$ is realizable. By Lemma 14, $\chi_{N \circ M}$ is a valid knowledge mapping and so is the restriction χ_N of $\chi_{N \circ M}$ to events from N. Thus, the MSC N is realizable. By Lemma 22, $C_{\mathfrak{M}} \restriction P \vdash \mathcal{M}$. It remains to verify that $\chi_M^{K_N \cap (P \times P)}$ is valid. But by Remark 21 and the definition of $\chi_M^{K_N}$, we obtain that $\chi_M^{K_N}$ is the restriction of $\chi_{N \circ M}$ to events of M. As $\chi_{N \circ M}$ is valid and $\mathit{Proc}(N) \cap \mathit{Proc}(M) \subseteq P$, $\chi_M^{K_N \cap (P \times P)}$ is valid, too.

For the converse direction, suppose that N is realizable and $C_{\mathfrak{M}} \restriction P \Vdash \mathcal{M}$. The former implies that χ_N is valid. By $C_{\mathfrak{M}} \restriction P \Vdash \mathcal{M}$, $\chi_M^{K_N \cap (P \times P)}$ is valid as well. Clearly, the union of χ_N and $\chi_M^{K_N \cap (P \times P)}$ is a valid knowledge mapping. □

So far, communication structures were used as abstractions of named MSCs. In the following, we show that they can also reflect migration PMSCs and their communication flow.

For relations $K_1, K_2 \subseteq (\mathbb{N} \times \mathbb{N}) \setminus \mathsf{id}_{\mathbb{N}}$, we set

$$K_1 \odot K_2 = K_1 \cup K_2 \cup (K_1 \circ K_2) \setminus \mathsf{id}_{\mathbb{N}} .$$

Note that $\odot$ is associative.

As we defined concatenation of a named MSC and a migration PMSC, we can also apply a migration PMSC to a communication structure directly. To this aim, we define a

partial mapping *next*, which takes as arguments a communication structure $C = (P, K, \ell)$ and a migration PMSC $\mathcal{M} = (M, \mu) \in \mathrm{m}\mathbb{P}$. Then, $next(C, \mathcal{M})$ is defined iff $C \vdash \mathcal{M}$, i.e., $\mathcal{M}$ is applicable at C. In that case, we set $next(C, \mathcal{M})$ to be the communication structure (P', K', ℓ') (over Π) where

- $P' = P \cup Proc(M)$,
- $K' = K \odot K_M$,
- $\ell'(\pi) = \ell(\pi)$ if $\mu(\pi)$ is undefined, and $\ell'(\pi) = q$ if $\mu(\pi) = (p, q)$ is defined.

The following lemma is an easy observation:

Lemma 24. *Let M and M' be* PMSCs *and $P \subseteq Proc(M) \cup Proc(M')$ such that $M \circ M'$ is defined and $Proc(M) \cap Proc(M') \subseteq P$. Then, the following hold:*

(a) $K_{M \circ M'} = K_M \odot K_{M'}$
(b) $K_{M \circ M'} \restriction (P \cup Proc(M)) = K_M \odot (K_{M'} \restriction P)$
(c) $K_{M \circ M'} \restriction (P \cup Proc(M')) = (K_M \restriction P) \odot K_{M'}$

We obtain, as a corollary from the previous lemma, that a communication structure is a faithful abstraction of a named MSC wrt. concatenation, as illustrated in Figure 11.

Corollary 25. Let $\mathfrak{M} = (N, \nu)$ be a named MSC and $\mathcal{M}$ be a migration PMSC such that $\mathfrak{M} \circ \mathcal{M}$ is defined. Moreover, let $P \subseteq Proc(\mathfrak{M})$ such that $\nu(\Pi) \cup (Proc(\mathfrak{M}) \cap Proc(\mathcal{M})) \subseteq P$. Then, $next(C_{\mathfrak{M}} \restriction P, \mathcal{M}) = C_{\mathfrak{M} \circ \mathcal{M}} \restriction (P \cup Proc(\mathcal{M}))$.

We are now prepared to prove Theorems 19 and 20 stating that emptiness and realizability for dynamic MSC grammars are EXPTIME-complete. We first show containment in EXPTIME (Proposition 26) and then the hardness result (Proposition 31).

Proposition 26. *Emptiness and realizability for dynamic MSC grammars are both in EXPTIME.*

Proof. Let $G = (\Pi, \mathcal{N}, S, \longrightarrow)$ be a dynamic MSC grammar. To answer the first question ($L(G) = \emptyset$?), we build a tree automaton $\mathcal{A}_G$ that accepts all parse trees of G that correspond to successful derivations of G. Thus, we have $L(\mathcal{A}_G) = \emptyset$ iff $L(G) = \emptyset$. To answer the second question (Is $L(G)$ realizable?), we build a tree automaton $\mathcal{B}_G$ for those parse trees that correspond to a successful derivation of a *non*-realizable MSC. According to Lemma 15, $L(G)$ is realizable iff all MSCs in $L(G)$ are realizable. Thus, $L(G)$ is realizable iff $L(\mathcal{B}_G) = \emptyset$.

Note that $\mathcal{B}_G$ will actually be just a slight extension of $\mathcal{A}_G$. To illustrate their idea, we use the dynamic MSC grammar G from Figure 4. The left-hand side of Figure 12 depicts the parse tree t of G that corresponds to the derivation from Figure 6. 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 (1) the current configuration of a system and (2) the flow of information in a migration PMSC. This information will be

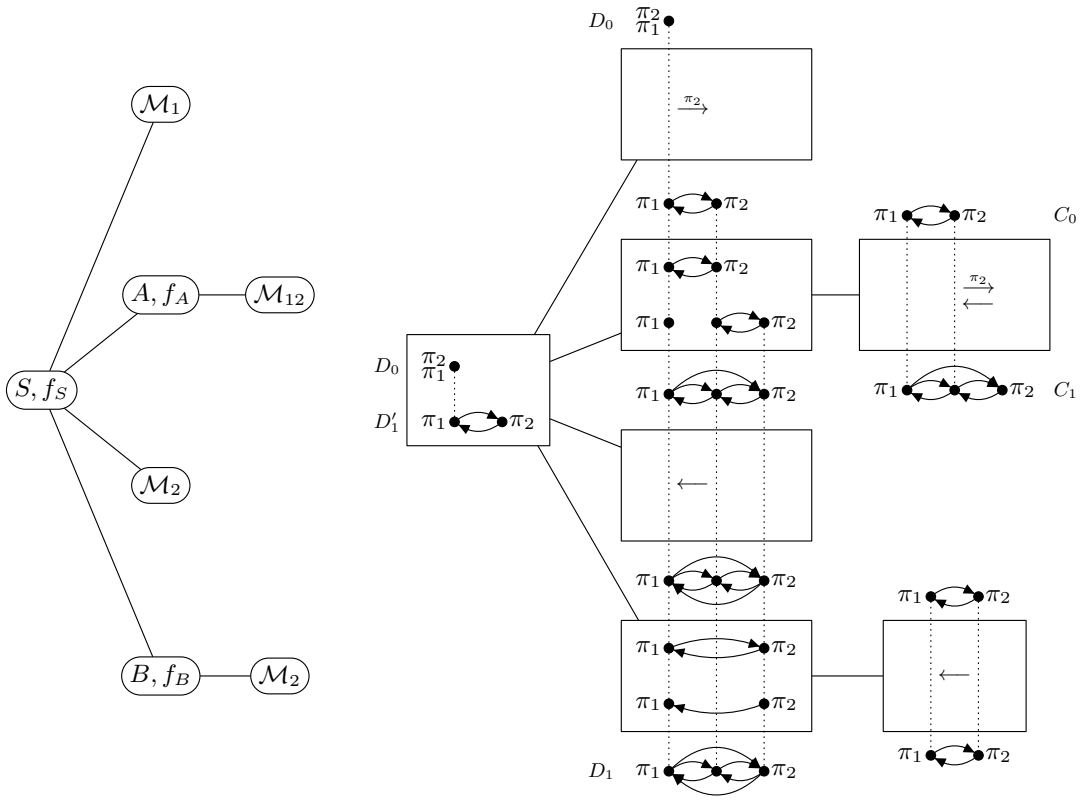

Fig. 12. A legal parse tree of G and a run of $\mathcal{A}_G/\mathcal{B}_G$

provided in terms of communication structures. The right-hand side of Figure 12 depicts a run of $\mathcal{A}_G/\mathcal{B}_G$ 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 a migration PMSC 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). Indeed, $\mathcal{M}_{12}$ is applicable at C_0, and when we apply $\mathcal{M}_{12}$, the outcome will be a new structure, $C_1 = \mathit{next}(C_0, \mathcal{M}_{12})$, 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. In the figure, 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. This transformation is uniquely described by a communication structure C_1' (the second structure in the state of node 2, with "known-to-relation" $K_{\mathcal{M}_{12}}$), which captures the information flow in $\mathcal{M}_{12}$.

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. To end up with finitely many states, 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 $\mathcal{A}_G$ will be finite, though exponential in $|G|$. Like elements of $\mathrm{m}\mathbb{P}$, a triple (C_0, A, C_1') can be applicable at (and be applied to) a communication structure. For example, the states that label the successors of the root transform D_0 into D_1, using a transformation that is described by D_1 as well. Now, we can reduce D_1 to D_1' 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_1') 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.

Let $\mathrm{m}\mathbb{P}_G$ denote the set of migration PMSCs that occur in some rule of G, and let $\mathcal{F}_G = \{(A, f) \mid A \longrightarrow_f \alpha\}$. In the following, we consider trees and tree automata over the ranked alphabet $\Omega = (\Gamma, \mathit{arity})$ with $\Gamma = \mathcal{F}_G \cup \mathrm{m}\mathbb{P}_G$ and, for $(A, f) \in \mathcal{F}_G$, $\mathit{arity}((A, f)) = \{|\alpha| \mid A \longrightarrow_f \alpha\}$ and, for $\mathcal{M} \in \mathrm{m}\mathbb{P}_G$, $\mathit{arity}(\mathcal{M}) = \{0\}$.

A *parse tree* of G is a tree $t = (\mathit{dom}_t, \mathit{val}_t)$ over Ω such that

- $\mathit{val}_t(\varepsilon) \in \mathcal{F}_G$,
- for all $u \in \mathit{dom}_t$, $\mathit{val}_t(u) \in \mathrm{m}\mathbb{P}_G$ iff u is a leaf, and
- for all $u \in \mathit{dom}_t$ with $\mathit{val}_t(u) = (A, f) \in \mathcal{F}_G$, there is an expression $\alpha = a_1 \dots a_n$ such that $A \longrightarrow_f \alpha$ is a rule, u has n successors, and, for all $i \in \{1, \dots, n\}$,
  - $\mathit{val}_t(u.i) = a_i$ if $a_i \in \mathrm{m}\mathbb{P}_G$ and
  - $\mathit{val}_t(u.i) = (a_i, f_i)$ for some f_i, otherwise.

If the parse tree t has n inner nodes, then it gives rise to a sequence $\rho_t = r_1 \dots r_n \in \longrightarrow^*$ of rules as follows. Suppose $u_1, \dots, u_n$ is the enumeration of all inner nodes according to the preorder traversal of t (e.g., $\varepsilon, 2, 21, 23, 42, 5$ is a preorder traversal). Suppose furthermore that u_i has n_i successors and that $\mathit{val}_t(u_i) = (A_i, f_i)$. Then, $r_i = (A_i, a_1 \dots a_{n_i}, f_i)$ where, for $j \in \{1, \dots, n_i\}$, $a_j = \mathit{val}_t(u_i.j)$ if $\mathit{val}_t(u_i.j) \in \mathrm{m}\mathbb{P}_G$, and $a_j = A$ if $\mathit{val}_t(u_i.j) = (A, f) \in \mathcal{F}_G$. We say that path tree t is *legal* if $\mathit{val}_t(\varepsilon) = (S, f)$ for some f and, given $\rho_t = r_1 \dots r_n$, we have that $\mathcal{C}_0 \overset{r_1}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}_G^* \dots \overset{r_n}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}_G^* ((M, \nu), \varepsilon)$ for some initial configuration $\mathcal{C}_0$, $M \in \mathbb{M}$, and ν. Note that, given t, M is uniquely determined up to isomorphism (i.e., modulo renaming of processes and events). Let us pick one MSC from that isomorphism class and denote it by $M(t)$.

The tree automaton $\mathcal{A}_G$. We will first construct $\mathcal{A}_G = (Q, \leftarrow, F)$ over Ω with

$$L(\mathcal{A}_G) = \{t \mid t \text{ is a legal parse tree of } G\}\,.$$

States assigned to leaves must be taken from $\mathrm{m}\mathbb{P}_G$. The other states, which will be associated with inner nodes, are composed of communication structures, which carry the finite amount of information needed to infer which processes are known to/communicate with other processes. A state of the tree automaton $\mathcal{A}_G$ is now either an element from $\mathrm{m}\mathbb{P}_G$ or a triple (C, A, C') where $A \in \mathcal{N}$, $C = (P, K, \ell)$ is a reduced communication

structure, and $C' = (P', K', \ell')$ is a communication structure such that $P \subseteq P' \subseteq \{1, \ldots, 2|\Pi|\}$. We set Q to be the set of all those states. The number of communication structures whose processes form a subset of $\{1, \ldots, 2|\Pi|\}$ is smaller than $2^{6|\Pi|^2+2|\Pi|}$. Hence, we have $|Q| \leq |\mathrm{m}\mathbb{P}_G| + 2^{12|\Pi|^2+4|\Pi|} \cdot |\mathcal{N}|$, which is exponential in $|G|$. We let F be the set of triples $(C, S, C') \in Q$ such that C consists of one single process (carrying all the process identifiers).

Towards the transitions of $\mathcal{A}_G$, we define relations $\rightsquigarrow, \underset{\tau}{\rightsquigarrow} \subseteq \mathcal{CS}(\Pi) \times (Q \cup \mathrm{m}\mathbb{P}) \times \mathcal{CS}(\Pi)$ (where τ is any renaming), which were illustrated in Figure 12. Let $C = (P, K, \ell)$ and $C' = (P', K', \ell')$ be communication structures and $s \in Q$.

(a) For $s = (M, \mu) \in \mathrm{m}\mathbb{P}$ and a renaming τ, we let $C \overset{s}{\underset{\tau}{\rightsquigarrow}} C'$ if $\mathit{next}(C, s\tau) = C'$ (in particular, $C \vdash s\tau$).

(b) For a renaming τ and $s = (\dot{C}, A, \mathring{C})$ with $\dot{C} = (\dot{P}, \dot{K}, \dot{\ell})$ and $\mathring{C} = (\mathring{P}, \mathring{K}, \mathring{\ell})$, we let $C \overset{s}{\underset{\tau}{\rightsquigarrow}} C'$ if (renaming is applied to communication structures and its components in the expected manner)

- $\mathit{red}(C) = \dot{C}\tau$, (1)
- $(\mathring{P}\tau \setminus \dot{P}\tau) \cap P = \emptyset$, (2)
- $P' = P \cup \mathring{P}\tau$, (3)
- $K' = K \odot \mathring{K}\tau$, and (4)
- $\ell'(\pi) = \mathring{\ell}\tau(\pi)$. (5)

We write $C \overset{s}{\rightsquigarrow} C'$ if $C \overset{s}{\underset{\tau}{\rightsquigarrow}} C'$ for some τ.

Let us construct the transitions of $\mathcal{A}_G$. For $s \in Q$, let K_s denote K_M if $s = (M, \mu) \in \mathrm{m}\mathbb{P}_G$. If $s = (C, A, (P, K, \ell))$, we set $K_s = K$.

(a) For every $\mathcal{M} \in \mathrm{m}\mathbb{P}_G$, we have a transition $\mathcal{M} \overset{\mathcal{M}}{\leftarrow} ()$.

(b) For a state $s = (\dot{C}, A, \mathring{C})$ with $\dot{C} = (\dot{P}, \dot{K}, \dot{\ell})$ and $\mathring{C} = (\mathring{P}, \mathring{K}, \mathring{\ell})$, and a rule $r = (A, \alpha, f)$ with $\alpha = a_1 \ldots a_k$ ($a_i \in \mathcal{N} \cup \mathrm{m}\mathbb{P}$), we have a transition

$$s \overset{(A,f)}{\leftarrow} (s_1, \ldots, s_k)$$

if there are renamings $\sigma, \tau_1, \ldots, \tau_k$ such that

- for all $p \in \mathit{Free}(\alpha\sigma)$, $\dot{\ell}(f\sigma(p)) = p$, (6)
- $\mathit{Bound}(\alpha\sigma) \cap \dot{P} = \emptyset$, (7)
- for all $i \in \{1, \ldots, k\}$, if $a_i \in \mathcal{N}$, then $s_i = (C_i, a_i, C_i')$ for some C_i, C_i', (8)
- for all $i \in \{1, \ldots, k\}$, if $a_i \notin \mathcal{N}$, then $s_i = a_i$ and $\tau_i = \sigma$, (9)
- $\dot{C} \overset{s_1}{\underset{\tau_1}{\rightsquigarrow}} \ldots \overset{s_k}{\underset{\tau_k}{\rightsquigarrow}} C'$ for some $C' = (P', K', \ell')$ such that (10)
  - $\mathring{P} = \ell(\pi) \cup \ell'(\pi)$,
  - $\mathring{K} = (K_{s_1}\tau_1 \odot \ldots \odot K_{s_k}\tau_k) \restriction \mathring{P}$, and
  - $\mathring{\ell} = \ell' \restriction \mathring{P}$.

We say that $(\sigma, \tau_1, \ldots, \tau_k)$ is a *witness* for $s \overset{(A,f)}{\leftarrow} (s_1, \ldots, s_k)$. The transition relation of $\mathcal{A}_G$ can be constructed in exponential time. Indeed, we have to check, for every rule (A, α, f) and every assignment of states to positions in α, whether communication structures exist that justify the assignment wrt. the relation $\rightsquigarrow$. It is easy to see that, hereby, we can restrict to communication structures (P, K, ℓ) with $P \subseteq \{1, \ldots, |\Pi| + |\alpha| m\}$ where $m = \max\{|Proc(\alpha)| \mid A \longrightarrow_f \alpha\}$. Once the communication structures have been chosen, checking if the relations $\overset{s}{\rightsquigarrow}$ hold can be done in polynomial time.

Correctness of $\mathcal{A}_G$. We will now show the correctness of our construction of $\mathcal{A}_G$, i.e., that $L(\mathcal{A}_G) = \{t \mid t \text{ is a legal parse tree of } G\}$. Clearly, $L(\mathcal{A}_G)$ contains only parse trees of G. Now, equality follows from Facts 27 and 28, which we state in the following.

Fact 27. Let t be a parse tree of G, say with $\rho_t = r_1 \ldots r_n \in \longrightarrow^+$, let $\mathfrak{M} = (N, \nu) \in \mathrm{n}\mathbb{M}$ and $\mathfrak{M}' = (N', \nu') \in \mathrm{n}\mathbb{M}$, and let $P \subseteq Proc(N)$ such that $\nu(\Pi) \subseteq P$. Consider the following statements:

1. $(\mathfrak{M}, A) \overset{r_1}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G \ldots \overset{r_n}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G (\mathfrak{M}', \varepsilon)$
2. there is a run *run* of $\mathcal{A}_G$ on t such that $C_{\mathfrak{M}} \restriction P \overset{run(\varepsilon)}{\rightsquigarrow} C_{\mathfrak{M}'} \restriction (P \cup \nu'(\Pi))$

Then, 1. implies 2.

Proof. Suppose 1. holds with

$$(\mathfrak{M}, A) \underset{\sigma}{\overset{r_1}{\Longrightarrow}}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G \ldots \overset{r_n}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G (\mathfrak{M}', \varepsilon)$$

and intermediate configurations $(\mathfrak{M}, A) = \mathcal{C}_0, \ldots, \mathcal{C}_{n_t} = (\mathfrak{M}', \varepsilon)$ (n_t being the number of nodes in t). Moreover, suppose $(\mathfrak{M}_1, \mathcal{M}_1.\alpha_1) \ldots (\mathfrak{M}_\theta, \mathcal{M}_\theta.\alpha_\theta)$ is the projection of $\mathcal{C}_0 \ldots \mathcal{C}_{n_t}$ onto those configurations whose second component starts with a migration PMSC $\mathcal{M}_i = (M_i, \mu_i)$. Note that θ is the number of leaves of t. In particular, $\alpha_\theta = \varepsilon$, $\mathfrak{M}' = \mathfrak{M} \circ \mathcal{M}_1 \circ \ldots \circ \mathcal{M}_\theta$, and, for $i \in \{1, \ldots, \theta\}$, $\mathfrak{M}_i = (N_i, \nu_i) = \mathfrak{M} \circ \mathcal{M}_1 \circ \ldots \circ \mathcal{M}_i$. Assume $r_1 = (A, \alpha, f)$ with $\alpha = a_1 \ldots a_k$, $C = C_{\mathfrak{M}} = (P, K, \ell)$, and $C' = C_{\mathfrak{M}'} = (P', K', \ell')$.

We define *run* by induction on the tree structure of t. Let $i \in \{1, \ldots, k\}$. If $a_i \in \mathrm{m}\mathbb{P}$, then we set $run(i) = a_i$. If $a_i \in \mathcal{N}$, then we suppose $run(i)$ to be given inductively. So let us define $run(\varepsilon)$. Let first $\dot{C} = (\dot{P}, \dot{K}, \dot{\ell}) = C_{\mathfrak{M}} \restriction \nu(\Pi)$ and $\mathring{C} = (\mathring{P}, \mathring{K}, \mathring{\ell})$ with

- $\mathring{P} = \nu(\Pi) \cup \nu'(\Pi)$,
- $\mathring{K} = (K_{M_1} \odot \ldots \odot K_{M_\theta}) \restriction \mathring{P}$, and
- $\mathring{\ell} = \nu'$.

Now, suppose ξ to be any renaming such that $\mathring{P}\xi \subseteq \{1, \ldots, 2|\Pi|\}$. We set $run(\varepsilon) = (\dot{C}\xi, A, \mathring{C}\xi)$. With this definition, for $\nu(\Pi) \subseteq P$, we indeed have $C_{\mathfrak{M}} \restriction P \overset{run(\varepsilon)}{\underset{\xi^{-1}}{\rightsquigarrow}} C_{\mathfrak{M}'} \restriction$

$(P \cup \nu'(\Pi))$, as one can easily verify (1)–(5). In particular, using Lemma 24, we obtain

$$\begin{aligned} K' \restriction (P \cup \nu'(\Pi)) &= K_{N'} \restriction (P \cup \nu'(\Pi)) \\ &= (K \odot K_{M_1} \odot \ldots \odot K_{M_\theta}) \restriction (P \cup \nu'(\Pi)) \\ &= K \restriction P \odot [(K_{M_1} \odot \ldots \odot K_{M_\theta}) \restriction (\nu(\Pi) \cup \nu'(\Pi))] \\ &= K \restriction P \odot \mathring{K} \\ &= K \restriction P \odot (\mathring{K}\xi)\xi^{-1} \end{aligned}$$

so that (4) is satisfied.

Next, we will verify that $run(\varepsilon) \overset{(A,f)}{\leftarrow} (run(1), \ldots, run(k))$. Let $1 \leq i_1 < \ldots < i_m \leq k$ be the indices i with $a_i \in \mathcal{N}$. Moreover, let $1 \leq \mathit{first}_1 \leq \mathit{last}_1 < \ldots < \mathit{first}_m \leq \mathit{last}_m \leq \theta$ be the indices such that $\mathcal{M}_{\mathit{first}_j}, \ldots, \mathcal{M}_{\mathit{last}_j}$ is the subsequence of $\mathcal{M}_1, \ldots, \mathcal{M}_\theta$ derived in the subtree rooted at i_j. Using Lemma 24, Corollary 25, and the induction hypothesis, we obtain:

$$\begin{aligned} C_{\mathfrak{M}} &\overset{\mathcal{M}_1}{\rightsquigarrow} \ldots \overset{\mathcal{M}_{\mathit{first}_1-1}}{\rightsquigarrow} C_{\mathfrak{M}_{\mathit{first}_1-1}} \\ &\overset{\substack{run(i_1)\\ \tau_{i_1}}}{\rightsquigarrow} C_{\mathfrak{M}_{\mathit{last}_1}} \restriction \underbrace{\mathit{Proc}(\mathfrak{M}_{\mathit{first}_1-1}) \cup \nu_{\mathit{last}_1}(\Pi)}_{R_1} \\ &\overset{\mathcal{M}_{\mathit{last}_1+1}}{\rightsquigarrow} \ldots \overset{\mathcal{M}_{\mathit{first}_2-1}}{\rightsquigarrow} C_{\mathfrak{M}_{\mathit{first}_2-1}} \restriction \underbrace{R_1 \cup \mathit{Proc}(\mathcal{M}_{\mathit{last}_1+1}) \cup \ldots \cup \mathit{Proc}(\mathcal{M}_{\mathit{first}_2-1})}_{P_2} \\ &\overset{\substack{run(i_2)\\ \tau_{i_2}}}{\rightsquigarrow} C_{\mathfrak{M}_{\mathit{last}_2}} \restriction \underbrace{P_2 \cup \nu_{\mathit{last}_2}(\Pi)}_{R_2} \\ &\quad \vdots \\ &\overset{\substack{run(i_m)\\ \tau_{i_m}}}{\rightsquigarrow} C_{\mathfrak{M}_{\mathit{last}_m}} \restriction \underbrace{P_m \cup \nu_{\mathit{last}_m}(\Pi)}_{R_m} \\ &\overset{\mathcal{M}_{\mathit{last}_m+1}}{\rightsquigarrow} \ldots \overset{\mathcal{M}_\theta}{\rightsquigarrow} C_{\mathfrak{M}_\theta} \restriction \underbrace{R_m \cup \mathit{Proc}(\mathcal{M}_{\mathit{last}_m+1}) \cup \ldots \cup \mathit{Proc}(\mathcal{M}_\theta)}_{P_{m+1}} \end{aligned}$$

for some renamings $\tau_{i_1}, \ldots, \tau_{i_m}$. For $i \in \{1, \ldots, k\}$ such that a_i is a migration PMSC, let $\tau_i = \mathsf{id}_{\mathbb{N}}$. From the above, we deduce

$$C_{\mathfrak{M}} \restriction \nu(\Pi) \overset{\substack{s_1\\ \tau_1}}{\rightsquigarrow} \ldots \overset{\substack{s_k\\ \tau_k}}{\rightsquigarrow} C_{\mathfrak{M}_\theta} \restriction (P_{m+1} \setminus (\mathit{Proc}(\mathfrak{M}) \setminus \nu(\Pi)))$$

Thus, $(\sigma \circ \xi, \tau_1 \circ \xi, \ldots, \tau_k \circ \xi)$ is a witness for $run(\varepsilon) \overset{(A,f)}{\leftarrow} (run(1), \ldots, run(k))$. $\Diamond$

The next fact will establish the reverse direction of Fact 27.

Fact 28. Let t be a parse tree of G and run be a run of $\mathcal{A}_G$ on t. Suppose $\rho_t = r_1 \ldots r_n \in \longrightarrow^+$ with $r_1 = (A, \alpha, f)$. Let $\mathfrak{M} = (N, \nu) \in \mathrm{n}\mathbb{M}$ and $P \subseteq \mathit{Proc}(N)$ such that $\nu(\Pi) \subseteq P$.

Moreover, let $C' = (P', K', \ell')$ be a communication structure such that $P' \cap (\mathit{Proc}(\mathfrak{M}) \setminus P) = \emptyset$. Consider the following statements:

1. $C_{\mathfrak{M}} \restriction P \overset{run(\varepsilon)}{\rightsquigarrow} C'$
2. $(\mathfrak{M}, A) \overset{r_1}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G \ldots \overset{r_n}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G (\mathfrak{M}', \varepsilon)$ for some $\mathfrak{M}' = (N', \nu')$ with $C' = C_{\mathfrak{M}'} \restriction (P \cup \nu'(\Pi))$

Then, 1. implies 2.

Proof. Let $C = (P, K, \ell) = C_{\mathfrak{M}} \restriction P$ and let $run(\varepsilon) = (\dot{C}, A, \mathring{C})$ where $\dot{C} = (\dot{P}, \dot{K}, \dot{\ell})$ and $\mathring{C} = (\mathring{P}, \mathring{K}, \mathring{\ell})$. Suppose τ is a renaming. We will actually show that $C_{\mathfrak{M}} \restriction P \overset{run(\varepsilon)}{\underset{\tau}{\rightsquigarrow}} C'$ implies the following stronger statement: There are a named MSC $\mathfrak{M}' = (N', \nu')$ and configurations $\mathcal{C}_0, \ldots, \mathcal{C}_{n_t}$ of G (n_t being the number of nodes of t) such that

- $C' = C_{\mathfrak{M}'} \restriction P'$ where $P' = P \cup \nu'(\Pi)$,
- $(\mathfrak{M}, A) \overset{r_1}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G \ldots \overset{r_n}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G (\mathfrak{M}', \varepsilon)$ with intermediate configurations $(\mathfrak{M}, A) = \mathcal{C}_0, \ldots, \mathcal{C}_{n_t} = (\mathfrak{M}', \varepsilon)$, and
- $\mathring{K}\tau = (K_{M_1} \odot \ldots \odot K_{M_\theta}) \restriction (\ell(\Pi) \cup \ell'(\Pi))$ where $(\mathfrak{M}_1, \mathcal{M}_1.\alpha_1) \ldots (\mathfrak{M}_\theta, \mathcal{M}_\theta.\alpha_\theta)$ is the projection of $\mathcal{C}_0 \ldots \mathcal{C}_{n_t}$ onto those configurations whose second component starts with a migration PMSC $\mathcal{M}_i = (M_i, \mu_i)$ (in particular, θ is the number of leaves in t, $\alpha_\theta = \varepsilon$, and $\mathfrak{M}' = \mathfrak{M} \circ \mathcal{M}_1 \circ \ldots \circ \mathcal{M}_\theta$).

Again, we proceed by induction on the tree structure of ρ_t. So suppose $C \overset{run(\varepsilon)}{\underset{\tau}{\rightsquigarrow}} C'$. Assume $r_1 = (A, \alpha, f)$ where $\alpha = a_1 \ldots a_k$. Then, we have that $run(\varepsilon) \overset{(A,f)}{\leftarrow} (run(1), \ldots, run(k))$ (with $run(\varepsilon) = (\dot{C}, A, \mathring{C})$) is a transition of $\mathcal{A}_G$. By definition of $\leftarrow$, there is a witness $(\sigma, \tau_1, \ldots, \tau_k)$ for that transition. Consider the renaming $\xi = \sigma \circ \tau$. Note that $\xi(\mathit{Free}(\alpha)) \subseteq \nu(\Pi)$. Due to (6), for every $p \in \mathit{Free}(\alpha\xi)$,

$$\begin{aligned} \ell(f\xi(p)) &= \ell(f(\sigma^{-1}(\tau^{-1}(p))) \\ &= \tau(\dot{\ell}(f(\sigma^{-1}(\tau^{-1}(p))))) \\ &= \tau(\tau^{-1}(p)) = p \end{aligned}$$

As we defined ξ such that $\mathit{Proc}(\mathfrak{M}) \cap \mathit{Bound}(\alpha\xi) = \emptyset$. We can deduce $(\mathfrak{M}, A) \overset{r_1}{\underset{\xi}{\Longrightarrow}}_G (\mathfrak{M}, \alpha\xi)$.

Let $1 \leq i_1 < \ldots < i_m \leq k$ be the indices i with $a_i \in \mathcal{N}$. We have

$$\dot{C} = C_0 \overset{run(1)}{\underset{\tau_1}{\rightsquigarrow}} C_1 \overset{run(2)}{\underset{\tau_2}{\rightsquigarrow}} \ldots \overset{run(k)}{\underset{\tau_k}{\rightsquigarrow}} C_k$$

for some $C_0, \ldots, C_k$. For $i \in \{0, \ldots, k\}$, we set $D_i = (P_i, K_i, \ell_i) = C_i\tau$ and, if $i \geq 1$ and $a_i \in \mathrm{m}\mathbb{P}$, $\mathcal{M}_i = a_i\xi$. We have

$$D_0 \overset{\mathcal{M}_1}{\underset{\mathsf{id}_{\mathbb{N}}}{\rightsquigarrow}} \ldots \overset{\mathcal{M}_{i_1-1}}{\underset{\mathsf{id}_{\mathbb{N}}}{\rightsquigarrow}} D_{i_1-1} \overset{run(i_1)}{\underset{\tau_{i_1} \circ \tau}{\rightsquigarrow}} D_{i_1} \overset{\mathcal{M}_{i_1+1}}{\underset{\mathsf{id}_{\mathbb{N}}}{\rightsquigarrow}} \ldots \overset{\mathcal{M}_{i_m-1}}{\underset{\mathsf{id}_{\mathbb{N}}}{\rightsquigarrow}} D_{i_m-1} \overset{run(i_m)}{\underset{\tau_{i_m} \circ \tau}{\rightsquigarrow}} D_{i_m} \overset{\mathcal{M}_{i_m+1}}{\underset{\mathsf{id}_{\mathbb{N}}}{\rightsquigarrow}} \ldots \overset{\mathcal{M}_k}{\underset{\mathsf{id}_{\mathbb{N}}}{\rightsquigarrow}} D_k$$

This implies $next(D_{j-1}, \mathcal{M}_j) = D_j$ for all $j \in \{1, \ldots, i_1 - 1\}$. By Lemmas 22 and 24, one obtains

$$(\mathfrak{M}, A) \stackrel{r_1}{\Longrightarrow}_G (\mathfrak{M}, (a_1 \ldots a_k)\xi) \stackrel{\mathrm{e}}{\Longrightarrow}^*_G (\mathfrak{M} \circ \mathcal{M}_1 \circ \ldots \circ \mathcal{M}_{i_1-1}, (a_{i_1} \ldots a_k)\xi)$$

Letting $\mathfrak{M}_1 = \mathfrak{M} \circ \mathcal{M}_1 \circ \ldots \circ \mathcal{M}_{i_1-1}$, we also have $D_{i_1-1} = C_{\mathfrak{M}_1}$ (Corollary 25). If $m = 0$ (i.e., there is no non-terminal symbol), then we are done. Otherwise, $a_{i_1} \in \mathcal{N}$. By induction hypothesis, there are configurations $\mathcal{C}^1_0, \ldots, \mathcal{C}^1_{n_1}$ of G (n_1 being the number of nodes in the subtree of t with root i_1) with projection $\mathcal{N}^1_1 \ldots \mathcal{N}^1_{h_1}$ with the appropriate properties. In particular, there is $d_1 \in \{2, \ldots, n\}$ such that

$$(\mathfrak{M}_1, (a_{i_1} \ldots a_k)\xi) \stackrel{r_2}{\Longrightarrow}_G \stackrel{\mathrm{e}}{\Longrightarrow}^*_G \ldots \stackrel{r_{d_1}}{\Longrightarrow}_G \stackrel{\mathrm{e}}{\Longrightarrow}^*_G (\mathfrak{M}_1 \circ \mathcal{N}^1_1 \circ \ldots \circ \mathcal{N}^1_{h_1}, (a_{i_1+1} \ldots a_k)\xi)$$

and $D_{i_1} = C_{\mathfrak{N}_1} \restriction P_{i_1}$ where $\mathfrak{N}_1 = \mathfrak{M}_1 \circ \mathcal{N}^1_1 \circ \ldots \circ \mathcal{N}^1_{h_1}$. Finally, $K_{run(i_1)}(\tau_{i_1} \circ \tau) = (K_{\mathcal{N}^1_1} \odot \ldots \odot K_{\mathcal{N}^1_{h_1}}) \restriction (\ell_{i_1-1}(\Pi) \cup \ell_{i_1}(\Pi))$. Continuing this scheme, we obtain

$$\begin{aligned}
(\mathfrak{M}, A) &\stackrel{r_1}{\Longrightarrow}_G \stackrel{\mathrm{e}}{\Longrightarrow}^*_G (\underbrace{\mathfrak{M} \circ \mathcal{M}_1 \circ \ldots \circ \mathcal{M}_{i_1-1}}_{\mathfrak{M}_1}, (a_{i_1} \ldots a_k)\xi) \\
&\stackrel{r_2}{\Longrightarrow}_G \stackrel{\mathrm{e}}{\Longrightarrow}^*_G \ldots \stackrel{r_{d_1}}{\Longrightarrow}_G \stackrel{\mathrm{e}}{\Longrightarrow}^*_G (\underbrace{\mathfrak{M}_1 \circ \mathcal{N}^1_1 \circ \ldots \circ \mathcal{N}^1_{h_1}}_{\mathfrak{N}_1}, (a_{i_1+1} \ldots a_k)\xi) \\
&\stackrel{\mathrm{e}}{\Longrightarrow}^*_G (\underbrace{\mathfrak{N}_1 \circ \mathcal{M}_{i_1+1} \circ \ldots \circ \mathcal{M}_{i_2-1}}_{\mathfrak{M}_2}, (a_{i_2} \ldots a_k)\xi) \\
&\vdots \\
\stackrel{r_{d_{m-1}+1}}{\Longrightarrow}_G \stackrel{\mathrm{e}}{\Longrightarrow}^*_G \ldots \stackrel{r_{d_m} = r_n}{\Longrightarrow}_G \stackrel{\mathrm{e}}{\Longrightarrow}^*_G &(\underbrace{\mathfrak{M}_m \circ \mathcal{N}^m_1 \circ \ldots \circ \mathcal{N}^m_{h_m}}_{\mathfrak{N}_m}, (a_{i_m+1} \ldots a_k)\xi) \\
&\stackrel{\mathrm{e}}{\Longrightarrow}^*_G (\underbrace{\mathfrak{N}_m \circ \mathcal{M}_{i_m+1} \circ \ldots \circ \mathcal{M}_k}_{\mathfrak{M}'}, \varepsilon)
\end{aligned}$$

and, for $j = 1, \ldots, m$, $K_{run(i_j)}(\tau_{i_j} \circ \tau) = (K_{\mathcal{N}^1_j} \odot \ldots \odot K_{\mathcal{N}^1_{h_j}}) \restriction (\ell_{i_j-1}(\Pi) \cup \ell_{i_j}(\Pi))$. Thus,

$$\begin{aligned}
\mathring{K}\tau = {} & K_{\mathcal{M}_1} \odot \ldots \odot K_{\mathcal{M}_{i_1-1}} \odot \bigodot_{j=1}^{m} (K_{run(i_j)}(\tau_{i_j} \circ \tau) \odot K_{\mathcal{M}_{i_j+1}} \odot \ldots \odot K_{\mathcal{M}_{i_{j+1}-1}}) \\
& \restriction (\ell_0(\Pi) \cup \ell_k(\Pi)) \\
= {} & K_{\mathcal{M}_1} \odot \ldots \odot K_{\mathcal{M}_{i_1-1}} \odot \bigodot_{j=1}^{m} (K_{\mathcal{N}^1_j} \odot \ldots \odot K_{\mathcal{N}^1_{h_j}} \odot K_{\mathcal{M}_{i_j+1}} \odot \ldots \odot K_{\mathcal{M}_{i_{j+1}-1}}) \\
& \restriction (\ell_0(\Pi) \cup \ell_k(\Pi))
\end{aligned}$$

where we set $i_{m+1} = k + 1$. Moreover, repeatedly applying Lemma 24 and the induction hypothesis, we deduce $D_k = C_{\mathfrak{M}'} \restriction P'$ so that we are done. $\Diamond$

Now let t be a parse tree from $L(\mathcal{A}_G)$ and run be an accepting run of $\mathcal{A}_G$ on t. Then, $run(\varepsilon)$ is of the form (C_1, S, C_2) such that C_1 contains one single process. Thus, there is an initial configuration $(\mathfrak{M}, S)$ such that $C_{\mathfrak{M}} \overset{run(\varepsilon)}{\rightsquigarrow} C'$ for some C'. By Fact 28, we conclude that t is legal. Conversely, using Fact 27, we obtain that any legal parse tree of G is accepted by $\mathcal{A}_G$.

The tree automaton $\mathcal{B}_G$. Let us build $\mathcal{B}_G = (Q', \Leftarrow, F')$ over Ω with

$$L(\mathcal{B}_G) = \{t \mid t \text{ is a legal parse tree of } G \text{ such that } M(t) \text{ is } \textit{not} \text{ realizable}\}.$$

The tree automaton $\mathcal{B}_G$ actually builds on $\mathcal{A}_G = (Q, \leftarrow, F)$ and just adds a flag construction to determine when the MSC belonging to a parse tree is *not* realizable. We set $Q' = Q \times \{0,1\}$. The second component of a state is a flag that changes to 1 once the concatenation of a communication structure and a migration PMSC is not realizable, in the sense of Lemma 23. Thus, $F' = F \times \{1\}$. We define the transitions as follows:

(a) For every $\mathcal{M} \in \mathrm{m}\mathbb{P}_G$, we have a transition $\mathcal{M} \overset{\mathcal{M}}{\Leftarrow} ()$.

(b) For a rule $r = (A, \alpha, f)$, say with $|\alpha| = k \geq 1$, states $s, s_1, \ldots, s_k \in Q$ of $\mathcal{A}_G$ with $s = (\dot{C}, A, \mathring{C})$, and flag values $b, b_1, \ldots, b_k \in \{0,1\}$, we have a transition $(s,b) \overset{(A,f)}{\Leftarrow} ((s_1,b_1), \ldots, (s_n,b_n))$ if there are renamings $\sigma, \tau_1, \ldots, \tau_k$ such that
- $s \overset{(A,f)}{\leftarrow} (s_1, \ldots, s_n)$,
- $(\sigma, \tau_1, \ldots, \tau_k)$ is a witness for $s \overset{(A,f)}{\leftarrow} (s_1, \ldots, s_n)$, and
- $b = b_1 \vee \ldots \vee b_n \vee \big[\exists\, C,\ i \in \{1, \ldots, k\} : \big(s_i \in \mathrm{m}\mathbb{P} \wedge \dot{C} \underset{\tau_1}{\overset{s_1}{\rightsquigarrow}} \ldots \underset{\tau_{i-1}}{\overset{s_{i-1}}{\rightsquigarrow}} C \not\Vdash s_i\big)\big]$.

Correctness of $\mathcal{B}_G$. Correctness of $\mathcal{B}_G$ is shown using (the proofs of) Facts 27 and 28 as well as Lemma 23.

So let $t \in L(\mathcal{B}_G)$ be a legal parse tree and $run = (run_1, run_2)$ be an accepting run of $\mathcal{B}_G$ on t. In particular, run_1 is an accepting run of $\mathcal{A}_G$ on t and $run_2 : \mathrm{dom}_t \to \{0,1\}$ with $run_2(\varepsilon) = 1$. Suppose $\rho_t = r_1 \ldots r_n \in \longrightarrow^+$. Due to Fact 28, $(\mathfrak{M}, S) \overset{r_1}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G \ldots \overset{r_n}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}^*_G (\mathfrak{M}', \varepsilon)$ for some $\mathfrak{M} = (N, \nu), \mathfrak{M}' = (N', \nu')$ such that $(\mathfrak{M}, S)$ is an initial configuration. Let $\mathcal{M}_1 = (M_1, \mu_1), \ldots, \mathcal{M}_\theta = (M_\theta, \mu_\theta)$ be the corresponding sequence of migration PMSCs, i.e., t has θ dedicated leaves, say $v_1, \ldots, v_\theta$, and $\mathfrak{M}' = \mathfrak{M} \circ \mathcal{M}_1 \circ \ldots \circ \mathcal{M}_\theta$. Note that $run_1(v_i)$ is a renaming of $\mathcal{M}_i$. Now let $u \in \mathrm{dom}_t$ be a node such that $run_2(u) = 1$ and $run_2(u_i) = 0$ for all the children $u_1, \ldots, u_k$ of u. There are a witness $(\sigma, \tau_1, \ldots, \tau_k)$ for the transition $run_1(u) \overset{(A,f)}{\leftarrow} (run_1(u_1), \ldots, run_1(u_k))$, a communication structure C, and indices $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, \theta\}$ such that $run_1(u_i) \in \mathrm{m}\mathbb{P}$, $u_i = v_j$, and $\dot{C} \underset{\tau_1}{\overset{run_1(u_1)}{\rightsquigarrow}} \ldots \underset{\tau_{i-1}}{\overset{run_1(u_{i-1})}{\rightsquigarrow}} C \not\Vdash run_1(u_i)$. Using Facts 27 and Facts 28 as well as Lemma 23, we obtain that $C \not\Vdash run_1(u_i)$ iff $(N \circ M_1 \circ \ldots \circ M_{j-1}) \circ M_j$ is realizable. We conclude that $N' \in [M(t)]$ is not realizable.

Conversely, let t be a legal parse tree of G such that $M(t)$ is not realizable. Suppose $\rho_t = r_1 \ldots r_n \in \longrightarrow^+$. We have $t \in L(\mathcal{A}_G)$, i.e., there is an accepting run run_1 of $\mathcal{A}_G$

on t. By Fact 28, $(\mathfrak{M}, S) \overset{r_1}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}_G^* \ldots \overset{r_n}{\Longrightarrow}_G \overset{\mathrm{e}}{\Longrightarrow}_G^* (\mathfrak{M}', \varepsilon)$ for some $\mathfrak{M} = (N, \nu)$ and $\mathfrak{M}' = (N', \nu')$ such that $(\mathfrak{M}, S)$ is an initial configuration. Moreover, $N' \in [M(t)]$. Again, let $\mathcal{M}_1 = (M_1, \mu_1), \ldots, \mathcal{M}_\theta = (M_\theta, \mu_\theta)$ be the corresponding sequence of migration PMSCs, i.e., t has θ dedicated leaves, say $v_1, \ldots, v_\theta$, and $\mathfrak{M}' = \mathfrak{M} \circ \mathcal{M}_1 \circ \ldots \circ \mathcal{M}_\theta$. To construct an accepting run of $\mathcal{B}_G$ on t, let $I \subseteq \{1, \ldots, \theta\}$ be the set of indices i such that $C_{\mathfrak{M} \circ \mathcal{M}_1 \circ \ldots \circ \mathcal{M}_{i-1}} \not\Vdash \mathcal{M}_i$. By Lemma 23, as N' is not realizable, I is not empty. We construct a mapping $run_2 : \mathrm{dom}_t \to \{0, 1\}$ as follows: for $u \in \mathrm{dom}_t$, let $run_2(u) = 1$ iff there are $i \in I$ and $u' \neq \varepsilon$ such that $v_i = uu'$ (i.e., u is an ancestor of v_i). One can easily verify that $run = (run_1, run_2)$ is an accepting run of $\mathcal{B}_G$ on t.

This concludes the correctness proof of $\mathcal{B}_G$.

As emptiness of $L(\mathcal{A}_G)$ and $L(\mathcal{B}_G)$ can be checked in exponential time wrt. $|G|$, we conclude that checking both emptiness and realizability for dynamic MSC grammars are in EXPTIME. □

Let us point out a crucial property of dynamic MSC grammars that we can extract from the previous proof.

Corollary 29. Let $G = (\Pi, \mathcal{N}, S, \longrightarrow)$ be a realizable dynamic MSC grammar. Then, $L(G)$ is $(|Proc(G)| + a \cdot |\Pi|)$-realizable where $a = \max\{|\alpha| \mid A \longrightarrow_f \alpha\}$.

Proof. The MSC associated with a legal parse tree is basically the concatenation of the MSCs that label the leaves, up to some renaming. The communication structure C that precedes such an MSC M (and which is actually not part of the run but uniquely determined up to renaming) is the system structure just before executing M as far as it concerns processes that can communicate with processes from M. There might be other processes at the execution time, but none of them is needed by processes from M to fulfill their communication obligations. As C contains at most $B = m + a \cdot |\Pi|$ processes, we can provide M with a B-bounded valid knowledge mapping. By Lemmas 13 and 16, $L(G)$ is B-realizable. □

Remark 30. We can also construct a tree automaton $\mathcal{B}'_G$ such that $L(\mathcal{B}'_G) = \{t \mid t \text{ is a legal parse tree of } G \text{ such that } M(t) \text{ is realizable}\}$. To obtain $\mathcal{B}'_G$, we simply take $\mathcal{A}_G$, but define, for $s = (M, \mu) \in \mathrm{m}\mathbb{P}_G$, $C \overset{s}{\rightsquigarrow} C'$ if $next(C, s) = C'$ *and* $C \Vdash s$. We have $L(\mathcal{A}_G) \setminus L(\mathcal{B}'_G) = \emptyset$ iff $L(G)$ is realizable, However, the corresponding decision procedure needs doubly exponential time.

Now let us show the hardness parts of Theorems 19 and 20.

Proposition 31. *Both problems, emptiness and realizability for dynamic MSC grammars, are EXPTIME-hard.*

Proof. We first show hardness for emptiness, by reduction from the intersection problem for tree automata. So let $\Omega = (\Gamma, arity)$ be a binary ranked alphabet, $n \geq 1$, and $\mathcal{A}_i = (Q_i, \leftarrow_i, F_i)$, $i \in \{1, \ldots, n\}$, be tree automata over Ω. Without loss of generality, we will

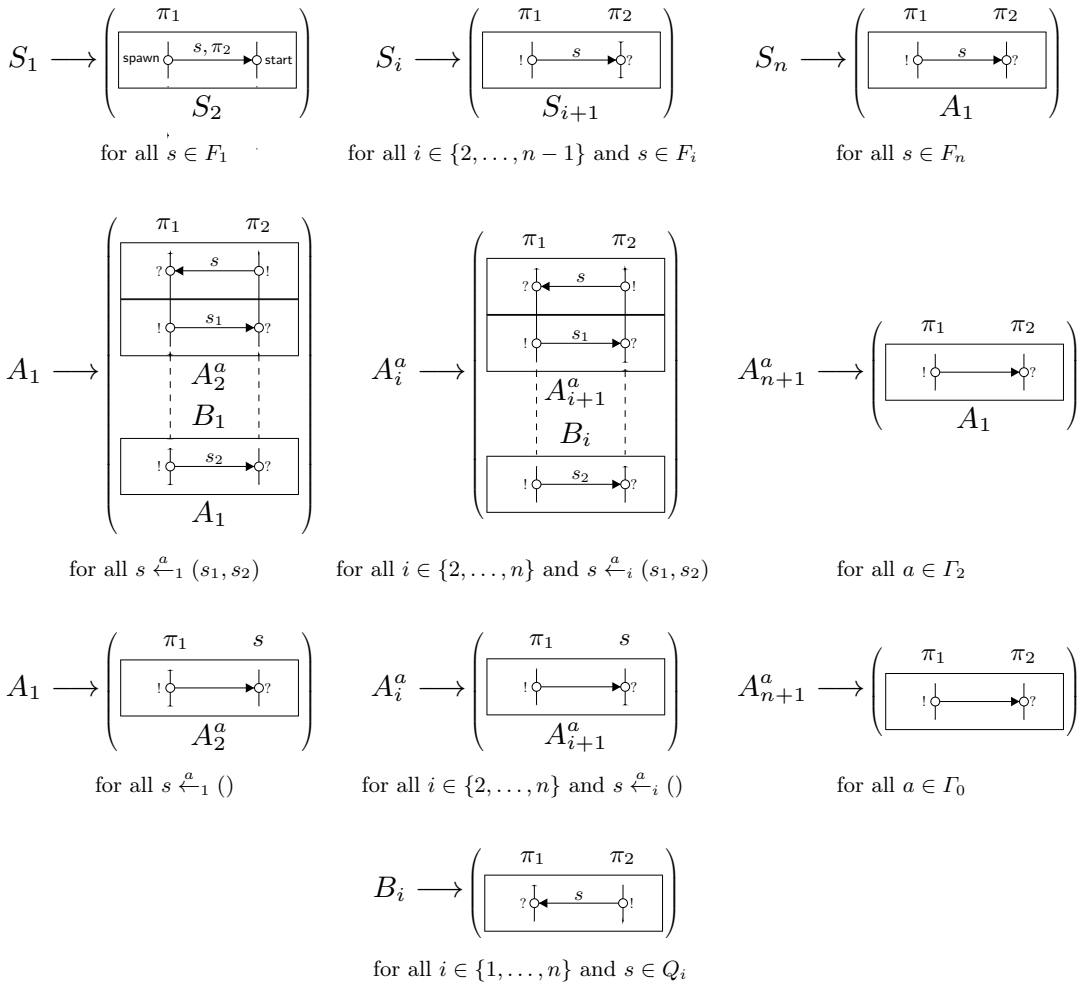

Fig. 13. Encoding of intersection for tree automata

actually make some extra assumptions. The sets Q_i are required to be pairwise disjoint. We assume $n \geq 2$ and $|arity)(a)| = 1$ for all $a \in \Gamma$. Set $\Gamma_0 = \{a \in \Gamma \mid arity(a) = \{0\}\}$ and $\Gamma_2 = \{a \in \Gamma \mid arity(a)\{2\}\}$. Finally, we assume that, for all $i \in \{1 \ldots, n\}$, $s \xleftarrow{a}_i ()$ implies $a \in \Gamma_0$, and $s \xleftarrow{a}_i (s_1, s_2)$ implies $a \in \Gamma_2$.

We construct a DGA $G = (\Pi, \mathcal{N}, S_1, \longrightarrow)$ such that $L(G) \neq \emptyset$ iff $L(\mathcal{A}_1) \cap \ldots \cap L(\mathcal{A}_n) \neq \emptyset$. First, let $\Pi = \{\pi_1, \pi_2\} \uplus \bigcup_{i \in \{1,\ldots,n\}} Q_i$. Then, $\mathcal{N} = \{S_1, \ldots, S_n\} \cup \{A_1\} \cup \bigcup_{a \in \Gamma} \{A_2^a, \ldots, A_{n+1}^a\} \cup \{B_1, \ldots, B_n\}$. The rules from $\longrightarrow$ are given by Figure 13.

The idea behind the grammar is the following. We will need only two processes, which can be accessed in terms of π_1 and π_2. The second process holds the "current" states of each tree automaton. Parse trees will represent the trees that may be accepted by the tree automata. As we work top-down, starting from the root, the first n rules are applied to select a final state for each component (first row in Figure 13). Subsequently, we enter a configuration in which the second process holds exactly one final state for each $\mathcal{A}_i$, and the first process assembles all the other states. When we apply an A_1-rule, we chose a first transition for $\mathcal{A}_1$, provided the letter at the root has arity 2. The first rule in the second row has the following meaning: We guess a transition $s \xleftarrow{a}_1 (s_1, s_2)$ of $\mathcal{A}_1$. The first migration PMSC checks that s is indeed located on the second process. In exchange, the second migration PMSC sends the state of the first successor to the second process. Further below, we will remember that the second successor has to carry state s_2; before, a B_1-rule will make sure that all states are located on the first process. Subsequently, we apply $A_2^a, \ldots, A_n^a$-rules and, in doing so, guess a-transitions for the automata $\mathcal{A}_2, \ldots, \mathcal{A}_n$, respectively. We will end up in a grammar configuration with two A_1 symbols. The first is used to continue the run at the first successor of the run tree to be simulated, the second, at the bottom of the configuration, is in charge of the second successor. Finally, in case of a transition of the form $s \xleftarrow{a}_i ()$, an A_i^a step will ensure that the current state of $\mathcal{A}_i$ is indeed s.

This shows EXPTIME-hardness of emptiness for DGA. To show hardness of realizability, we simply generate a non-realizable MSC like that from Figure 7 before we run the grammar. Then, the grammar is realizable iff it generates the empty set iff the intersection of the tree-automata languages is empty. □

Note that the hardness proof of realizability relies on that of the emptiness problem. We leave open if the result still holds for *deadlock-free* grammars, where every partial derivation can be directed to a final configuration.

7 Realizability and Finite Dynamic Communicating Automata

A realizable dynamic MSC grammar is not necessarily implementable as a finite DCA, as the behavior of a single process needs not be finite-state. We will determine a simple (but non-trivial) class of dynamic MSC grammars that are finitely realizable. To guarantee finiteness, we restrict to right-linear rules: a rule $r = A \longrightarrow_f \alpha$ is *right-linear* if α is of the form $\mathcal{M}$ or $\mathcal{M}.B$. If α is of the form $\mathcal{M}$, we also call r *terminal*. Our class is inspired by *local-choice HMSCs* as introduced in [HJ00]. Local-choice HMSCs are scenario

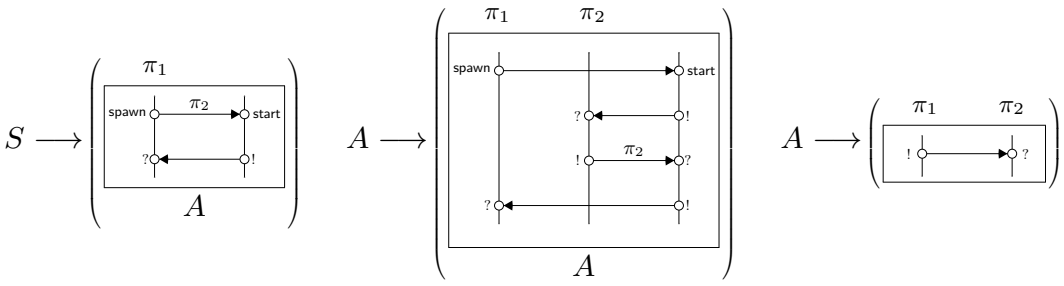

Fig. 14. A local and realizable grammar

descriptions over a fixed number of processes in which every choice of the specification 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 decision. Note that locality allows for a deadlock-free implementation if the number of processes is fixed [GMSZ06]. This is not guaranteed in our setting.

To adapt the notion of local-choice to dynamic MSC grammars, 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. For a right-linear rule $r = A \longrightarrow_f (M, \mu).\alpha$, we set $Active(r) = f(Free(M)) \cup \mathrm{dom}(\mu)$.

Definition 32. *A dynamic MSC grammar* $(\Pi, \mathcal{N}, S, \longrightarrow)$ *is* local *if, for every rule* $r = A \longrightarrow_f \alpha$*,*

- r *is right-linear,*
- *the partial order of* $M(\alpha)$ *has a unique minimal event, denoted* $first(r)$*, and*
- *there is* $\pi \in Active(r)$ *such that* $\alpha = \mathcal{M}.B$ *implies that, for all* B*-rules* $r' = B \longrightarrow_g \beta$*,* $g(loc(first(r'))) = \pi$ *(we set* $next\_leader(r) = \pi$*).*

Note that a local grammar is not necessarily realizable. The grammar from Figure 14 is both local and realizable. We can set $next\_leader(r) = \pi_1$ for all three rules r.

Theorem 33. *Both emptiness and realizability for local dynamic MSC grammars are PSPACE-complete.*

Proof. Containment in PSPACE and hardness are easily shown by slight modifications of the proofs of Propositions 26 and 31. The difference is that we construct finite word automata instead of tree automata. □

We now establish that every *realizable* local dynamic MSC grammar is actually realized by a finite DCA:

Theorem 34. *Let* G *be a* realizable *local dynamic MSC grammar. 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. Let $G = (\Pi, \mathcal{N}, S, \longrightarrow)$ be a local dynamic MSC grammar such that $L(G)$ is realizable. We assume that any two PMSCs that occur at different places in the grammar have disjoint sets of events. We will construct a DCA $\mathcal{A}_G$ such that $L(\mathcal{A}_G) = L(G)$.

A state of $\mathcal{A}_G$ will locally keep track of the progress that has been made to implement a rule. The leader process, i.e., the process that executes the first event of a PMSC, may choose the next rule and inform its communication partners about this choice. We pursue a simple strategy of transmitting process identities: When a process p sends a message to, or spawns process q, then p may communicate to q all identities it knows in terms of process variables. In turn, a receiving process may update its process variables at discretion. When a variable is used to address another process, however, one cannot be sure if it is the "right" one according to the rule that has to be simulated. Actually, a process has to guess which variable holds the correct identity. However, the subsequent execution can pass through only if that guess is correct. The reason is that both identifiers and events of processes are held in local states and are sent in messages so that communicating processes can be sure to identify each other correctly. Moreover, as the grammar is realizable, there is, for every $M \in L(G)$, at least one execution that allows $\mathcal{A}_G$ to simulate M. Note that, as G is right-linear, $L(G)$ is $(|\Pi| + |\mathit{Proc}(G)|)$-realizable so that, indeed, $|\Pi| + |\mathit{Proc}(G)|$ variables will suffice to implement $L(G)$.

Before we formally define $\mathcal{A}_G$, let us give some some useful definitions. Suppose $r = A \longrightarrow_f (M, \mu).\alpha$. For $p \in \mathit{Proc}(M)$, $q \in \mathit{Free}(M)$, and $\Lambda \subseteq \Pi$, let

$$\begin{aligned}
\mathit{Loss}_r(p) &= \{\pi \in \Pi \mid \mu(\pi) = (p, \hat{p}) \text{ for some } \hat{p} \neq p\} \\
\mathit{Gain}_r(p) &= \{\pi \in \Pi \mid \mu(\pi) = (\hat{p}, p) \text{ for some } \hat{p} \neq p\} \\
\mathit{New}_r(\Lambda, p) &= (\Lambda \setminus \mathit{Loss}_r(p)) \cup \mathit{Gain}_r(p) \\
\mathit{Req}_r(q) &= \{\pi \in \Pi \mid \mu(\pi) = (q, \hat{q}) \text{ for some } \hat{q}\} \cup \{f(q)\}
\end{aligned}$$

The meaning of the latter two is the following. When a process carries the identifiers from Λ and simulates process p in rule r, then it will carry those from $\mathit{New}_r(\Lambda, p)$ when it exits r. The set $\mathit{Req}_r(q)$ contains those identifiers that a process must carry when it enters rule r to simulate q.

A state of our automaton will be a triple $s = (r, e', \Lambda)$ where

- $r = (A, \alpha, f)$ is the rule that is currently processed,
- $e' \in \mathit{Proc}(\alpha)$ is the event that has been executed last by the process being in s, and
- $\Lambda \subseteq \Pi$ is the set of process identifiers currently held.

Suppose a process is in state $s = (r, e', \Lambda)$. Intuitively, e' has just been executed and, if there is e such that $e' = \mathit{pred}(e)$, then e is the event that has to be simulated next, i.e., the next action executed corresponds to the type of e. If, on the other hand, $e' \in \max(M(\alpha))$, then the next state of the current process depends on a case distinction. If $\mathit{next\_leader}(r) \notin \Lambda$, then the process needs to receive a message (from a leader) to be informed about the next rule, if any (see transition Rec_Next in Figure 15). Otherwise, a suitable next rule is chosen by the leader process (see transitions Spawn_Next and Send_Next).

We can now specify $\mathcal{A}_G = (X, \mathit{Msg}, Q \uplus \{\iota\}, \Delta, \iota, F)$ as follows:

- $X = \{x_\pi \mid \pi \in \Pi\} \cup \{x_p \mid p \in Proc(G)\}$,
- Msg contains all pairs $(r, (e, \hat{e}))$ where $r = (A, \alpha, f) \in \longrightarrow$ and $(e, \hat{e})$ is a message pair in $M(\alpha)$,
- Q contains all triples (r, e, Λ) as defined above, and
- F contains all $(r, e, \Lambda) \in Q$ such that $r = (A, \alpha, f)$ is a terminal rule and $e \in \max(M(\alpha))$.

The transition relation Δ is given by Figure 15. Hereby, r and r' will always stand for rules $r = A \longrightarrow_f (M, \mu).\alpha$ and, respectively $r' = A' \longrightarrow_{f'} (M', \mu').A'$. Moreover, events e and $\hat{e}$ as well as the binary relations $\prec_{\mathsf{msg}}$ and $\prec_{\mathsf{spawn}}$ refer to the PMSC $M = (\mathcal{P}, (E_p)_{p\in\mathcal{P}}, \prec, \lambda)$ that occurs in r. Finally, we say that a rule r is *initial* if $\mathcal{C} \stackrel{r}{\Longrightarrow_G} \stackrel{\mathrm{e}}{\Longrightarrow_G} \mathcal{C}'$ for some initial configuration $\mathcal{C}$ and some $\mathcal{C}'$.

Let us explain the different transition types:

Init A first rule r is chosen.

Spawn Process p spawns a process $\hat{p}$. In the spawning process, identifier x will henceforth refer to $\hat{p}$.

Send In executing e, process p sends a message to $\hat{p}$. The receiving process is identified using an arbitrary process variable $x \in X$. We assume that p knows $\hat{p}$. Thus, there is indeed a variable identifying $\hat{p}$. As p sends the event it performs along with the message, a process can receive it only if it executes the corresponding receive event.

Rec Process p receives a message from process $\hat{p}$. This includes an agreement on the current rule r and the communicating events $(\hat{e}, e)$ that are simulated. Process p has the possibility to update some of the process identifiers. The identities of the sender are correctly identified since the event performed by the sending process is contained in the message.

Spawn_Next Having executed e', process $loc(e')$, which carries the process identifiers in Λ', has fully executed its part ($e' \in \max(M')$) and faces a non-terminal symbol A. As it carries $next\_leader(r')$, it chooses r to be the next rule. For that rule, it indeed carries all the requested identifiers, i.e., $Req_r(p) \subseteq \Lambda'$. The first event of r being a spawn event, located on p, a corresponding action is executed. The new process begins in $\hat{e}$, which is a start event. Starting without any identifier, its identifier set will be $New_r(\emptyset, \hat{p})$ when it exits r. Moreover, it will be aware of an arbitrary number of identities known to the spawning process.

Send_Next This transition is very similar to the previous one. The executing process is the leader in a the new rule r and first performs a send action. In doing so, it communicates all identities that it knows to the receiving process.

Rec_Next Again, a process $loc(e')$ has fully executed its part and faces a non-terminal symbol A. However, it does not carry $next\_leader(r')$ so that it has to wait for the leader process and receive its choice in terms of a message (possibly from a process that is not the leader). When the message is received, the process can choose which of its variables shall be updated.

Renaming At any time, a process can reorder its variables.

$$\textsf{Init}\quad \frac{\begin{array}{cc} r \text{ initial} & \eta : (X \uplus \{\textsf{self}\})^X \\ \mathit{first}(r) \prec_{\textsf{spawn}} \hat{e} & a = x \leftarrow \textsf{spawn}((r, \hat{e}, \mathit{New}_r(\emptyset, \hat{p})), \eta) \end{array}}{\iota \xrightarrow{a} (r, \mathit{first}(r), \mathit{New}_r(\Pi, \mathit{loc}(\mathit{first}(r))))}$$

$$\textsf{Spawn}\quad \frac{\begin{array}{cc} & \eta : (X \uplus \{\textsf{self}\})^X \\ e' \prec_{\textsf{proc}} e \prec_{\textsf{spawn}} \hat{e} & a = x \leftarrow \textsf{spawn}((r, \hat{e}, \mathit{New}_r(\emptyset, \hat{p})), \eta) \end{array}}{(r, e', \Lambda) \xrightarrow{a} (r, e, \Lambda)}$$

$$\textsf{Send}\quad \frac{e' \prec_{\textsf{proc}} e \prec_{\textsf{msg}} \hat{e} \qquad a = x\,!\,((r, (e, \hat{e})), \mathsf{id}_X)}{(r, e', \Lambda) \xrightarrow{a} (r, e, \Lambda)}$$

$$\textsf{Rec}\quad \frac{\begin{array}{ccc} & & \Xi \subseteq X \\ e' \prec_{\textsf{proc}} e & \hat{e} \prec_{\textsf{msg}} e & a = x\,?\,((r, (\hat{e}, e)), \Xi) \end{array}}{(r, e', \Lambda) \xrightarrow{a} (r, e, \Lambda)}$$

$$\textsf{Spawn\_Next}\quad \frac{\begin{array}{cc} e = \mathit{first}(r) \prec_{\textsf{spawn}} \hat{e} & A' = A \\ e' \in \max(M') & \mathit{Req}_r(p) \subseteq \Lambda' \\ p = \mathit{loc}(e) & \mathit{next\_leader}(r') \in \Lambda' \\ \eta : (X \uplus \{\textsf{self}\})^X & a = x \leftarrow \textsf{spawn}((r, \hat{e}, \mathit{New}_r(\emptyset, \hat{p})), \eta) \end{array}}{(r', e', \Lambda') \xrightarrow{a} (r, e, \mathit{New}_r(\Lambda', p))}$$

$$\textsf{Send\_Next}\quad \frac{\begin{array}{cc} & A' = A \\ e = \mathit{first}(r) \prec_{\textsf{msg}} \hat{e} & \mathit{Req}_r(p) \subseteq \Lambda' \\ e' \in \max(M') & \mathit{next\_leader}(r') \in \Lambda' \\ p = \mathit{loc}(e) & a = x\,!\,((r, (e, \hat{e})), \mathsf{id}_X) \end{array}}{(r', e', \Lambda') \xrightarrow{a} (r, e, \mathit{New}_r(\Lambda', p))}$$

$$\textsf{Rec\_Next}\quad \frac{\begin{array}{cc} \hat{e} \prec_{\textsf{msg}} e \in \min(M) & \mathit{Req}_r(p) \subseteq \Lambda' \\ e' \in \max(M') & \mathit{next\_leader}(r') \notin \Lambda' \\ p = \mathit{loc}(e) & a = x\,?\,((\hat{r}, (\hat{e}, e)), \Xi) \qquad \Xi \subseteq X \end{array}}{(r', e', \Lambda') \xrightarrow{a} (r, e, \mathit{New}_r(\Lambda', p))}$$

$$\textsf{Renaming}\quad \frac{\sigma : X \to X}{(r, e, \Lambda) \xrightarrow{\mathsf{rn}(\sigma)} (r, e, \Lambda)}$$

Fig. 15. Transitions of $\mathcal{A}_G$

Let us show correctness of our construction, i.e., $L(G) = L(\mathcal{A}_G)$. The proof is routine once the inductive argument is grasped, which is based on the partial-order semantics of DCA.

From dynamic MSC grammar to DCA. Let (N, ν) be a named MSC and $run = (state, proc)$ be an MSC run of the DCA $\mathcal{A}_G$ on N. For an event e of N, assume that $state(e) = (r_e, event_e, \Lambda_e)$ with $r_e = (A_e, (M_e, \mu_e).\alpha_e, f_e)$. We say that run is *compatible* with (N, ν) if

- for all $e \in \max(N)$, $event_e \in \max(M_e)$ and $\Lambda_e = \nu^{-1}(loc(e))$,
- for all $(\hat{p}, p) \in K_N \restriction \nu(\Pi)$, there is $x \in X$ such that $proc(\max_p(N))[x] = \hat{p}$, and
- there is exactly one $q \in Proc(N)$ such that $next\_leader(r_{\max_q(N)}) \in \nu^{-1}(q)$ (we fix q in the following).

Now let $r = (A, (M, \mu).\alpha, f) \in \longrightarrow$ such that $f(loc(first(M))) = q$ and $A = \alpha_{\max_q(N)}$, and suppose

$$((N, \nu), A) \xRightarrow{r}_G ((N, \nu), (M\sigma, \mu).\alpha) \xRightarrow{\mathrm{e}}_G ((N', \nu'), \alpha)$$

where (N, ν) and (N', ν') are named MSCs, σ is a renaming, and $N' = N \circ M\sigma$. Let E and E' be the set of events of N and N', respectively. In particular, we assume that $E' \setminus E$ is the set of events of M. Let $(state, proc)$ be an MSC run of $\mathcal{A}_G$ on N that is compatible with (N, ν). We define a new mapping $state' : E' \to Q$ by

$$state'(e) = \begin{cases} state(e) & \text{if } e \in E \\ (r, e, New_r(\emptyset, loc(e))) & \text{if } e \notin E \text{ and } loc_M(e) \in Bound(M) \\ (r, e, New_r(\nu^{-1}(loc_{N'}(e)), loc_M(e))) & \text{if } e \notin E \text{ and } loc_M(e) \in Free(M) \end{cases}$$

As $N \circ M$ is $|X|$-realizable, there is $proc' : E' \to Proc(N')^X$ such that $(state', proc')$ is a run of $\mathcal{A}_G$ on N' and compatible with (N', ν'). If we start in an initial configuration of G, we can successively apply this construction, as its precondition is maintained as an invariant. Thus, we obtain an accepting run of $\mathcal{A}_G$ for any derivation in the grammar. We deduce $L(G) \subseteq L(\mathcal{A}_G)$.

From DCA to dynamic MSC grammar. Let $N = (\mathcal{P}, (E_p)_{p \in \mathcal{P}}, \prec, \lambda) \in L(\mathcal{A}_G)$ and $run = (state, proc)$ be an accepting MSC run of $\mathcal{A}_G$ on N. For $e \in E$, suppose $state(e) = (r_e, event_e, \Lambda_e)$ where $r_e = (A_e, (M_e, \mu_e).\alpha_e, f_e)$.

We call a configuration $((N', \nu'), A)$ of G *compatible* with run if

- $N = N' \circ M$ for some M with a unique minimal event $e_{\min}$ such that we have both $\nu'(next\_leader(r_{pred(e_{\min})})) = loc(e_{\min})$ and $A_{e_{\min}} = A$ (we fix $e_{\min}$ in the following), and
- for all $e' \in \max(N')$, both $event_{e'} \in \max(M_{e'})$ and $\Lambda_{e'} = (\nu')^{-1}(loc(e'))$.

According to the definition of $\mathcal{A}_G$, if $((N', \nu'), A)$ is compatible with run, then there is a renaming σ, with $\sigma(p) = \nu'(f_{e_{\min}}(p))$ for all $p \in Free(M_{e_{\min}})$, such that

$$((N', \nu'), A) \xRightarrow[\sigma]{r_{e_{\min}}}_G \xRightarrow{\mathrm{e}}_G ((N', \nu') \circ (M_{e_{\min}}, \mu_{e_{\min}})\sigma, \alpha_{e_{\min}}) =: \mathcal{C}$$

and either $N = N' \circ M_{e_{\min}}\sigma$, or $\mathcal{C}$ is compatible with *run*.

Now let $e \in E$ be the first spawn event of N, i.e., $start(N) \prec_{\mathsf{proc}} e$. By the definition of $\mathcal{A}_G$, we have $N = \mathsf{I}_{loc(e)} \circ M_e\sigma \circ M$ for some renaming σ and PMSC M. In particular, as r_e is initial, we have, $\mathcal{C} \xRightarrow[\sigma]{r_e}_G \xRightarrow{\mathrm{e}}_G ((N', \nu'), \alpha_e)$ for $N' = \mathsf{I}_{loc(e)} \circ M_e\sigma$ and suitable ν' and initial configuration $\mathcal{C}$. Moreover, for all $e' \in \max(N')$, $event_{e'} \in \max(M_{e'})$ and $\Lambda_{e'} = (\nu')^{-1}(loc(e'))$. In those maximal events, every process is supposed to take a _Next-transition. Thus, M has a unique minimal event so that $\alpha_e \neq \varepsilon$ and $((N', \nu'), \alpha_e)$ is compatible with *run*. Continuing this scheme, we derive a valid derivation of N in the grammar. We deduce $L(\mathcal{A}_G) \subseteq L(G)$.

This concludes the proof of Theorem 34. □

Remark 35. There are several strategies to reduce the non-determinism in $\mathcal{A}_G$ that results from guessing the correct variable to address a process. For instance, one could ensure that a variable x_p, for some bounded process p, always has the correct value in a process q when q communicates with p. Moreover, the DCA constructed from a realizable and local grammar is not necessarily deadlock-free, unlike in the case of a fixed number of processes [GMSZ06]. As future work, it remains to define a notion of a *deadlock-free* grammar meaning that every partial derivation can be brought to a successful one.

8 Related Work

Modeling of infinite systems is not a recent problem, and several computation models have been proposed to handle both the control flow of programs with recursive calls, or dynamic creation of communicating objects such as threads. One of the older models are pushdown automata, that allows for the modeling of recursion using a stack. A large literature is devoted to this model, and we refer the interested reader to [ABB97]. The recursive state machines (RSM) proposed in [ABE+05] are a related formalism. Note that, however, these formalisms do not allow for concurrency: stacks in pushdown automata model the context of a call, but there is a single control flow that is passed to the latest called procedure. Similarly, in RSMs, the control flow of an execution is confined to the latest called machine and returns to the call when this machine stops.

Dynamic hierarchical machines (DHM), proposed in [LMSPT03], are hierarchical automata that allow for dynamic activation of state machines (which can be seen as thread creation), and communication between these machines. A DHM $\mathcal{A}_1$ can send a process (another DHM $\mathcal{A}_3$) to a concurrent DHM $\mathcal{A}_2$. After this machine passing, $\mathcal{A}_3$ runs either in parallel with $\mathcal{A}_1$ and $\mathcal{A}_2$, or inside $\mathcal{A}_2$. Communications among machines are synchronous. Furthermore, a machine creation necessarily occurs during a synchronization of two transitions.

Well-formed communicating recursive state machines [BLP08] (wf-CRS) extend RSMs with concurrency. Some fork transitions in state machines allow for creation of pools of concurrent machines that communicate synchronously. When a fork occurs, the execution of the currently running machine is stopped, and the control flow is split and passed to

the newly created threads. When these threads terminate, the control flow is returned to the calling machine. Communication is synchronous and limited to concurrent threads created by the same fork.

Dynamic Petri nets [BS01] (DPN for short) are another way to model dynamic creation of processes. The elements composing DPNs are the usual transitions and places of colored Petri nets plus higher-level transitions that modify the structure of the net, by reconfiguring it or appending places and transitions. This model has the expressive power of the join calculus. However, it does not allow the message ordering needed to implement FIFO channels.

9 Conclusion and Future Work

We introduced dynamic communicating automata as a model of programs with process creation and asynchronous communication, and dynamic MSC grammars as a specification language. We solved the realizability problem and characterized a class of local dynamic MSC grammars that can be implemented by finite DCA. This is only a first step, and several issues regarding dynamic MSC grammars and DCA remain open:

One challenge is to extend the class of dynamic MSC grammars that can be automatically translated into equivalent finite DCA beyond that of right-linear specifications (and preferably without deadlock). For example, the grammar from Figure 4 is not right-linear, but realizable by a finite DCA. Similarly, when we add to the second rule of the grammar of Figure 14 a message from the third to the second process, the resulting non-right-linear grammar is still finitely realizable.

Moreover, we would like to study regular MSC languages. A nice theory of regular sets of MSCs over a fixed number of processes has been established [HMK+05] (a set of MSCs is regular if its linearization language is regular). When extending this notion to our setting, any regular set of MSCs should have an implementation in terms of a DCA.

The linearizations of a set of (dynamic) MSCs are words over an infinite alphabet. Those words have been studied in the literature as *data words* [Seg06]. In particular, there are some connections between automata and logics [BMS+06], and it would be interesting to transfer these results to our, more specialized setting. For example, it is an open question if the linearizations of channel-bounded realizable MSCs can be recognized by some data automaton with a decidable emptiness problem. This would allow us to partially decide the satisfiability problem for the two-variable logic from [BMS+06].

Indeed, we think that logics (e.g., MSO logic) may serve as an alternative specification language for DCA implementations. There have indeed been several results that nicely extend the classical (effective) expressive equivalence of MSO and finite automata to communicating automata over a fixed number of processes [BL06,GKM06,HMK+05]. Moreover, temporal logics have been studied and successfully applied for model checking [Pel00,BKM10]. An extension to dynamic MSCs could contain a freeze quantifier to store processes [DL09], as added to LTL in the context of data words. However, a straightforward extension and a corresponding translation into automata exists for none of these logics.

Last, it remains to explore connections of our formalism with well-established ones such as series-parallel languages [LW00] and the π-calculus [Mil99]. Notably, a chart semantics has been defined for the latter [BGP08], which might help to apply our automata model and proof techniques to other specification languages. In [Mey08,WZH10], a notion similar to communication structures is used to identify *depth-bounded* π-calculus specifications, which allow for decidable verification problems. It would be interesting to see if those techniques can be exploited in our setting.

References

ABB97. J. Autebert, J. Berstel, and L Boasson. Context-free languages and pushdown automata. In *Handbook of Formal Languages*, volume 1, pages 111–174. Springer, 1997.

ABE+05. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, Thomas W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

AEY05. R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

AMKN05. B. Adsul, M. Mukund, K. Narayan Kumar, and Vasumat Narayanan. Causal closure for MSC languages. In *Proceedings of FSTTCS'05*, volume 3821 of *Lecture Notes in Computer Science*, pages 335–347. Springer, 2005.

Arm07. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

BGP08. J. Borgström, A. D. Gordon, and A. Phillips. A chart semantics for the pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 194(2):3–29, 2008.

BH10. B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. In *Proceedings of CSR'10*, volume 6072 of *Lecture Notes in Computer Science*, pages 48–59. Springer, 2010.

BKM10. B. Bollig, D. Kuske, and I. Meinecke. Propositional dynamic logic for message-passing systems. *Logical Methods in Computer Science*, 2010. To appear.

BL06. B. Bollig and M. Leucker. Message-passing automata are expressively equivalent to EMSO logic. *Theoretical Computer Science*, 358(2-3):150–172, 2006.

BLP08. 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.

BMS+06. M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *Proceedings of LICS'06*, pages 7–16. IEEE Computer Society, 2006.

BS01. M. G. Buscemi and V. Sassone. High-level petri nets as type theories in the join calculus. In *Proceedings of FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 104–120. Springer-Verlag, 2001.

BZ83. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2), 1983.

DL09. S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*, 10(3), 2009.

GGH+09. T. Gazagnaire, B. Genest, L. Hélouët, P. S. Thiagarajan, and S. Yang. Causal message sequence charts. *Theoretical Computer Science*, 410(41):4094–4110, 2009.

GKM06. B. Genest, D. Kuske, and A. Muscholl. A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Information and Computation*, 204(6):920–956, 2006.

GMP03. E.L. Gunter, A. Muscholl, and D. Peled. Compositional message sequence charts. *STTT*, 5(1):78–89, 2003.

GMSZ06. 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.

HJ00. L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from HMSCs. In *Proceedings of FMICS'00*, pages 203–224. Springer, 2000.

HMK+05. 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.

Koz77. D. Kozen. Lower bounds for natural proof systems. In *SFCS'77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 254–266, 1977.

LMM02. M. Leucker, P. Madhusudan, and S. Mukhopadhyay. Dynamic message sequence charts. In *Proceedings of FSTTCS'02*, volume 2556 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 2002.

LMSPT03. R. Lanotte, A. Maggiolo-Schettini, A. Peron, and S. Tini. Dynamic hierarchical machines. *Fundam. Inform.*, 54(2-3):237–252, 2003.

Loh03. M. Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theoretical Computer Science*, 309(1-3):529–554, 2003.

LW00. K. Lodaya and P. Weil. Series-parallel languages and the bounded-width property. *Theoretical Computer Science*, 237(2):347–380, 2000.

Mey08. R. Meyer. On boundedness in depth in the π-calculus. In *Proceedings of IFIP TCS'08*, volume 273 of *IFIP*, pages 477–489. Springer-Verlag, 2008.

Mil99. R. Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, New York, NY, USA, 1999.

Pel00. D. Peled. Specification and verification of message sequence charts. In *Proceedings of FORTE/PSTV'00*, volume 183 of *IFIP Conference Proceedings*, pages 139–154. Kluwer, 2000.

RGG96. E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on message sequence charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.

Seg06. Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In *Proceedings of CSL'06*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2006.

Sei94. H. Seidl. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

WZH10. T. Wies, D. Zufferey, and T. A. Henzinger. Forward analysis of depth-bounded processes. In *Proceedings of FOSSACS'10*, volume 6014 of *Lecture Notes in Computer Science*, pages 94–108, 2010.