Dynamic Communicating Automata and Branching High-Level MSCs

Benedikt Bollig[1,⋆], Aiswarya Cyriac[1,⋆], Loïc Hélouët[2],
Ahmet Kara[3,⋆⋆], and Thomas Schwentick[3,⋆⋆]

[1] LSV, ENS Cachan, CNRS & INRIA, France
[2] INRIA/IRISA Rennes, France
[3] Lehrstuhl Informatik 1, TU Dortmund, Germany

Abstract. We study dynamic communicating automata (DCA), an extension of classical communicating finite-state machines that allows for dynamic creation of processes. The behavior of a DCA can be described as a set of message sequence charts (MSCs). While DCA serve as a model of an implementation, we propose branching high-level MSCs (bHMSCs) on the specification side. Our focus is on the implementability problem: given a bHMSC, can one construct an equivalent DCA? As this problem is undecidable, we introduce the notion of executability, a decidable necessary criterion for implementability. We show that executability of bHMSCs is EXPTIME-complete. We then identify a class of bHMSCs for which executability effectively implies implementability.

1 Introduction

Communicating automata (CA) [7] are a popular model of boolean concurrent programs, in which a fixed finite number of finite-state processes exchange messages through unbounded FIFO channels. One particular research branch considers a semantics of CA in terms of message sequence charts (MSCs). MSCs propose a visual representation of system executions, can be composed by formalisms like high-level MSCs (HMSCs), and are standardized by the ITU [13]. A natural question in this context is the implementability problem, which asks if a given HMSC can be translated into an equivalent CA [11,1,12,20,10,17,9].

Most previous formal approaches to communicating systems and MSCs restrict to a *fixed finite* set of processes. This limits their applicability, as, nowadays, many applications are designed for an open world, where the participating actors are not entirely known in advance. Example domains include mobile computing and ad-hoc networks. In [4], dynamic communicating automata (DCA) were introduced as a model of programs with process creation. In a DCA, a process may (i) send and receive messages, or (ii) spawn a new process which is equipped with a unique process identifier (pid). Pids can be stored in registers and be exchanged through messages. The use of registers in DCA suggests close

⋆ Supported by DIGITEO LoCoReP and LIA InForMel.
⋆⋆ We acknowledge the financial support by the German DFG, grant SCHW 678/4-1.

A.-H. Dediu, C. Martín-Vide, and B. Truthe (Eds.): LATA 2013, LNCS 7810, pp. 177–189, 2013.

connections with register automata (also known as finite-memory automata) and formal languages over infinite alphabets (cf. [21] for an overview).

DCA are inherently hard to analyze and to synthesize. To facilitate the specification of dynamic systems, we introduce branching HMSCs (bHMSCs). Just like DCA generalize CA, bHMSCs extend HMSCs. They are based on branching automata [15,16], which rely on a natural principle of distributed computing: a process can start a number of parallel subprocesses and resume its activity once these subprocesses terminate. Each subprocess may start some subclients so that the number of processes running in parallel is a priori not bounded. Like DCA, bHMSCs use finitely many registers to store pids. In a sense, bHMSCs combine branching automata and register automata.

In this paper, we study the implementability question: given a bHMSC, is there an equivalent DCA? This question is undecidable already in the case of a bounded number of processes [12]. Therefore, we consider the notion of executability, a necessary condition for implementability, which amounts to the question if, in every scenario, communicating processes may know each other at the time of communication. We prove executability of bHMSCs to be EXPTIME-complete. Moreover, we identify the fragment of guarded join-free bHMSCs, for which executability and implementability coincide. In this case we also provide an exponential construction of an equivalent DCA.

Related Work. A first step towards MSCs over an evolving set of processes was made in [14], where MSO model checking is shown decidable for *fork-and-join MSC grammars.* Branching HMSCs are similar to these grammars, but take into account pids as message contents and distinguish messages and process creation. Moreover, (implementable) subclasses can be identified more easily. Nevertheless, several of our results apply to the formalism from [14] once the latter is adjusted to our setting. In [5], an MSC semantics was given for the π-calculus. Note that the problems studied in [14] and [5] are very different from ours and do not distinguish between a specification and an implementation.

The present paper supersedes [4] in several aspects. Branching HMSCs are more expressive than the previous formalism, simpler to understand, and more adequate, since they are based on a natural, well-established extension of finite automata to parallelism. Moreover, we extend DCA in such a way that messages themselves can carry (visible) process identifiers. This aspect is important and frequently used (e.g., in the leader election protocol). Finally, we provide tight complexity bounds for the executability problem and solve the implementability problem for a class of specifications that cannot be handled by [4].

Other formalisms with dynamic process creation (not necessarily involving message passing) can be found, for example, in [8,18,6,2]. However, these papers consider neither an MSC based semantics nor implementability aspects.

Outline. In Section 2, we define MSCs. Branching HMSCs and DCA are presented in Sections 3 and 4, respectively. In Section 5, we study executability. Section 6 identifies a fragment of bHMSCs for which executability and implementability coincide. We conclude in Section 7. Proofs can be found in [3].

2 Dynamic Message Sequence Charts

For sets A and B, let $[A \rightharpoonup B]$ denote the set of partial mappings from A to B. We identify $f \in [A \rightharpoonup B]$ with the set $\{a \mapsto f(a) \mid a \in \mathrm{dom}(f)\}$. A *ranked alphabet* is a nonempty finite set A where every letter $a \in A$ has an arity $\mathit{arity}(a) \in \mathbb{N}$.

Let P be a set of *process names* (or, simply, *processes*). Later, P will be instantiated either by the infinite set $\mathbb{P} = \{0, 1, 2, \ldots\}$ of *process identifiers* (pids, for short), or by a finite set of registers. We fix a ranked alphabet A of *message labels*. The set of *messages* (over P) is defined as $A(P) \overset{\text{def}}{=} \{a(p_1, \ldots, p_n) \mid a \in A$, $n = \mathit{arity}(a)$, and $p_1, \ldots, p_n \in P\}$.

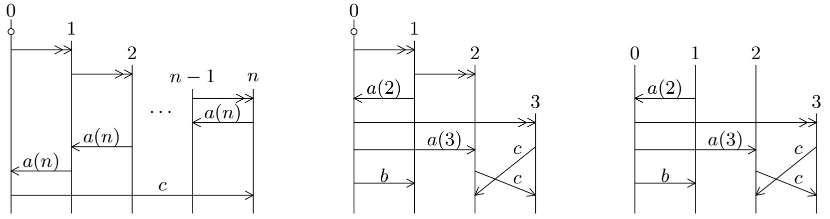
A message sequence chart (MSC) consists of a number of processes. Each process $p \in P$ is represented by a set of events E_p, totally ordered by a direct-successor relation $\lhd_{\mathsf{proc}}$. Every event has a *type* from $\mathcal{T} = \{\mathsf{start}, \mathsf{spawn}, !, ?\}$. The minimal event of a process has type start. Subsequent events can then execute spawn (spawn), send (!), or receive (?) actions. The relation $\lhd_{\mathsf{msg}}$ associates each send event with a unique receive event which is always on a different process. The exchange of messages between two processes has to conform with a FIFO policy. Similarly, $\lhd_{\mathsf{spawn}}$ relates a spawn event $e \in E_p$ with the unique start event of a different process $q \neq p$, meaning that p has created q.

Definition 1 (MSC). *A* message sequence chart (MSC) *over A and P is a tuple $M = (E, \lhd, \lambda, \mu)$ where E is a nonempty finite set of* events*, $\lhd$ is the edge relation, which is partitioned into $\lhd_{\mathsf{proc}} \uplus \lhd_{\mathsf{spawn}} \uplus \lhd_{\mathsf{msg}}$, the mapping $\lambda : E \to \mathcal{T} \times P$ assigns a type and a process to each event, and $\mu : \lhd_{\mathsf{msg}} \to A(P)$ labels a message edge with a message. For each type $\theta \in \mathcal{T}$, we let $E_\theta \overset{\mathit{def}}{=} \{e \in E \mid \lambda(e) \in \{\theta\} \times P\}$. We define the mapping $\mathit{pid} : E \to P$ such that $\mathit{pid}(e) = p$ if $\lambda(e) \in \mathcal{T} \times \{p\}$. Accordingly, for $p \in P$, set $E_p \overset{\mathit{def}}{=} \{e \in E \mid \mathit{pid}(e) = p\}$. We require the following:*

1. *$(E, \lhd^*)$ is a partial order with a unique minimal element $\mathit{init}(M) \in E_{\mathsf{start}}$,*
2. *$\lhd_{\mathsf{proc}} \subseteq \bigcup_{p \in P}(E_p \times E_p)$ and, for each $p \in P$, $\lhd_{\mathsf{proc}} \cap (E_p \times E_p)$ is the direct-successor relation of some total order on E_p,*
3. *$E_{\mathsf{start}} = \{e \in E \mid$ there is no $e' \in E$ such that $e' \lhd_{\mathsf{proc}} e\}$,*
4. *$\lhd_{\mathsf{spawn}}$ and $\lhd_{\mathsf{msg}}$ are subsets of $\bigcup_{p,q \in P \mid p \neq q}(E_p \times E_q)$,*
5. *$\lhd_{\mathsf{spawn}}$ induces a bijection between E_{spawn} and $E_{\mathsf{start}} \setminus \{\mathit{init}(M)\}$,*
6. *$\lhd_{\mathsf{msg}}$ induces a bijection between $E_!$ and $E_?$ satisfying the following (FIFO): for $e_1, e_2 \in E_p$ and $f_1, f_2 \in E_q$ with $e_1 \lhd_{\mathsf{msg}} f_1$ and $e_2 \lhd_{\mathsf{msg}} f_2$, we have $e_1 \lhd^*_{\mathsf{proc}} e_2$ iff $f_1 \lhd^*_{\mathsf{proc}} f_2$.*

The set of MSCs over A and P is denoted by $\mathbb{MSC}(A, P)$.

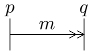
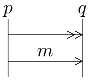
MSCs enjoy a natural graphical representation. Figure 1 depicts the MSCs $M(n)$ and M_0 over $A = \{a, b, c\}$ and $\mathbb{P}$, where $\mathit{arity}(a) = 1$ and $\mathit{arity}(b) = \mathit{arity}(c) = 0$. The events are the endpoints of arrows. Each arrow is either an element of $\lhd_{\mathsf{spawn}}$ (those with two arrow heads) or an element of $\lhd_{\mathsf{msg}}$ (those with one arrow head and a label from $A(\mathbb{P})$). The relation $\lhd_{\mathsf{proc}}$ orders (top-down) two consecutive points located on the same process line. Event $\mathit{init}(M)$, which is located on the process with pid 0, is depicted as a small circle.



$M(n)$ M_0 M_0'

Fig. 1. Two MSCs and a partial MSC

We do not distinguish MSCs that differ only in their event names. We say that two MSCs over A and $\mathbb{P}$ are *equivalent* if one can be obtained from the other by a renaming of pids. The equivalence class of M is denoted $[M]$. Moreover, for a set L of MSCs, we let $[L] = \bigcup_{M \in L}[M]$. We say that L is *closed* if $L = [L]$.

Depending on the application, a spawn in an MSC may have different interpretations, such as create subprocess, contact server, etc. In some cases, one may therefore wish to communicate a message to the new process. This can be simulated in our framework by a message edge that immediately follows a spawn. For a message m, we will actually use  as an abbreviation for .

3 Branching High-Level Message Sequence Charts

In this section, we propose a generalization of HMSCs that is suited to our dynamic setting. It is inspired by branching automata over series-parallel pomsets [15,16]. 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 2 (partial MSC). *Let $M = (E, \lhd, \lambda, \mu) \in \mathbb{MSC}(A, P)$ and let $E' \subseteq E$ be a nonempty upward-closed set containing only* complete *messages and spawning pairs: for all $(e, f) \in \lhd^* \cup \lhd_{\mathsf{msg}}^{-1} \cup \lhd_{\mathsf{spawn}}^{-1}$, we have that $e \in E'$ implies $f \in E'$. Then, the restriction of M to E' is called a* partial MSC *over A and P. The set of partial MSCs is denoted by* $\mathrm{p}\mathbb{MSC}(A, P)$.

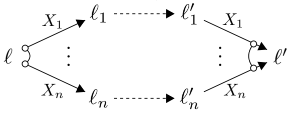
In Figure 1, M_0' is a partial MSC that is not an MSC. Notations such as $pid(e)$ carry over from MSCs to partial MSCs as expected. Let $M = (E, \lhd, \lambda, \mu) \in \mathrm{p}\mathbb{MSC}(A, P)$ be a partial MSC. By $MsgPar(M)$, we denote the set of $p \in P$ that occur as parameters in messages, i.e., those p, for which there is $a(p_1, \ldots, p_n) \in \mu(\lhd_{\mathsf{msg}})$ with $p \in \{p_1, \ldots, p_n\}$. For every $p \in P$ with $E_p \neq \emptyset$, there are a unique minimal and a unique maximal event in the total order $(E_p, \lhd^* \cap (E_p \times E_p))$, which we denote by $\min_p(M)$ and $\max_p(M)$, respectively.

We let $\mathit{Pids}(M) \overset{\text{def}}{=} \{p \in P \mid E_p \neq \emptyset\}$. By $\mathit{Free}(M) \overset{\text{def}}{=} \{p \in \mathit{Pids}(M) \mid E_{\mathsf{start}} \cap E_p = \emptyset\}$, we denote the set of *free* processes of M. Intuitively, free processes of a partial MSC M are processes that are not initiated in M. Moreover, $\mathit{Bnd}(M) \overset{\text{def}}{=} \mathit{Pids}(M) \setminus \mathit{Free}(M)$ denotes the set of *bound* processes. In Figure 1, we have $\mathit{Bnd}(M_0') = \{3\}$ and $\mathit{Free}(M_0') = \{0, 1, 2\}$.

Let $M = (E, \lhd, \lambda, \mu)$ and $M' = (E', \lhd', \lambda', \mu')$ be partial MSCs over A and P. The *concatenation* $M \circ M'$ glues identical processes together. It is defined if (i) $\mathit{Bnd}(M') \cap \mathit{Pids}(M) = \emptyset$, (ii) $\mathit{Free}(M') \neq \emptyset$, and (iii) $\mathit{Free}(M) = \emptyset$ implies $\mathit{Free}(M') \subseteq \mathit{Pids}(M)$. In that case, $M \circ M' \overset{\text{def}}{=} (\hat{E}, \hat{\lhd}, \hat{\lambda}, \hat{\mu})$ where $\hat{E} = E \uplus E'$, $\hat{\lhd}_{\mathsf{proc}} = \lhd_{\mathsf{proc}} \cup \lhd'_{\mathsf{proc}} \cup \{(\max_p(M), \min_p(M')) \mid p \in \mathit{Pids}(M) \cap \mathit{Pids}(M')\}$, $\hat{\lhd}_{\mathsf{msg}} = \lhd_{\mathsf{msg}} \cup \lhd'_{\mathsf{msg}}$, $\hat{\lhd}_{\mathsf{spawn}} = \lhd_{\mathsf{spawn}} \cup \lhd'_{\mathsf{spawn}}$, $\hat{\lambda} = \lambda \cup \lambda'$, and $\hat{\mu} = \mu \cup \mu'$.

Next we define a formalism to describe sets of MSCs. This is analogous to branching automata, but the transitions are labelled with partial MSCs.

Definition 3 (bHMSC). *A* branching high-level MSC (bHMSC) *over the set of message labels A is a tuple $\mathcal{H} = (L, X, L_{\text{init}}, L_{\text{acc}}, x_0, T)$ where L is the finite set of* locations*, $L_{\text{init}} \subseteq L$ is the set of* initial locations*, $L_{\text{acc}} \subseteq L$ is the set of* accepting locations*, X is the finite set of* registers *with* initial register $x_0 \in X$*, and T is the finite set of* transitions*. There are two types of transitions:*

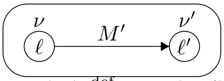
- *A sequential transition is a triple $(\ell, M, \ell') \in L \times \mathrm{p}\mathbb{MSC}(A, X) \times L$, usually written $\ell \xrightarrow{M} \ell'$, such that $\mathit{Free}(M) \neq \emptyset$ and $\mathit{MsgPar}(M) \cap \mathit{Bnd}(M) = \emptyset$ (the latter guarantees an unambigous interpretation of message parameters).*
- *A fork-and-join transition is of the form $\ell \to \{(\ell_1, X_1, \ell_1'), \ldots, (\ell_n, X_n, \ell_n')\} \to \ell'$, where $n \geq 1$ is the* degree *of the transition, $\ell, \ell_1, \ldots, \ell_n, \ell_1', \ldots, \ell_n', \ell'$ are locations from L, and $X_1, \ldots, X_n$ are nonempty and pairwise disjoint subsets of X. It may also be depicted as* 

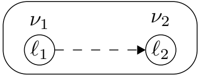
Fork-and-join transitions are similar to the split operator in [14]. At location ℓ, n subcomputations are started in $\ell_1, \ldots, \ell_n$, respectively, keeping only the register contents (pids) from $X_1, \ldots, X_n$. The other register contents are inaccessible until each subcomputaion i terminates at ℓ_i' (the registers as such may be used, but not their contents at ℓ). Then, the main computation resumes in ℓ', and registers in X_i adopt the final assignment from the i-th subcomputation.

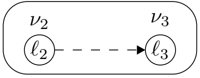
We associate MSCs with a bHMSC through the notion of runs, which we will define next after some preparation. A partial mapping $\nu : X \rightharpoonup \mathbb{P}$ is a *register assignment* if it is injective. The set of register assignments is denoted by $\mathcal{R}(X)$. For $\nu \in \mathcal{R}(X)$ and $Y \subseteq X$, we let $\nu_{\restriction Y} \overset{\text{def}}{=} \{x \mapsto \nu(x) \mid x \in \mathrm{dom}(\nu) \cap Y\}$. Given $\nu, \nu' \in \mathcal{R}(X)$ and an $M \in \mathrm{p}\mathbb{MSC}(A, X)$ that occurs in $\mathcal{H}$, we write $\nu \xrightarrow{M} \nu'$ (to be read as: M can be instantiated and performed at ν and yields ν') if

- $\mathit{Free}(M) \cup \mathit{MsgPar}(M) \subseteq \mathrm{dom}(\nu)$ (i.e., free processes can be instantiated),
- $\mathrm{dom}(\nu') = \mathrm{dom}(\nu) \cup \mathit{Bnd}(M)$, and ν and ν' coincide on $X \setminus \mathit{Bnd}(M)$ (i.e., registers remain unchanged unless they are overwritten for a new process),
- $\nu'(\mathit{Bnd}(M)) \cap \nu(X) = \emptyset$ (i.e., bound processes obtain fresh pids).

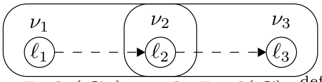
A run $G = (V, R, loc, reg, \rho)$ of the bHMSC $\mathcal{H}$ consists of a finite directed acyclic graph (V, R), $R \subseteq V \times V$, with a unique source node $in(G)$, a unique sink node $out(G)$, and labeling functions $loc : V \to L$, $reg : V \to \mathcal{R}(X)$, and $\rho : R \to 2^X \cup \mathrm{p}\mathbb{MSC}(A, \mathbb{P})$. The set of runs of $\mathcal{H}$ is defined inductively as follows:

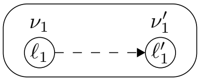
- Let $\nu, \nu' \in \mathcal{R}(X)$ be register assignments and let $\ell \xrightarrow{M} \ell'$ be a sequential transition such that $\nu \xrightarrow{M} \nu'$. Set $M' = \nu'(M)$, which we obtain from M by uniformly replacing x with $\nu'(x)$. Then, the graph $G =$ 

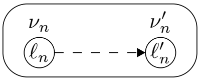
is a run of $\mathcal{H}$. We set $Pids(G) \overset{\text{def}}{=} \nu(X) \cup Pids(M')$ and $Bnd(G) \overset{\text{def}}{=} Bnd(M')$.
- Consider runs $G_1 =$ 

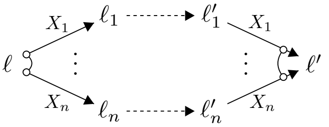
and $G_2 =$ 

of $\mathcal{H}$.

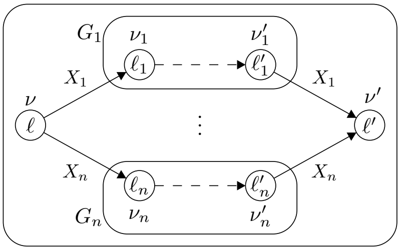
  If $Pids(G_1) \cap Bnd(G_2) = \emptyset$, then the graph $G =$ 

is a run of $\mathcal{H}$. We set $Pids(G) \overset{\text{def}}{=} Pids(G_1) \cup Pids(G_2)$ and $Bnd(G) \overset{\text{def}}{=} Bnd(G_1) \cup Bnd(G_2)$.
- For $n \geq 1$, let $G_1 =$ 

$, \ldots, G_n =$ 

be runs, 

be a fork-and-join transition, and $\nu, \nu' \in \mathcal{R}(X)$ be register assignments. Then, the graph

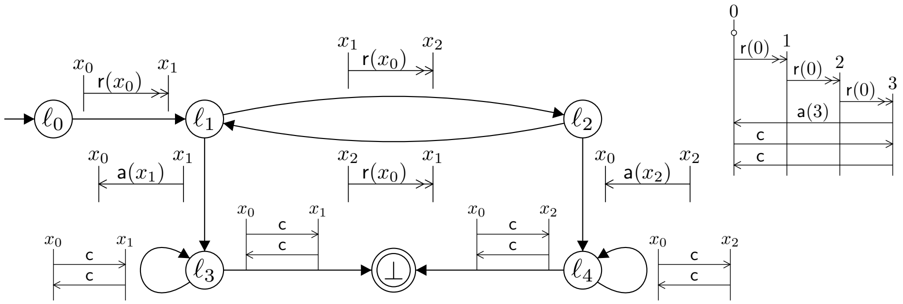
  $G =$ 

  is a run of $\mathcal{H}$ if $Bnd(G_i) \cap (\nu(X) \cup \bigcup_{j \neq i} Pids(G_j)) = \emptyset$ and $\nu_i = \nu_{\restriction X_i}$ for all $i \in \{1, \ldots, n\}$, and $\nu' = \nu_{\restriction X_0} \cup \bigcup_{i \in \{1,\ldots,n\}} (\nu_i')_{\restriction X_i}$ where $X_0 = X \setminus (X_1 \cup \ldots \cup X_n)$. We set $Pids(G) \overset{\text{def}}{=} \nu(X) \cup \bigcup_{i \in \{1,\ldots,n\}} Pids(G_i)$ and $Bnd(G) \overset{\text{def}}{=} \bigcup_{i \in \{1,\ldots,n\}} Bnd(G_i)$.

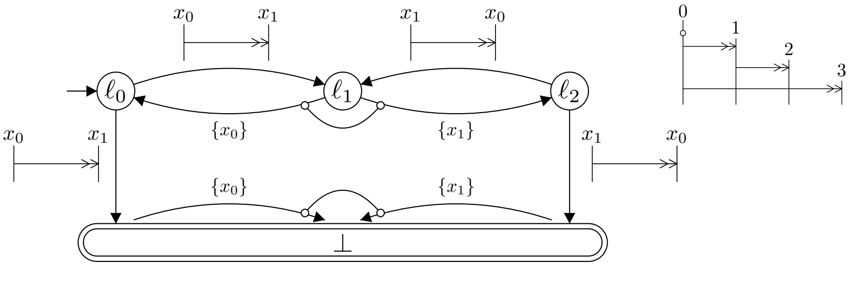
By choosing any enumeration $M_1, \ldots, M_n \in \mathrm{p}\mathbb{MSC}(A, \mathbb{P})$ of the partial MSCs occurring in G that respects the partial order induced by the edge relation R, we define $M(G) \overset{\text{def}}{=} M_1 \circ \ldots \circ M_n \in \mathrm{p}\mathbb{MSC}(A, \mathbb{P})$. Since, in a fork-and-join, subcomputations employ disjoint sets of pids, $M(G)$ is well defined and does not depend on the chosen enumeration. We call run G *accepting* if $loc(in(G)) \in L_{\text{init}}$,

$loc(out(G)) \in L_{\mathrm{acc}}$, and $reg(in(G)) = \{x_0 \mapsto p\}$ for some $p \in \mathbb{P}$. The language of $\mathcal{H}$ is $L(\mathcal{H}) \overset{\mathrm{def}}{=} \{ \overset{p}{\phi} \circ M(G) \mid G$ is an accepting run of $\mathcal{H}$ with $reg(in(G)) = \{x_0 \mapsto p\}\} \subseteq \mathbb{MSC}(A, \mathbb{P})$. Note that $L(\mathcal{H})$ is always closed.

Example 4. The bHMSC below models a peer-to-peer protocol. It has only sequential transitions and is defined over $A = \{\mathsf{r}, \mathsf{a}, \mathsf{c}\}$ (request, acknowledgment, communication) with $arity(\mathsf{r}) = arity(\mathsf{a}) = 1$ and $arity(\mathsf{c}) = 0$. The initial register is x_0. A request is forwarded to new processes along with the pid p of the initial process. At some point, a process acknowledges the request, sending its own pid q to the initial process. Processes p and q may then communicate and exchange messages. A generated MSC is depicted beside the bHMSC.



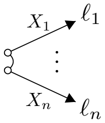
Example 5. The following bHMSC has one fork-and-join transition whose target state $\perp$ is the only final state. Due to the fork, registers can be used simultaneously at different places so that the generated MSCs have a tree-like structure.



Examples 4 and 5 represent important subclasses of bHMSCs, *sequential* and *join-free* bHMSCs, respectively, which we define in the following.

A bHMSC is called *sequential* if it contains only sequential transitions. Thus, the bHMSC from Example 4 is sequential.

Let $\mathcal{H} = (L, X, L_{\mathrm{init}}, L_{\mathrm{acc}}, x_0, T)$ be a bHMSC. By L_{seq}, L_{fork}, and $L_{\perp}$ we denote the sets of locations with outgoing sequential transitions, with outgoing fork-and-join transitions, and without outgoing transitions, respectively.

We say that bHMSC $\mathcal{H}$ is *join-free* if there is a distinguished location $\perp \in L$ such that $L_{\mathrm{acc}} = L_{\perp} = \{\perp\}$ and all fork-and-join transitions are of the form $\ell \to \{(\ell_1, X_1, \perp), \ldots, (\ell_n, X_n, \perp)\} \to \perp$. Thus, the bHMSCs from Examples 4 and 5 are *join-free.* The run of a join-free bHMSC may be viewed as a tree, as it can always be completed towards a run with a single target node. We will, therefore, consider that a fork-and-join transition is of the form ℓ  and rather call it a *fork transition.* Note that any bHMSC generating the MSCs $M(n)$ from Figure 1 is inherently not join-free. Moreover:

Lemma 6. *Join-free bHMSCs are more expressive than sequential bHMSCs.*

The first natural question to ask for a bHMSC $\mathcal{H}$ is whether $L(\mathcal{H}) \neq \emptyset$, i.e., the nonemptiness problem.

Theorem 7. *Nonemptiness of bHMSCs is EXPTIME-complete. It is already EXPTIME-hard for join-free bHMSCs. Nonemptiness of sequential bHMSCs is NP-complete.*

The proofs of the upper bounds use a notion of symbolic runs. EXPTIME-hardness is shown by a reduction from the intersection-nonemptiness problem for tree automata; for NP-hardness, we use a reduction from 3-CNF-SAT.

4 Dynamic Communicating Automata

In this section, we introduce an extension of the model of dynamic communicating automata as presented in [4]. A configuration of a DCA consists of several processes that can exchange messages through FIFO channels. A process can spawn new processes so that there is a priori no bound on the number of processes that participate in a system execution. In contrast to [4], we allow a message to contain process identities and receptions to be non-selective (i.e., a receiver may receive a message without knowing the sender).

Definition 8 (DCA). *A* dynamic communicating automaton (DCA) *over the ranked message alphabet A is a tuple $\mathcal{D} = (S, X, S_{\mathrm{init}}, S_{\mathrm{acc}}, \Delta)$ where S is a finite set of* states *with initial states $S_{\mathrm{init}} \subseteq S$ and accepting states $S_{\mathrm{acc}} \subseteq S$, X is a finite set of* registers*, and Δ is the set of* transitions*. A transition is of the form (s, α, s') where $s, s' \in S$, and α is an action, possibly a* send action *$!_x(a(x_1, \ldots, x_n))$, a* receive action *$?_y(a(y_1, \ldots, y_n))$, or a* spawn action *$x := \mathsf{spawn}(s, z)$, where $x, z \in X$, $y \in X \cup \{*\}$, $s \in S$, $a(x_1, \ldots, x_n) \in A(X \uplus \{\mathsf{self}\})$, and $a(y_1, \ldots, y_n) \in A(X \uplus \{-\})$ such that, for all $i, j \in \{1, \ldots, n\}$, $y_i = y_j \in X$ implies $i = j$.*

When a process executes $!_x(a(\overline{x}))$ with $\overline{x} = (x_1, \ldots, x_n)$, it sends a message to the process whose pid is stored in register x. The message consists of label a as well as $n = \mathit{arity}(a)$ many pids stored in registers $\overline{x}$ (or the sender's pid if $x_i = \mathsf{self}$). Executing $?_y(a(\overline{y}))$, a process receives a message from the process whose pid is stored in y (selective receive) or, in case $y = *$, from any process (non-selective receive). The message must be of the form $a(p_1, \dots, p_n)$. In the resulting configuration, the receiving process updates its local registers $y_1, \dots, y_n$ to $p_1, \dots, p_n$, respectively, unless $y_i = -$. Finally, a process executing $x := \mathsf{spawn}(s, z)$ spawns a new process, whose fresh pid is henceforth stored in register x. The new process starts in state s. Its registers are a copy of the registers of the spawning process, except for z, which is set to the pid of the spawning process.

A *run* of DCA $\mathcal{D}$ on an MSC $M = (E, \lhd, \lambda, \mu) \in \mathbb{MSC}(A, \mathbb{P})$ is a pair (σ, τ), where $\sigma : E \to S$ and $\tau : E \to [X \rightharpoonup \mathbb{P}]$, respecting the following conditions:

- $\sigma_{init(M)} \in S_{\text{init}}$,
- $\tau_{init(M)}$ is undefined everywhere,
- for all $e_1, e_2, f \in E$ with $e_1 \lhd_{\mathsf{proc}} e_2 \lhd_{\mathsf{spawn}} f$, the relation Δ contains a local transition $\sigma_{e_1} \xrightarrow{x := \mathsf{spawn}(s,y)} \sigma_{e_2}$ such that $\sigma_f = s$, $\tau_{e_2} = \tau_{e_1}[x \mapsto \mathit{pid}(f)]$, and $\tau_f = \tau_{e_1}[y \mapsto \mathit{pid}(e_1)]$, and
- for all $e_1, e_2, f_1, f_2 \in E$ with $e_1 \lhd_{\mathsf{proc}} e_2 \lhd_{\mathsf{msg}} f_2$ and $f_1 \lhd_{\mathsf{proc}} f_2$, the relation Δ contains transitions $\sigma_{e_1} \xrightarrow{!_x(a(x_1,\dots,x_n))} \sigma_{e_2}$ and $\sigma_{f_1} \xrightarrow{?_y(a(y_1,\dots,y_n))} \sigma_{f_2}$ such that $\{x, x_1, \dots, x_n\} \subseteq \mathrm{dom}(\tau_{e_1}) \cup \{\mathsf{self}\}$, $\tau_{e_2} = \tau_{e_1}$, $\tau_{e_1}(x) = \mathit{pid}(f_1)$, $\big(y = * \text{ or } \tau_{f_1}(y) = \mathit{pid}(e_1)\big)$, and, letting $p_i = \begin{cases} \tau_{e_1}(x_i) & \text{if } x_i \in X \\ \mathit{pid}(e_1) & \text{if } x_i = \mathsf{self}\,, \end{cases}$ we have $$\mu(e_2, f_2) = a(p_1, \dots, p_n) \text{ and } \tau_{f_2}(z) = \begin{cases} p_i & \text{if } z = y_i \\ \tau_{f_1}(z) & \text{if } z \notin \{y_1, \dots, y_n\}\,. \end{cases}$$

Here, σ_e and τ_e denote $\sigma(e)$ and $\tau(e)$, respectively. Moreover, $\tau_e[x \mapsto p]$ is the partial mapping that maps x to p and coincides with τ_e on all other arguments.

The run (σ, τ) is accepting if $\sigma_e \in S_{\text{acc}}$ for all $e \in \{\max_p(M) \mid p \in \mathit{Pids}(M)\}$. By $L(\mathcal{D})$, we denote the set of MSCs M over A and $\mathbb{P}$ such that there is an accepting run of $\mathcal{D}$ on M. Note that $L(\mathcal{D})$ is closed, i.e., $L(\mathcal{D}) = [L(\mathcal{D})]$. Nonemptiness is undecidable for CA, and consequently also for DCA.

There are languages L that are not the language of a DCA, but for which there is a DCA *implementing* them up to some refinement. The refinement allows a DCA to attach more information to a message than the specification provides, for example additional pids. This is formalized as follows. Let A, B be ranked alphabets and let $h : B \to A$. We say that the pair (B, h) is a refinement of A if, for all $b \in B$, $\mathit{arity}(h(b)) \leq \mathit{arity}(b)$. We can extend h to a mapping $h : \mathbb{MSC}(B, \mathbb{P}) \to \mathbb{MSC}(A, \mathbb{P})$ as follows: for an MSC $M = (E, \lhd, \lambda, \mu) \in \mathbb{MSC}(B, \mathbb{P})$, we let $h(M) = (E, \lhd, \lambda, \mu') \in \mathbb{MSC}(A, \mathbb{P})$ where $\mu'(e, f) = h(b)(p_1, \dots, p_{\mathit{arity}(h(b))})$ whenever $\mu(e, f) = b(p_1, \dots, p_n)$. The mapping is then further extended to sets of MSCs as expected.

Definition 9 (realizable, implementable). *We call a set* $L \subseteq \mathbb{MSC}(A, \mathbb{P})$ realizable *if* $[L] = L(\mathcal{D})$ *for some DCA* $\mathcal{D}$. *We say that* L *is* implementable *if there are a refinement* (B, h) *of* A *and a DCA* $\mathcal{D}$ *over* B *such that* $[L] = h(L(\mathcal{D}))$.

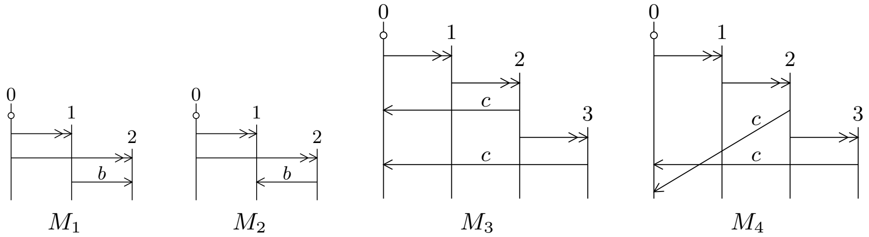


Fig. 2. Realizability vs. Implementability

For both realizability and implementability, it is necessary that the sender p of a message knows the receiver q at the time of sending, i.e., q should be stored in some register of p. Note that this aspect does not arise in simple CA.

Example 10. The MSC language $\{M_1\}$ (see Figure 2) is not implementable, as process 1 does not know 2 when sending message b. However, $\{M_2\}$ is implementable (and even realizable), as 2 may know 1: when spawning 2, process 0 can communicate the pid 1 to 2. The language $\{M_3\}$ is not realizable: as process 0 does neither know 2 nor 3 when it receives the messages, it has to use a non-selective receive. But then, the DCA also accepts M_4. On the other hand, $\{M_3, M_4\}$ is realizable. However, $\{M_3\}$ and $\{M_4\}$ are implementable by refining the messages from 2 and 3.

5 Executability

An accepting run of a bHMSC generates an MSC. However, this MSC need not be implementable always, as Example 10 shows. Unfortunately, implementability (and also realizability) is undecidable for bHMSCs, which follows from undecidability for HMSCs over a fixed finite set of processes [12,1].

Theorem 11 (cf. [12,1]). *Implementability and realizability of bHMSCs are undecidable. This already holds for sequential bHMSCs.*

We now focus on implementability and introduce an effective necessary criterion, called executability: every sender in a generated MSC should be "aware of" the receiver and the processes whose pids are used as message parameters.

Given an MSC M, a process q and an event e of M, we write $q \rightsquigarrow_M e$ if there is a path from the minimal event $\min_q(M)$ of q to e in M. This path might involve the reversal of the spawn edge that started q. That is, $q \rightsquigarrow_M e$ if $(\min_q(M), e) \in (\lhd \cup \lhd_{\mathsf{spawn}}^{-1})^*$. Intuitively, $q \rightsquigarrow_M e$ indicates that the process executing e is aware of process q. Next, we formally define executability of MSCs.

Definition 12 (executability). *Let $M \in \mathbb{MSC}(A, \mathbb{P})$. A message $(e, f) \in \lhd_{\mathsf{msg}}$ of M with message contents $a(p_1, \ldots, p_n)$ is* executable *if $q \rightsquigarrow_M e$, for every*

$q \in \{pid(f), p_1, \ldots, p_n\}$. Moreover, M is executable *if each of its messages is executable. Finally, a bHMSC $\mathcal{H}$ is* executable *if each MSC from $L(\mathcal{H})$ is executable.*

For example, in Figure 2, M_2, M_3, M_4 are executable, while M_1 is not. Let M be an MSC and $\mathcal{H}$ be a bHMSC. One can verify that 1) M is executable iff $\{M\}$ is implementable, and 2) $\mathcal{H}$ is executable if it is implementable (while the converse might fail). Unlike implementability, executability is decidable:

Theorem 13. *Executability of bHMSCs is EXPTIME-complete. Moreover, the lower bound already holds for bHMSCs that are join-free.*

The lower bound is deduced from the lower bound of the nonemptiness problem (Theorem 7). For the upper bound, we abstract the knowledge of processes by a finite number of *awareness relations*, so as to work over symbolic runs.

6 Implementing Guarded Join-Free bHMSCs

We identify a subclass of bHMSCs for which executability and implementabiliy coincide. *Guarded* bHMSCs are based on the notion of a leader process, which determines the next transition to be taken in a bHMSC. They are an adaptation of locality from [10]. For $M = (E, \lhd, \lambda, \mu) \in \mathrm{p}\mathbb{MSC}(A, X)$, $Y \subseteq X$, and $x \in X$, we write $Y \preceq_M x$ if $x \in \mathit{Pids}(M) \cap Y$ and, for all $y \in \mathit{Pids}(M) \cap Y$, $\max_y(M) \lhd^* \max_x(M)$. Intuitively, all processes in $\mathit{Pids}(M) \cap Y$ terminate before x.

Definition 14 (guarded). *A join-free bHMSC $\mathcal{H} = (L, X, L_{\mathrm{init}}, L_{\mathrm{acc}}, x_0, T)$ is called* guarded *if $L = L_{\mathrm{seq}} \uplus L_{\mathrm{fork}} \uplus \{\bot\}$, $L_{\mathrm{init}} \subseteq L_{\mathrm{seq}}$, and there is a mapping $\mathit{leader} : L_{\mathrm{seq}} \to X$ such that*

1. *for all partial MSCs $M = (E, \lhd, \lambda, \mu) \in \mathrm{p}\mathbb{MSC}(A, X)$ that occur in $\mathcal{H}$, $(E, \lhd^*)$ has a unique minimal element e; we let $\mathit{first}(M) \overset{\mathit{def}}{=} \mathit{pid}(e)$,*
2. *for all sequential transitions $\ell \xrightarrow{M} \ell'$, it holds $\mathit{leader}(\ell) = \mathit{first}(M)$, and, if $\ell' \in L_{\mathrm{seq}}$, also $X \preceq_M \mathit{leader}(\ell')$, and*
3. *for all transition patterns $\ell \xrightarrow{M} \ell' \begin{array}{l} \xrightarrow{X_1} \ell_1 \\ \quad\vdots \\ \xrightarrow{X_n} \ell_n \end{array}$ and all $i \in \{1, \ldots, n\}$, we have $\ell_i \in L_{\mathrm{seq}}$ and $X_i \preceq_M \mathit{leader}(\ell_i)$.*

Example 15. The bHMSCs from Examples 4 and 5 are both guarded.

Theorem 16. *A guarded join-free bHMSC is implementable if and only if it is executable. Moreover, if it is implementable, an equivalent DCA can be constructed in exponential time.*

Towards an implementation of a given guarded join-free bHMSC $\mathcal{H}$, we first enrich locations of $\mathcal{H}$ with awareness relations (in the same spirit as in the proof of Theorem 13). Then, we rely on techniques employed in the context of a bounded number of processes [11,10], to build a DCA (together with a refinement) that recognizes $L(\mathcal{H})$.

Note that guardedness does not yield better complexities:

Theorem 17. *Nonemptiness and executability of guarded join-free bHMSCs are both EXPTIME-complete.*

7 Future Work

In future work, we aim at finding classes of bHMSCs for which executability and implementability coincide and that are not necessarily join-free or guarded (e.g., by transferring concepts like fork-acyclicity from branching automata to bHMSCs). Moreover, connections with the π-calculus [19] should be explored.

Acknowledgments. We thank the anonymous reviewers as well as Martin Schuster and Thomas Zeume for their helpful comments.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. Theoretical Computer Science 331(1), 97–114 (2005)
2. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. Logical Methods in Computer Science 7(4) (2011)
3. Bollig, B., Cyriac, A., Hélouët, L., Kara, A., Schwentick, T.: Dynamic Communicating Automata and Branching High-Level MSCs. Research Report LSV-12-20, LSV (November 2012)
4. Bollig, B., Hélouët, L.: Realizability of Dynamic MSC Languages. In: Ablayev, F., Mayr, E.W. (eds.) CSR 2010. LNCS, vol. 6072, pp. 48–59. Springer, Heidelberg (2010)
5. Borgström, J., Gordon, A., Phillips, A.: A chart semantics for the Pi-calculus. Electronic Notes in Theoretical Computer Science 194(2), 3–29 (2008)
6. Bozzelli, L., La Torre, S., Peron, A.: Verification of well-formed communicating recursive state machines. Theoretical Computer Science 403(2-3), 382–405 (2008)
7. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the ACM 30(2) (1983)
8. Buscemi, M.G., Sassone, V.: High-Level Petri Nets as Type Theories in the Join Calculus. In: Honsell, F., Miculan, M. (eds.) FOSSACS 2001. LNCS, vol. 2030, pp. 104–120. Springer, Heidelberg (2001)
9. Genest, B., Kuske, D., Muscholl, A.: A Kleene theorem and model checking algorithms for existentially bounded communicating automata. Information and Computation 204(6), 920–956 (2006)

10. Genest, B., Muscholl, A., Seidl, H., Zeitoun, M.: Infinite-state high-level MSCs: Model-checking and realizability. Journal of Computer and System Sciences 72(4), 617–647 (2006)
11. Hélouët, L., Jard, C.: Conditions for synthesis of communicating automata from HMSCs. In: Proceedings of FMICS 2000, pp. 203–224. Springer (2000)
12. Henriksen, J.G., Mukund, M., Narayan Kumar, K., Sohoni, M.A., Thiagarajan, P.S.: A theory of regular MSC languages. Inf. Comput. 202(1), 1–38 (2005)
13. ITU-TS: ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva (February 2011)
14. Leucker, M., Madhusudan, P., Mukhopadhyay, S.: Dynamic Message Sequence Charts. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 253–264. Springer, Heidelberg (2002)
15. Lodaya, K., Weil, P.: Series-parallel languages and the bounded-width property. Theoretical Computer Science 237(1-2), 347–380 (2000)
16. Lodaya, K., Weil, P.: Rationality in algebras with a series operation. Information and Computation 171(2), 269–293 (2001)
17. Lohrey, M.: Realizability of high-level message sequence charts: closing the gaps. Theoretical Computer Science 309(1-3), 529–554 (2003)
18. Meyer, R.: On Boundedness in Depth in the π-Calculus. In: Ausiello, G., Karhumäki, J., Mauri, G., Ong, L. (eds.) Proceedings of IFIP TCS 2008, vol. 273, pp. 477–489. Springer, Boston (2008)
19. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Information and Computation 100(1), 1–40 (1992)
20. Morin, R.: Recognizable Sets of Message Sequence Charts. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 523–534. Springer, Heidelberg (2002)
21. Segoufin, L.: Automata and Logics for Words and Trees over an Infinite Alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)