# Dynamic Communicating Automata and Branching High-Level MSCs

Benedikt Bollig, Aiswarya Cyriac, Loïc Hélouët,
Ahmet Kara, and Thomas Schwentick

November 2012

Research report LSV-12-20

# 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
[2] INRIA, Rennes
[3] Lehrstuhl Informatik 1, TU Dortmund

**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 on the specification side. Our focus is on the implementability problem: given a branching high-level MSC, 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 branching high-level MSCs is EXPTIME-complete. We then identify a class of branching high-level MSCs for which executability and implementability coincide. In particular, we present an algorithm to transform an executable branching high-level MSC into an equivalent DCA.

## 1 Introduction

Communicating automata (CA) [6] 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 [12]. A natural question in this context is the implementability problem, which asks if a given HMSC can be translated into an equivalent CA [10, 1, 11, 19, 9, 16, 8].

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 [3], 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 exchanged through messages. The use of registers in DCA suggests close

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

connections with register automata (also known as finite-memory automata) and formal languages over infinite alphabets (cf. [20] for an overview).

DCA are inherently hard to analyze and to synthesize. To facilitate the specification of dynamic systems, we introduce branching HMSCs. Just like DCA generalize CA, branching HMSCs extend HMSCs. They are based on branching automata [14, 15], 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, branching HMSCs use finitely many registers to store pids. In a sense, branching HMSCs combine branching automata and register automata.

In this paper, we study the implementability question: given a branching HMSC, is there an equivalent DCA? This question is undecidable already in the case of a bounded number of processes. 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 branching HMSCs to be EXPTIME-complete. Moreover, we identify the fragment of guarded join-free branching HMSCs, 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 [13], 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 [13] once the latter is adjusted to our setting. In [4], an MSC semantics was given for the $\pi$-calculus. Note that the problems studied in [13] and [4] are very different from ours and do not distinguish between a specification and an implementation.

The present paper supersedes [3] 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 [3].

Other formalisms with dynamic process creation (not necessarily involving message passing) can be found, for example, in [7, 17, 5, 2]. However, these papers consider neither an MSC based semantics nor implementability aspects.

**Outline.** Section 2 settles some notations and defines MSCs. Branching HMSCs are presented in Section 3. We then study their nonemptiness problem in Section 4. We introduce DCA in Section 5. Section 6 studies the executability problem. Finally, in Section 7, we identifiy a fragment of branching HMSCs for which executability and implementability coincide. We conclude in Section 8.

## 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 set $A$ where every letter $a \in A$ has an arity $\mathit{arity}(a) \in \mathbb{N}$. For any set $B$, we let $A(B) = \{a(b_1, \ldots, b_n) \mid a \in A,\ n = \mathit{arity}(a), \text{ and } b_1, \ldots, b_n \in B\}$.

Let $\mathbb{P} = \{0, 1, 2, \ldots\}$ be the infinite set of *process identifiers* (pids, for short). We also fix a finite ranked alphabet $A$ of *message labels*. A *message* is of the form $a(p_1, \ldots, p_n)$ where $a \in A$ is a label, $n = \mathit{arity}(a)$, and $p_1, \ldots, p_n \in \mathbb{P}$ are the pids attached to $a$. That is, a message is an element from $A(\mathbb{P})$.

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

*Definition 1 (MSC).* A *message sequence chart (MSC)* over $A$ and a nonempty set $P$ (of processes) 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 $E_\theta \overset{\text{def}}{=} \{e \in E \mid \lambda(e) \in \{\theta\} \times P\}$. 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 $E_p \overset{\text{def}}{=} \{e \in E \mid \mathit{pid}(e) = p\}$. We require the following:

1. $(E, \lhd^*)$ is a partial order with a unique minimal element $\mathit{init}(M)$,
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. $E_{\mathsf{start}} = \{e \in E \mid \text{there is no } e' \in E \text{ such that } e' \lhd_{\mathsf{proc}} e\}$,
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 $E_{\mathsf{spawn}}$ 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 $\mathit{arity}(b) =$

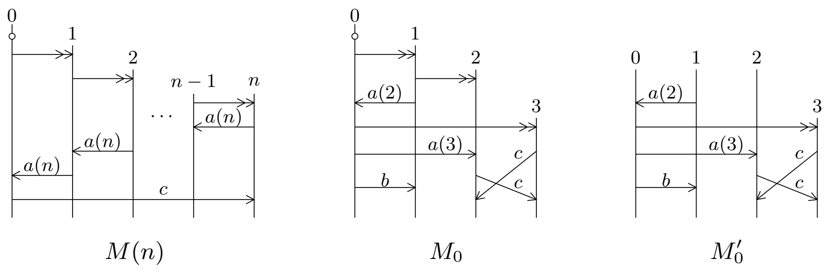
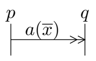
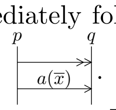


**Fig. 1.** Two MSCs and a partial MSC

*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 *init*$(M)$, which is located on the process with pid 0, is depicted as a small circle.

Depending on the application, a spawn in an MSC may have quite 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. Actually, we will use 

as an abbreviation for 

.

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 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 $L = [L]$.

## 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 [14, 15]. 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. Let $M = (E, \lhd, \lambda, \mu) \in \mathrm{p}\mathbb{MSC}(A, P)$ be a partial MSC. Again, for $e \in E$, we denote by *pid*$(e)$ the unique process $p \in P$ such that $e \in E_p$. 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 $E_p \neq \varnothing$, 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 also set $\min(M) = \{\min_p(M) \mid p \in P \text{ such that } E_p \neq \varnothing\}$ (which, in a partial MSC, may be different from $E_{\mathsf{start}}$) and define $\max(M)$ accordingly.

We let $\mathit{Pids}(M) \overset{\text{def}}{=} \{p \in P \mid E_p \neq \varnothing\}$. By $\mathit{Free}(M) \overset{\text{def}}{=} \{p \in \mathit{Pids}(M) \mid E_{\mathsf{start}} \cap E_p = \varnothing\}$, 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, $\mathit{Bnd}(M) \overset{\text{def}}{=} \mathit{Pids}(M) \backslash \mathit{Free}(M)$ 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* $M \circ M'$ amounts to drawing identical processes one below the other. It is defined if $\mathit{Bnd}(M') \cap \mathit{Pids}(M) = \varnothing$, i.e., processes spawned in $M'$ do not already exist in $M$. 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 $\hat{\mu} = \mu \cup \mu'$.

Note that $M \circ M'$ is indeed a partial MSC with $\mathit{Pids}(M \circ M') = \mathit{Pids}(M) \cup \mathit{Pids}(M')$ and $\mathit{Bnd}(M \circ M') = \mathit{Bnd}(M) \cup \mathit{Bnd}(M')$.

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 a 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.

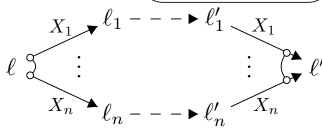
- A sequential transition is a triple $(\ell, M, \ell') \in L \times \mathrm{p}\mathbb{MSC}(A, X) \times L$, usually written $\ell \xrightarrow{M} \ell'$, such that $\mathit{Free}(M) \neq \varnothing$ and $\mathit{MsgPar}(M) \cap \mathit{Bnd}(M) = \varnothing$ (the latter guarantees an unambigous interpretation 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 $n \geqslant 1$ 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 $$\begin{tikzcd} & \ell_1 \arrow[r, dashed] & \ell_1' \arrow[dr, "X_1"] & \\ \ell \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots & \vdots & \ell' \\ & \ell_n \arrow[r, dashed] & \ell_n' \arrow[ur, "X_n"'] & \end{tikzcd}$$

The idea of a fork-and-join transition is that, at location $\ell$, $n$ 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 $\ell_i'$ (the registers as such may be used, but not their contents at $\ell$). Then, the main computation resumes in $\ell'$, and registers in $X_i$ adopt the final assignment from the $i$-th subcomputation.

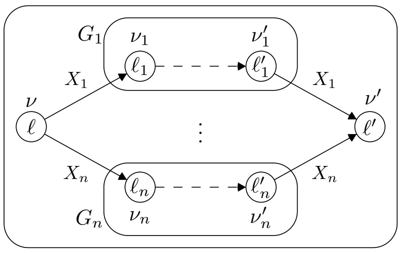
We associate MSCs with a bHMSC through the notion of runs, which we will define next after some preparation. A partial mapping $\nu : X \rightharpoonup \mathbb{P}$ is a *register assignment* if it is injective on $\mathrm{dom}(\nu)$. The set of register assignments is denoted by $\mathcal{R}(X)$. For $\nu \in \mathcal{R}(X)$ 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 $\nu$ and yields $\nu'$) if

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

A run $G = (V, R, \mathit{loc}, \mathit{reg}, \rho)$ of the bHMSC $\mathcal{H}$ consists of a finite directed acyclic graph $(V, R)$ with a unique source node $\mathit{in}(G)$, a unique sink node $\mathit{out}(G)$, and labeling functions $\mathit{loc} : V \to L$, $\mathit{reg} : V \to \mathcal{R}(X)$, 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 $\delta = \ell \xrightarrow{M} \ell'$ 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 $G = G(\delta, \nu, \nu') \overset{\text{def}}{=}$ $\begin{tikzcd} \overset{\nu}{\ell} \arrow[r, "M'"] & \overset{\nu'}{\ell'} \end{tikzcd}$ is a run of $\mathcal{H}$. We set $\mathit{Pids}(G) \overset{\text{def}}{=} \nu(X) \cup \mathit{Pids}(M')$ and $\mathit{Bnd}(G) \overset{\text{def}}{=} \mathit{Bnd}(M')$.
- Consider runs $G_1 = \begin{tikzcd} \overset{\nu_1}{\ell_1} \arrow[r, dashed] & \overset{\nu_2}{\ell_2} \end{tikzcd}$ and $G_2 = \begin{tikzcd} \overset{\nu_2}{\ell_2} \arrow[r, dashed] & \overset{\nu_3}{\ell_3} \end{tikzcd}$ of $\mathcal{H}$. If $\mathit{Pids}(G_1) \cap \mathit{Bnd}(G_2) = \varnothing$, then the graph $G = G_1 \circ G_2 \overset{\text{def}}{=}$ $\begin{tikzcd} \overset{\nu_1}{\ell_1} \arrow[r, dashed] & \overset{\nu_2}{\ell_2} \arrow[r, dashed] & \overset{\nu_3}{\ell_3} \end{tikzcd}$ is a run of $\mathcal{H}$. We set $\mathit{Pids}(G) \overset{\text{def}}{=} \mathit{Pids}(G_1) \cup \mathit{Pids}(G_2)$ and $\mathit{Bnd}(G) \overset{\text{def}}{=} \mathit{Bnd}(G_1) \cup \mathit{Bnd}(G_2)$.
- For $n \geqslant 1$, let $G_1 = \begin{tikzcd} \overset{\nu_1}{\ell_1} \arrow[r, dashed] & \overset{\nu_1'}{\ell_1'} \end{tikzcd}$, $\ldots$, $G_n = \begin{tikzcd} \overset{\nu_n}{\ell_n} \arrow[r, dashed] & \overset{\nu_n'}{\ell_n'} \end{tikzcd}$ be runs, $\delta =$ 

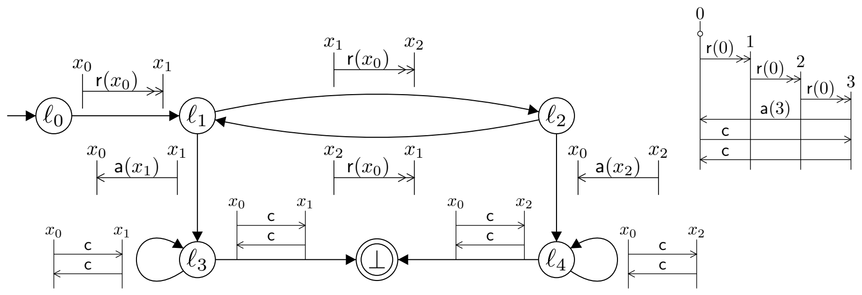
be a fork-and-join transition, and $\nu, \nu' \in \mathcal{R}(X)$ be register assignments. Then, the graph

$G = FJ(\delta, \nu, G_1, \ldots, G_n, \nu') \overset{\text{def}}{=}$ 

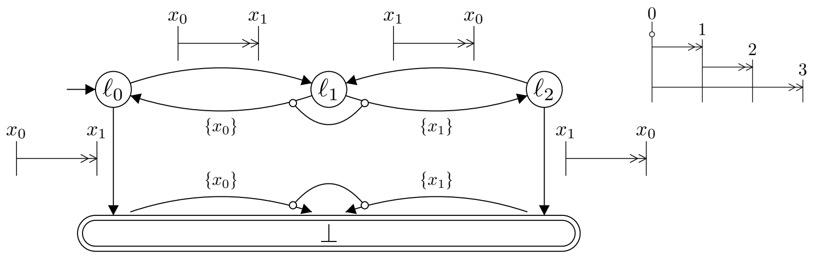
is a run of $\mathcal{H}$ if $\mathit{Bnd}(G_i) \cap (\nu(X) \cup \bigcup_{j \neq i} \mathit{Pids}(G_j)) = \varnothing$ and $\nu_i = \nu_{\restriction X_i}$ for all $i \in \{1, \ldots, n\}$, and $\nu' = \nu_{\restriction X_0} \cup \bigcup_{i \in \{1,\ldots,n\}} (\nu_i')_{\restriction X_i}$ where $X_0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

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 $M(G) \overset{\text{def}}{=} M_1 \circ \ldots \circ M_n \in \mathrm{p}\mathbb{MSC}(A, \mathbb{P})$. Clearly, $M(G)$ is well defined and does not depend on the chosen enumeration. We call run $G$ *accepting* if $loc(in(G)) \in L_{\text{init}}$, $loc(out(G)) \in L_{\text{acc}}$, and $reg(in(G)) = \{x_0 \mapsto p\}$ for some $p \in \mathbb{P}$. The language of $\mathcal{H}$ is $L(\mathcal{H}) \overset{\text{def}}{=} \{ \overset{p}{\diamond} \circ M(G) \mid G$ 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.

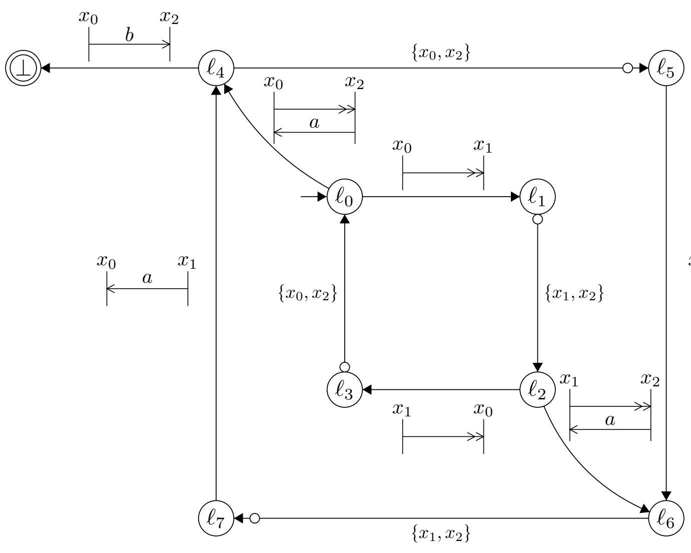
Let $\mathcal{H} = (L, X, L_{\text{init}}, L_{\text{acc}}, x_0, T)$ be a bHMSC. By $L_{\text{seq}}$, $L_{\text{fork}}$, and $L_{\perp}$ we denote the sets of locations with outgoing sequential transitions, with outgoing fork-and-join transitions, and without outgoing transitions, respectively. Without loss of generality, we can assume that a bHMSC is *normalized*, i.e., it satisfies (i) $L_{\text{acc}} = L_{\perp} = \{\perp\}$ for some distinguished location $\perp \in L$, (ii)

$L = L_{\text{seq}} \uplus L_{\text{fork}} \uplus \{\bot\}$, (iii) $L_{\text{fork}} \cap L_{\text{init}} = \varnothing$, and (iv) any two distinct fork transitions have distinct source states. Both example bHMSCs are normalized.

A bHMSC is called *join-free* if it is normalized and all fork-and-join transitions are of the form $\ell \to \{(\ell_1, X_1, \bot), \ldots, (\ell_n, X_n, \bot)\} \to \bot$. The run of a join-free bHMSC can 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

$$\begin{tikzcd}
 & \ell_1 \\
\ell \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots \\
 & \ell_n
\end{tikzcd}$$

and rather call it a *fork transition*. A bHMSC generating the MSCs $M(n)$, with $n \geqslant 1$, from Figure 1 can be found below. It is (inherently) not join-free.



Note that there is no sequential bHMSC that is equivalent to the join-free bHMSC from Example 5. The reason is that there is a bound on the number of pids that a run of a sequential bHMSC can remember at each position.

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

## 4 Nonemptiness of bHMSCs

To check whether a given bHMSC is nonempty, it is *not* enough to check whether some final location is reachable from an initial location. It is possible that a path from an initial location to a final location involves a partial MSC which uses a register $x$ as a free register, where the run so far has not yet initialized the register $x$. Hence it is important to check which registers have been initialized (or defined) before taking a transition. However, checking nonemptiness is decidable.

**Theorem 7.** *Nonemptiness of bHMSCs is EXPTIME-complete.*

The proof of the lower bound is given in Subsection 4.3. The upper bound will be shown in Subsection 4.2. Before that we show in Subsection 4.1 that the nonemptiness problem for a bHMSC $\mathcal{H}$ can be reduced to the question whether there exists a *symbolic run* for $\mathcal{H}$. Instead of concrete register assignments and partial MSCs over $\mathbb{P}$, symbolic runs take only into account which registers are defined at the nodes and which partial MSCs over $X$ are used in the sequential transitions.

## 4.1 Symbolic Runs

Let $\mathcal{H} = (L, X, L_{\text{init}}, L_{\text{acc}}, x_0, T)$ be a bHMSC over the set $A$ of message labels. A symbolic run $S = (V, R, \mathit{loc}, \mathit{def}, \pi)$ of $\mathcal{H}$ is a labelled graph defined almost like a (normal) run of $\mathcal{H}$. However, instead of a function $\mathit{reg}$ that maps every node $v$ to a register assignment there is a function $\mathit{def}$ that maps nodes to a set of registers (basically, $\mathit{def}(v)$ is just the domain of $\mathit{reg}(v)$). Furthermore, the partial MSCs with which edges can be labelled are over $A$ and $X$ as opposed to $A$ and $\mathbb{P}$. To reflect this latter difference, we denote the label mapping of a symbolic run by $\pi$ instead of $\rho$.

For a partial MSC $M \in \text{p}\mathbb{MSC}(A, X)$ and sets $D, D' \subseteq X$ we write $D \xrightarrow{M} D'$ if $\mathit{Free}(M) \cup \mathit{MsgPar}(M) \subseteq D$ and $D' = D \cup \mathit{Bnd}(M)$. The set of symbolic runs is defined inductively as follows:

– Let $D, D' \subseteq X$ and $M$ be a partial MSC such that $\ell \xrightarrow{M} \ell'$ is a sequential transition from $T$ and $D \xrightarrow{M} D'$. Then, the graph $S =$ $\begin{tikzcd} \overset{D}{\ell} \arrow[r, "M"] & \overset{D'}{\ell'} \end{tikzcd}$ is a symbolic run of $\mathcal{H}$ with $\mathit{Bnd}(S) \overset{\text{def}}{=} \mathit{Bnd}(M)$.

– Let $S_1 =$ $\begin{tikzcd} \overset{D_1}{\ell_1} \arrow[r, dashed] & \overset{D_2}{\ell_2} \end{tikzcd}$ and $S_2 =$ $\begin{tikzcd} \overset{D_2}{\ell_2} \arrow[r, dashed] & \overset{D_3}{\ell_3} \end{tikzcd}$ be symbolic runs of $\mathcal{H}$. Then $S = S_1 \circ S_2 \overset{\text{def}}{=}$ $\begin{tikzcd} \overset{D_1}{\ell_1} \arrow[r, dashed] & \overset{D_2}{\ell_2} \arrow[r, dashed] & \overset{D_3}{\ell_3} \end{tikzcd}$ is a symbolic run of $\mathcal{H}$ with $\mathit{Bnd}(S) \overset{\text{def}}{=} \mathit{Bnd}(S_1) \cup \mathit{Bnd}(S_2)$.

– For $1 \leqslant i \leqslant n$, let $S_i =$ $\begin{tikzcd} \overset{D_i}{\ell_i} \arrow[r, dashed] & \overset{D'_i}{\ell'_i} \end{tikzcd}$ be symbolic runs of $\mathcal{H}$.

Let
$$\begin{tikzcd} & \ell_1 \arrow[r, dashed] & \ell'_1 \arrow[dr, "X_1"] & \\ \ell \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots & \vdots & \ell' \\ & \ell_n \arrow[r, dashed] & \ell'_n \arrow[ur, "X_n"'] & \end{tikzcd}$$
be a fork-and-join-transition from $T$ and $D, D' \subseteq X$.

Then,
$$\begin{tikzcd} & \overset{D_1}{\ell_1} \arrow[r, dashed] & \overset{D'_1}{\ell'_1} \arrow[dr, "X_1"] & \\ \overset{D}{\ell} \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & & \vdots & \overset{D'}{\ell'} \\ & \underset{D_n}{\ell_n} \arrow[r, dashed] & \underset{D'_n}{\ell'_n} \arrow[ur, "X_n"'] & \end{tikzcd}$$
is a symbolic run of $\mathcal{H}$ with $\mathit{Bnd}(S) \overset{\text{def}}{=}$ $\bigcup_{i \in \{1,\ldots n\}} (\mathit{Bnd}(S_i) \cap X_i)$ where

- $D_i = X_i \cap D$ for $i \in \{1, \ldots, n\}$, and
- $D' = (D \cap X_0) \cup \bigcup_{i \in \{1,\ldots,n\}} (D'_i \cap X_i)$ (Recall that $X_0 = X \backslash (X_1 \cup \ldots \cup X_n)$)

We call a symbolic run $S$ accepting if $loc(in(S)) \in L_{\text{init}}$, $loc(out(S)) \in L_{\text{acc}}$, and $def(in(S)) = \{x_0\}$.

We define a mapping $\mathsf{symb}$ from (normal) runs to symbolic runs. Intuitively, from a run $G$ we get the corresponding symbolic run $\mathsf{symb}(G)$ by replacing the register assignments $\nu$ at the nodes by their domains and the partial MSCs over $A$ and $\mathbb{P}$ at the edges by the partial MSCs over $A$ and $X$ they result from (which is the actual partial MSC that originally labelled the transition). More formally, for a run $G = (V, R, loc, reg, \rho)$ the corresponding symbolic run $\mathsf{symb}(G) = (V', R', loc', def, \pi)$ is defined as follows:

- $V' = V$
- $R' = R$
- $loc' = loc$
- for all $v \in V'$: $def(v) = \mathrm{dom}(reg(v))$
- for all $(v, v') \in R$:

$$\pi(v, v') = \begin{cases} X' & \text{if } \rho(v, v') = X' \subseteq X \\ M & \text{if the edge } (v, v') \text{ results from a sequential transition } \delta = \ell \xrightarrow{M} \ell' \end{cases}$$

The relationship between runs and symbolic runs is highlighted by the following lemma.

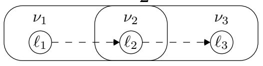
**Lemma 8.** *Let $\mathcal{H}$ be a bHMSC.*

*(a) For every run $G$ of $\mathcal{H}$, $\mathsf{symb}(G)$ is a symbolic run of $\mathcal{H}$. Furthermore, $\mathsf{symb}(G)$ is accepting if and only if $G$ is accepting.*
*(b) For every symbolic run $S$ of $\mathcal{H}$, there exists a run $G$ of $\mathcal{H}$ such that $S = \mathsf{symb}(G)$.*

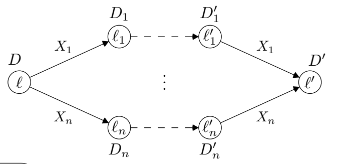
*Proof.* Statement (a) follows by a straightfoward induction.

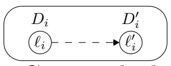
Towards (b), we will explain how to construct inductively a normal run $G$ with $\mathsf{symb}(G) = S$ from a given symbolic run $S$.

For the base case, let $S = \begin{tikzcd} \overset{D}{\ell} \arrow[r, "M"] & \overset{D'}{\ell'} \end{tikzcd}$ be a symbolic run resulting from a sequential transition $\ell \xrightarrow{M} \ell'$. Let $\nu$ and $\nu'$ be register assignments assigning distinct processes to all registers in $D$ and $D'$ respectively such that $\nu(Bnd(M)) \cap \nu'(Bnd(M)) = \varnothing$. As we have an infinite supply of process id, $\nu$ and $\nu'$ can be indeed defined in this way. Clearly, $G = \begin{tikzcd} \overset{\nu}{\ell} \arrow[r, "\nu'(M)"] & \overset{\nu'}{\ell'} \end{tikzcd}$ is a normal run and $\mathsf{symb}(G) = S$.

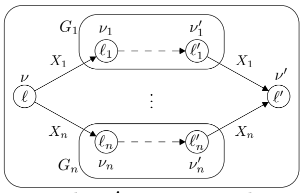
For the first inductive case, let $S = \begin{tikzcd} \overset{D_1}{\ell_1} \arrow[r, dashed] & \overset{D_2}{\ell_2} \arrow[r, dashed] & \overset{D_3}{\ell_3} \end{tikzcd}$ be a symbolic run resulting from the sub runs $S_1 = \begin{tikzcd} \overset{D_1}{\ell_1} \arrow[r, dashed] & \overset{D_2}{\ell_2} \end{tikzcd}$ and $S_2 = \begin{tikzcd} \overset{D_2}{\ell_2} \arrow[r, dashed] & \overset{D_3}{\ell_3} \end{tikzcd}$. By induction hypothesis, there are $G_1 = \begin{tikzcd} \overset{\nu_1}{\ell_1} \arrow[r, dashed] & \overset{\nu_2}{\ell_2} \end{tikzcd}$ and $G_2 = \begin{tikzcd} \overset{\nu_2'}{\ell_2} \arrow[r, dashed] & \overset{\nu_3}{\ell_3} \end{tikzcd}$ such that $\mathsf{symb}(G_1) = S_1$ and $\mathsf{symb}(G_2) = S_2$. We can assume that $\nu_2 = \nu_2'$ and that the process ids that appear in $G_2$ but not in the $\mathsf{Image}(\nu_2)$ are different from those appearing in $G_1$. Notice that such runs exist, again because we have an infinite supply of process ids (or we can change $\nu_2'$ and build a new run $G_2'$ such that this condition holds). Then, obviously for the run $G =$ 

it holds that $\mathsf{symb}(G) = S$.

For the second inductive case, let $S =$ 

be a symbolic run with subruns $S_i =$ 

for $1 \leqslant i \leqslant n$. By induction hypothesis, there exists normal runs $G_i =$ such that $\mathsf{symb}(G_i) = S_i$ for $1 \leqslant i \leqslant n$. Again, as we have an infinite supply of processes we can assume that for every pair $i, j \in \{1, \ldots, n\}$ the runs $G_i$ and $G_j$ do not contain any common process.

Now consider the run $G =$ 

with subruns $G_1, \ldots, G_n$ and $\nu$ and $\nu'$ are defined on $D$ and $D'$ respectively such that, $\nu_i = \nu_{\upharpoonright X_i}$ and $\nu' = \nu_{\upharpoonright X_0} \cup \bigcup_i \nu_i'$. Clearly, $S = \mathsf{symb}(G)$. □

Now we can establish the link between the nonemptiness problem for bHMSCs and symbolic runs.

**Lemma 9.** *For every bHMSC $\mathcal{H}$, $L(\mathcal{H}) \neq \varnothing$ if and only if there is an accepting symbolic run of $\mathcal{H}$.*

*Proof.*

$$\begin{aligned} L(\mathcal{H}) \neq \varnothing &\Leftrightarrow \text{there is an MSC } M \in L(\mathcal{H}) \\ &\Leftrightarrow \text{there is an accepting run } G \text{ of } \mathcal{H} \text{ with } \mathit{reg}(\mathit{in}(G)) = \{x_0 \mapsto p\} \\ &\phantom{\Leftrightarrow} \text{ for some } p \in \mathbb{P} \text{ (by definition of } L(\mathcal{H})) \\ &\Leftrightarrow \text{there is an accepting symbolic run } S \text{ of } \mathcal{H} \text{ (by Lemma 8).} \end{aligned}$$

□

Now we are ready to describe the algorithm for nonemptiness checking.

## 4.2 EXPTIME-Procedure for Nonemptiness Checking

We are now prepared to prove the following.

**Lemma 10.** *Nonemptiness of bHMSCs can be tested in exponential time.*

*Proof.* Given a bHMSC $\mathcal{H}$, a *symbolic state* is a pair $s = (\ell, D)$ where $\ell \in L$ and $D \subseteq X$. Our algorithm computes the set $R$ of pairs $(s, s')$ of symbolic states for which there exists a symbolic run with $s$ as initial and $s'$ as final state. By Lemma 9, it suffices to check whether some pair $((\ell_0, \{x_0\}), (\ell_f, D)) \in R$, for some $\ell_0 \in L_{\text{init}}$, $\ell_f \in L_{\text{acc}}$ and $D \subseteq X$.

The relation $R$ can be computed by a straightforward, monotone fixed point computation. As the number of symbolic states is exponential in the size of $\mathcal{H}$, the number of iterations of the algorithm is at most exponential. Each iteration checks for an (at most) exponential number of pairs $(s, s')$ whether it can be obtained by sequential or parallel composition of given pairs. For the former the number of choices is quadratic, for the latter exponential. Altogether, the running time is at most exponential. □

### 4.3 Nonemptiness is EXPTIME-hard

To complete the proof of Theorem 7 it remains to show the following lemma.

**Lemma 11.** *Nonemptiness of bHMSCs is hard for EXPTIME.*

*Proof.* The proof is by a reduction from the intersection nonemptiness problem for deterministic top-down automata on binary trees:

Input: Tree automata $T_1, \ldots, T_n$ over a set of labels of rank zero or two.

Question: Is $\bigcap_{i=1}^{n} L(T_i) \neq \varnothing$?

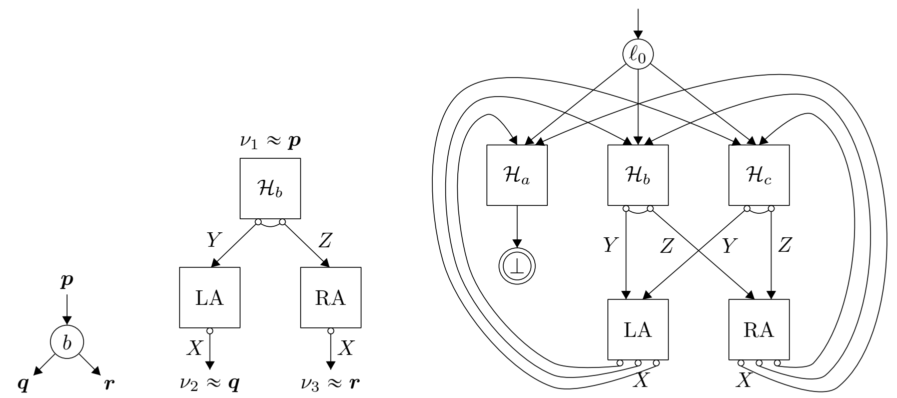
This problem is known[4] to be EXPTIME-complete [21].

The idea is to construct from given tree automata $T_1, \ldots, T_n$ a join-free bHMSC $\mathcal{H}$ (that is even *guarded*, cf. Section 7) such that any accepting run of $\mathcal{H}$ corresponds to a tree $t$ which has an accepting run in each of $T_1, \ldots, T_n$. More technically, each accepting run of $\mathcal{H}$ will correspond to a tree $t$ whose nodes are labelled, besides the label from $\Sigma$, with a vector $\boldsymbol{q} = (q_1, \ldots, q_n)$ of states from $Q$, one for each automaton $T_i$, such that, for every $i$, the projection of $t$ to the $i$-th components of these vectors yields an accepting run of $T_i$ on $t$.

For simplicity, we assume that all tree automata have the same set $Q$ of states. The initial state of every $T_j$ is $f$. Transitions are of the form $(b, q, q_\ell, q_r)$ for binary symbols $b$ (meaning that if the automaton assigns a state $q$ to a node labeled by $b$, then it will state $q_\ell$ to its left child and $q_r$ to its right child) and $(a, q)$ for 0-ary symbols $a$ (meaning that a leaf with label $a$ can be assigned the state $q$). The transitions of $T_i$ is denoted $\Delta_i$.

The bHMSC $\mathcal{H}$ has a main register $w$ and registers $x_p^j, y_p^j, z_p^j$, for every $p \in Q$ and $j \in \{1, \ldots, n\}$. For maintaining guardedness assumption it needs to have one more register $w'$. That is, its register set is $X \cup Y \cup Z$ where $X = \{w, w'\} \cup \{x_p^j \mid < j \leqslant n, p \in Q\}$, $Y = \{y_p^j \mid < j \leqslant n, p \in Q\}$ and $Z = \{z_p^j \mid < j \leqslant n, p \in Q\}$. Roughly,

[4] The statement of Theorem 4 in [21] does not mention the restriction to binary trees, but the proof uses only symbols of at most binary rank.



(a) A transition (b) partial run of the HMSC (c) Connections among the sub-bHMSCs

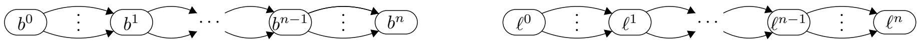
**Fig. 2.** Reduction

a defined register $x^j_p$ corresponds to automaton $T_j$ assuming state $p$ at the current node, a defined register $y^j_q$ corresponds to state $q$ of $T_j$ at its left child and a defined register $z^j_r$ corresponds to state $r$ of $T_j$ at its right child. Our reduction also makes sure that at most one $x^j_p$ (resp. $y^j_p$, $z^j_p$) is defined at any time for each $j \in \{1, \ldots, n\}$.

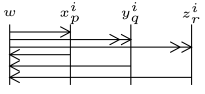
More formally, a register assignment $\nu : X \rightharpoonup \mathbb{P}$ *corresponds* to a vector $\boldsymbol{q} = (q_1, \ldots, q_n)$ of states from $Q$ if, for every $j \in \{1, \ldots, n\}$ and every $p \in Q$, it holds that $\nu(x^j_p)$ is defined if and only if $q_j = p$. In this case we write $\nu \approx \boldsymbol{q}$.

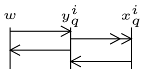
We construct $\mathcal{H}$ with the intention that every transition of the product tree automata can be simulated. To this end, $\mathcal{H}$ has a sub-bHMSC $\mathcal{H}_b$, for every binary symbol $b \in \Sigma$. If a run enters $\mathcal{H}_b$ with a register assignment $\nu_1$ such that $\nu_1 \approx \boldsymbol{p}$ and $Y$ and $Z$ are undefined at $\nu_1$, $\mathcal{H}_b$ will compute $\Delta(\boldsymbol{p}, b) = (\boldsymbol{q}, \boldsymbol{r})$ by defining the corresponding registers in $Y$ and $Z$. Then the run branches into two: the left branch "undefines" $X$ and $Z$ registers and the right branch "undefines" $X$ and $Y$ registers. Then the branches go through a left-adjust (LA) sub-BHMSC (resp, right-adjust (RA)) which will define the corresponding $X$ registers and "undefines" $Y$ (reps. $Z$). The leader is always $w$, except for RA, which has $w'$ as the leader. Thus the final register assignments of the branches yield $\nu_2 \approx \boldsymbol{q}$ and $\nu_3 \approx \boldsymbol{r}$ (see Figure 2(b)). Each of these sub-bHMSCs are described next.

$\mathcal{H}$ has a sub-bHMSC $\mathcal{H}_b$, for every binary symbol $b \in \Sigma$ (see Figure 3(a)). It has $n + 1$ location $b^0, \ldots b^n$. The initial location of $\mathcal{H}_b$ is $b^0$. For each transition $(b, p, q, r) \in \Delta_i$, it has an edge going from $b^{i-1}$ to $b^i$ labelled by a pMSC which defines $y^i_q$ and $z^i_r$ if $x^i_p$ is already defined. For this, $w$ first send a message to $x^i_p$ and then spawns $y^i_q$ and $z^i_r$. Then, for the sake of maintaining the guardedness property, $x^i_p$, $y^i_q$ and $z^i_r$ sends a message back to $w$. Thus when it reaches $b^n$, $Y$ and $Z$ contain a unique vector of states $\boldsymbol{q}$ and $\boldsymbol{r}$ respectively, if $\nu_1 \approx \boldsymbol{p}$ and $Y$ and $Z$ are undefined at $\nu_1$. Notice that the number of transitions from $b^{i-1}$ to



(a) $\mathcal{H}_b$ (b) LA

**Fig. 3.** There is a transition labeled 

from $b^{i-1}$ to $b^i$ iff $(b, p, q, r) \in \varDelta_i$. There is a transition labeled 

from $\ell^{i-1}$ to $\ell^i$ for every $q \in Q$.

$b^i$ is at most $|Q|$. Since we have a deterministic automata, at most one of these transitions is enabled.

Since $b^n$ is a fork transition with RA as a successor, and the leader of RA is $w'$, $w'$ need to be present in all MSCs labeling a transition from $b^{n-1}$ to $b^n$. For this, in the MSCs labeling a transition from $b^{n-1}$ to $b^n$, $w$ spawns $w'$.

Now we describe the sub-bHMSC LA (Figure 3(b)). If LA is reached at a register assignment with $X$ and $Z$ undefined and $Y$ corresponding to a vector $\boldsymbol{q}$ of states, then when it leaves LA, the registers $Y$ and $Z$ will be undefined, but the corresponding registers in $X$ will be defined. For this, LA has $n+1$ locations $\ell^0, \ldots \ell^n$. For each $0 < i \leqslant n$ and $q \in Q$, there is a transition from $\ell^{i-1}$ to $\ell^i$ labeled by a pMSC in which $y_q^i$ spawns $x_q^i$. Finally once it reaches $\ell^n$, it forks again to "undefine" the registers $X$ and $Z$. The target location is $\sigma^0$ for some $\sigma \in \varSigma$. Indeed there will be $|\varSigma|$ many fork transitions at $\ell^n$.

The right-adjust sub-bHMSC (RA) is similar, where $w'$ takes the role of $w$. The transitions leading to the last state of RA need to involve their next leader $w$. Thus in these transitions $w'$ spawns $w$ first, and $x_q^n$ and $z_q^n$ sends a message back to $w$ instead of $w'$.

Finally, there is a sub-bHMSC $\mathcal{H}_a$, for every nullary symbol $a \in \varSigma$. Like the previous sub-bHMSCs it consists of levels. In level $j$ there is one transition, for each transition $(a, p)$ of $T_j$. This transition is labelled by an MSC in which $w$ sends a message to $x_p^j$ and vice versa. This transition is enabled only if $x_p^j$ is defined. The last location of $\mathcal{H}_a$ is $\perp$ (i.e, $a^n = \perp$).

Finally, $\mathcal{H}$ has an additional location $\ell_0$ which is the initial location. It has, for every symbol $\sigma \in \varSigma$, a transition to location $\sigma^0$ with an MSC in which $w$ spawns $x_f^j$, for every $j \in \{1, \ldots, n\}$. A bHMSC for an alphabet $a, b, c$ with $a$ nullary and $b$ and $c$ binary will have the connections as shown in Figure 2(c).

It remains to show that $\mathcal{H}$ has an accepting run if and only if $\displaystyle\bigcap_{i=1}^{n} L(T_i) \neq \varnothing$. We first make the following key observation:
There is a transition of the form shown in Figure 2(a) in the product automaton $\displaystyle\bigtimes_{i=1}^{n} T_i$ if and only if the bHMSC $\mathcal{H}$ permits a partial run of the form Figure 2(b).

Indeed, if $\bigcap_{i=1}^{n} L(T_i) \neq \varnothing$, then there is tree $t$ whose edges can be labelled with vectors of states of the accepting runs of the automata $T_i$. From this labelled tree, we can obtain a partial run of the bHMSC $\mathcal{H}$ as explained in Figures 2(a) and 2(b). Since the tree $t$ is accepting, the vector $\boldsymbol{q}$ labeling a leaf $a$ allows $\mathcal{H}_a$ to end up in $\perp$. This run can be made initial by adding $\ell_0$ since the initial transition gives a register assignement $\nu \approx \boldsymbol{f}$. Since all leaves are labelled by unary symbols in the tree $t$, the corresponding run of $\mathcal{H}$ has all branches ending up in $\perp$, and hence accepting.

To show the converse, if $\mathcal{H}$ has an accepting run, then a tree $t$ can be extracted (as in Figure 2(a) and Figure 2(b)). Indeed the register assignment at $\sigma^0$ for $\sigma \in \Sigma$ allows labeling the tree with vectors of states as well. This gives an accepting run of $\bigtimes_{i=1}^{n} T_i$ on $t$. Thus $\bigcap_{i=1}^{n} L(T_i) \neq \varnothing$. □

## 4.4 Nonemptiness of Sequential bHMSCs

Next, we consider the special case of sequential bHMSCs.

**Theorem 12.** *Nonemptiness of sequential bHMSCs is NP-complete.*

*Proof.* For checking nonemptiness, it is not just enough to find a path from the initial location to the final location: We need to ensure that, for every transition $(\ell, M, \ell')$ taken in this path, the registers corresponding to $\mathit{Free}(M) \cup \mathit{MsgPar}(M)$ are already defined. Moreover, taking a loop might define some registers which were undefined before, thus enabling more transitions. However, notice that taking the same loop twice will not define new registers. Thus, for checking nonemptiness we will rather guess a sequence of transitions from the initial location to a final location as well as the set of defined registers in the intermediate configurations.

*Claim.* A bHMSC $\mathcal{H}$ is nonempty if and only if there is a sequence

$$\ell_0, X_0, t_1, \ell_1, X_1, t_2, \ell_2, X_2, \dots, t_n, \ell_n, X_n$$

such that

1. $\ell_i \in L$ and $X_i \subseteq X$ for all $0 \leqslant i \leqslant n$, and $t_i \in \Delta$ for all $1 \leqslant i \leqslant n$,
2. $\ell_0 \in L_{\text{init}}$, $X_0 = \{x_0\}$ and $\ell_n \in L_{\text{acc}}$ and
3. each transition $t_i$ is of the form $(\ell_{i-1}, M, \ell_i)$ with
   (a) $\mathit{Free}(M) \cup \mathit{MsgPar}(M) \subseteq X_{i-1}$ and
   (b) $X_i = X_{i-1} \cup \mathit{Bnd}(M)$

The claim holds since if there is an accepting run, we could easily abstract such a sequence from the run. On the other hand if there is such a sequence, we can extend the sequence to an accepting run.

Notice that the sequence $(X_i)_i$ is monotonically non-decreasing. Moreover, we can truncate loops in this sequence (that is $\ell_i = \ell_j$ and $X_i = X_j$ for some $i < j$). Thus if there exists a witness sequence, then there exists one which is polynomial in the size of the bHMSC $\mathcal{H}$.

The NP procedure for nonemptiness of the HMSC $\mathcal{H}$ simply guesses such a sequence and verifies the conditions listed above in polynomial time.

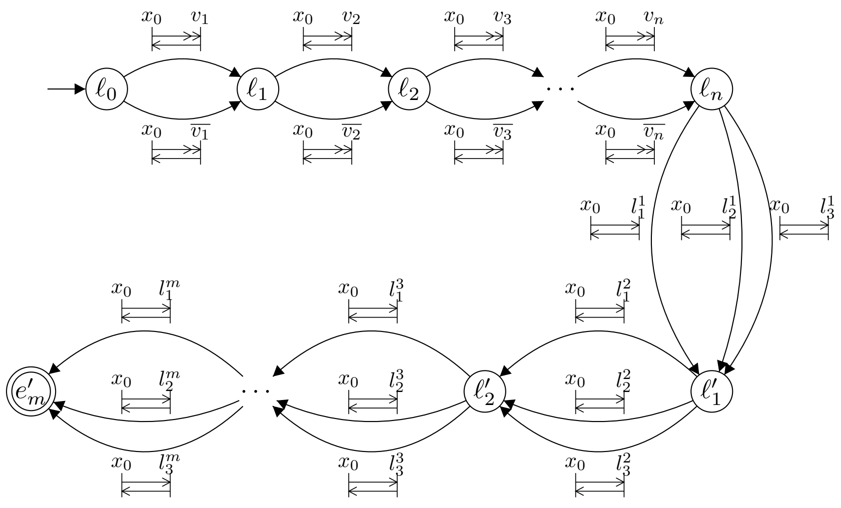


**Fig. 4.** Reduction from 3-CNF-SAT to nonemptiness of guarded sequential HMSC.

Now we show the NP hardness by reducing 3-CNF-SAT to the nonemptiness problem of HMSC. The problem 3-CNF-SAT which is given below is a well known NP-Complete problem. Let $V = \{v_1, \ldots v_n\}$ be a set of propositional variables. By $\overline{V}$, we denote the set $\{\overline{v_1}, \ldots, \overline{v_n}\}$, the set of negations of propositional variables. Let $\mathsf{Lit} = V \cup \overline{V}$ be the set of literals.

Input: $\varphi \equiv \bigwedge_{i=1}^{m} C_i$ where $C_i = l_1^i \vee l_2^i \vee l_3^i$ and $l_j^i \in \mathsf{Lit}$ for $1 \leqslant j \leqslant 3$.
Question: Is there a satisfying truth assignment of the variables $V$ such that $\varphi$ evaluates to `true`?

Our reduction is as follows. On the input $\varphi$, we construct an HMSC $\mathcal{H}_\varphi$ as given in the Figure 4. Remember that $l_j^i$ is actually some $v_k$ or $\overline{v_k}$. Thus the HMSC $\mathcal{H}_\varphi$ uses $2n+1$ registers: $X = \{x_0\} \cup \mathsf{Lit}$. On going from location $\ell_{i-1}$ to location $\ell_i$ the run defines one and only one of the two registers $v_1$ and $\overline{v_1}$. Thus on reaching location $\ell_n$, the configuration corresponds to a unique truth assignment: The register $v_i$ is defined if and only if the propositional variable $v_i$ is set to `true` by the truth assignment, and the register $\overline{v_i}$ is defined if and only if the propositional variable $v_i$ is set to `false` by the truth assignment. Thus there is a run from $\ell_0$ to $\ell_n$ corresponding to every truth assignment, and there is a truth assignment corresponding to every run from $\ell_0$ to $\ell_n$. The run can be extended to reach the location $\ell'_1$ if and only if the current truth assignment satisfies the clause $C_1$. Inductively, the run can be extended to reach the location $\ell'_i$ if and only if all the clauses $C_1, \ldots C_i$ are satisfiable by the current truth assignment. Hence there is an accepting run of the HMSC $\mathcal{H}_\varphi$ if and only if $\phi$ is satisfiable. This proves the NP-hardness. □

We note that nonemptiness of the fork-and-join grammars from [13] is decidable in polynomial time. This is due to the fact that their model does not allows for a separation of spawns from messages. In fact, the complexity comes from checking a certain well-formedness arising from this distinction.

## 5 Dynamic Communicating Automata

In this section, we introduce an extension of the model of dynamic communicating automata as presented in [3]. 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 [3], 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 13 (DCA).* A *dynamic communicating automaton (DCA)* over the ranked message alphabet $A$ is a tuple $\mathcal{D} = (S, X, S_{\text{init}}, S_{\text{acc}}, \Delta)$ where $S$ is a finite set of *states* with initial states $S_{\text{init}} \subseteq S$ and accepting states $S_{\text{acc}} \subseteq S$, $X$ is a nonempty finite set of registers, and $\Delta$ is the set of transitions. A transition is of the form $(s, \alpha, s')$ where $s, s' \in S$, and $\alpha$ is an action, possibly a *send action* $!_x(a(x_1, \dots, x_n))$, a *receive action* $?_y(a(y_1, \dots, y_n))$, or a *spawn action* $x := \mathsf{spawn}(s, z)$, where $x, z \in X$, $y \in X \cup \{*\}$, $s \in S$, $a(x_1, \dots, x_n) \in A(X \cup \{\mathsf{self}\})$, and $a(y_1, \dots, y_n) \in A(X \cup \{\bot\})$ such that, for all $i, j \in \{1, \dots, n\}$, $y_i = y_j \in X$ implies $i = j$.

When a process executes $!_x(a(\overline{x}))$ with $\overline{x} = (x_1, \dots, x_n)$, 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 $y_i = \bot$. 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 $(\sigma, \tau)$, where $\sigma : E \to S$ and $\tau : E \to [X \rightharpoonup \mathbb{P}]$, respecting the following conditions:

- $\sigma_{\mathit{init}(M)} \in S_{\text{init}}$,
- $\tau_{\mathit{init}(M)}$ 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 $\Delta$ contains a local transition $\sigma_{e_1} \xrightarrow{x := \mathsf{spawn}(s,y)} \sigma_{e_2}$

$$\sigma_{e_1} \xrightarrow{x := \mathsf{spawn}(s,y)} \sigma_{e_2}$$

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 $\Delta$ contains transitions $\sigma_{e_1} \xrightarrow{!_x(a(\overline{x}))} \sigma_{e_2}$ and $\sigma_{f_1} \xrightarrow{?_y(a(\overline{y}))} \sigma_{f_2}$

$$\sigma_{e_1} \xrightarrow{!_x(a(\overline{x}))} \sigma_{e_2} \quad \text{and} \quad \sigma_{f_1} \xrightarrow{?_y(a(\overline{y}))} \sigma_{f_2}$$

with $\overline{x} = (x_1, \ldots, x_n)$ and $\overline{y} = (y_1, \ldots, y_n)$ such that, letting

$$p_i = \begin{cases} \tau_{e_1}(x_i) & \text{if } x_i \in X \\ pid(e_1) & \text{if } x_i = \mathsf{self}\,, \end{cases}$$

we have $\tau_{e_2} = \tau_{e_1}$, $\mu(e_2, f_2) = a(p_1, \ldots, p_n)$, $\tau_{e_1}(x) = pid(f_1)$, $\big(y = * \text{ or } \tau_{f_1}(y) = pid(e_1)\big)$, and

$$\tau_{f_2}(z) = \begin{cases} p_i & \text{if } z = y_i \text{ and } y_i \neq \bot \\ \tau_{f_1}(z) & \text{if } z \notin \{y_1, \ldots, y_n\}\,. \end{cases}$$

Here, $\sigma_e$ and $\tau_e$ 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 $\tau_e$ on all other values.

The run $(\sigma, \tau)$ is accepting if $\sigma(e) \in S_{\mathrm{acc}}$ for all $e \in \max(M)$. By $L(\mathcal{D})$, we denote the set of MSCs $M$ over $A$ and $\mathbb{P}$ such that there is an accepting run of the DCA $\mathcal{D}$ over $M$. Note that $L(\mathcal{D})$ is closed, i.e., $L(\mathcal{D}) = [L(\mathcal{D})]$. Emptiness is undecidable for CA, and consequently also for DCA.

Undecidability of nonemptiness for DCA follows from the undecidability for CA. However, we will present a proof that even works when we restrict to *channel bounded MSCs* (in which case nonemptiness is decidable for CA).

**Theorem 14.** *The nonemptiness problem for DCA is undecidable.*

*Proof.* The proof is by a reduction from the nonemptiness problem for Minsky two counter machines (2-CM) [18]. A 2-CM is a nondeterministic finite automaton equipped with two counters $\mathcal{C}_1$ and $\mathcal{C}_2$. Every transition of the automaton is associated with an action from $\{\mathsf{inc}_i, \mathsf{dec}_i, \mathsf{ifzero}_i \mid 1 \leqslant i \leqslant 2\}$. For $i \in \{1, 2\}$ action $\mathsf{inc}_i$ increments and $\mathsf{dec}_i$ decrements the value of counter $C_i$ by 1. The action $\mathsf{dec}_i$ can only be applied if the value of $C_i$ is not 0 and $\mathsf{ifzero}_i$ can only be applied if the value of $C_i$ is 0. At the beginning the values of both counters are 0. It is well-known that the nonemptiness problem for 2-CM is not decidable [18].

Given a 2-CM one can construct a DCA such that the 2-CM is nonempty if and only if the DCA is nonempty. The main idea of the reduction is to simulate the behavior of the 2-CM by representing counter values by process IDs (pids). To be more precise, a configuration of the 2-CM is represented by a *main process* with two registers $x_1$ and $x_2$ and a sequence of *counter processes* where every two neighbored processes are linked to each other by their register assignments. The registers $x_1$ and $x_2$ of the main process contain the pids of some counter processes. The pid in $x_1$ stands for the value of counter $C_1$ and the pid in $x_2$ for the value of $C_2$. Every counter process has three registers from which the first one points to the main process and the other two point to some preceding and some successive counter process. Exemplarily we describe how the behavior of counter $C_1$ is simulated by the DCA, the simulation of $C_2$ is analogue. At the beginning, in register $x_1$ of the main process the pid of a special counter process called *bottom process* is stored. It represents counter value 0. Whenever $C_1$ is incremented, the main process sends an inc-message to the counter process $p$ stored in $x_1$. If there exists some successor for $p$, then $p$ sends the pid of the successor to the main process, if not, $p$ spawns a new successor and sends the pid of the new process to the main process. The latter stores the received pid in $x_1$. Before executing a decrement the main process asks the process in $x_1$ whether it is the bottom process or not. If yes, the computation stops. If not, the main process sends a dec-message to the process in $x_1$ who afterwards sends back the pid of its predecessor. The main process also takes care that the action ifzero is executed only if the process stored in $r_1$ is not the bottom process. $\square$

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 $b \in B$, $arity(h(b)) \leqslant arity(b)$. 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 $\mu'(e, f) = h(b)(p_1, \ldots, p_{arity(h(b))})$ whenever $\mu(e, f) = b(p_1, \ldots, p_n)$. The mapping is then further extended to sets of MSCs as expected.

*Definition 15 (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 $[L] = h(L(\mathcal{D}))$.

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 16.* The MSC language $\{M_1\}$ (see Figure 5) is not implementable, as process 1 does not know 2 when sending message $b$. However, $\{M_2\}$ is implementable, 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 has to accept $M_4$. 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. For $\{M_4\}$, there is an alternative: the first message is refined to contain the pid of the sender of the second message (i.e, 3) so that 0 can use a selective receive for the second message.

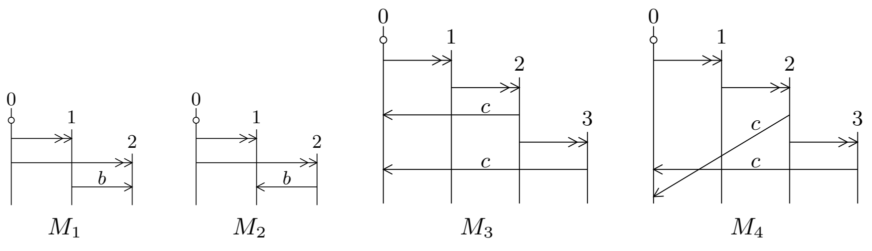


**Fig. 5.** Realizability vs. Implementability

## 6 Executability of bHMSCs

An accepting run of a bHMSC generates an MSC. However, this MSC need not be implementable always, as Example 16 shows. Unfortunately, implementability (and also realizability) is undecidable for bHMSCs, which follows from undecidability for HMSCs over a fixed finite set of processes [11, 1].

**Theorem 17 (cf. [11, 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 18 (executability).* Let $M \in \mathbb{MSC}(A, \mathbb{P})$. A message $(e, f) \in \lhd_{\mathsf{msg}}$ of $M$ with message contents $a(p_1, \ldots, p_{arity(a)})$ is *executable* if $q \rightsquigarrow_M e$, for every $q \in \{pid(f), p_1, \ldots, p_{arity(a)}\}$. 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 5, $M_2, M_3, M_4$ are executable, while $M_1$ is not.

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 a decidable property:

**Theorem 19.** *Executability of bHMSCs is EXPTIME-complete. The lower bound already holds for bHMSCs that are join-free.*

From the algorithmic perspective, if an MSC $M = M_1 \circ M_2$ where $M_1$ is an MSC and $M_2$ is a partial MSC, it is desirable to infer the executability of $M$ from that of $M_1$ and $M_2$. However, $M_2$ is a partial *MSC*, and it is not evident what the executability of $M_2$ should mean. When a partial MSC is concatenated to another MSC (here $M_1$), the executability also depends on the awareness relation *induced* by the latter.

We denote such an *awareness relation* by $K \subseteq \mathbb{P} \times \mathbb{P}$. Intuitively, $K(p,q)$ means that the process $p$ is aware of another process $q$. Naturally, we require $K$ to be reflexive.

Now we can analyse the executabiltiy of a partial MSC under the assumption of an awareness relation $K$. A partial MSC $M = (E, \lhd, \lambda, \mu)$ from $\mathbb{MSC}(A, \mathbb{P})$ is executable at $K$ if for every message $(e,f) \in \lhd_{\mathsf{msg}}$ with message contents $a(p_1, \ldots, p_{arity(a)})$, and for every $q \in \{pid(f), p_1, \ldots, p_{arity(a)}\}$, either 1) $q \rightsquigarrow_M e$, or 2) there is a $q'$ such that $K(q', q)$ and $q' \rightsquigarrow_M e$.

By $K$ and $M$ a *resulting awareness relation* is induced: the awareness relation which results from the execution of $M$ at $K$. We write $K \xrightarrow{M} K'$ if $K'$ is the resulting awareness relation. It is easy to see that $K'$ is uniquely determined as follows: $K' = K \cup \{(p,q) \mid q \rightsquigarrow_M \max_p(M)$ or there is a $q'$ such that $K(q',q)$ and $q' \rightsquigarrow_M \max_p(M)\}$.

The following remarks are easy consequences from the above definitions.

*Remark 20.* A partial MSC $M$ is executable at $K$ if and only if $M$ is executable at $K_{\restriction Free(M)}$.

*Remark 21.* A partial MSC $M = M_1 \circ M_2$ is executable at $K$ if and only if there is $K_1$ such that $M_1$ is executable at $K$, $K \xrightarrow{M_1} K_1$ and $M_2$ is executable at $K_1$. Furthermore, $K \xrightarrow{M} K'$ if $K_1 \xrightarrow{M_2} K'$.

*Remark 22.* A partial MSC $M = M_1 || \cdots || M_n$ (in other words $M = M_1 \circ \cdots \circ M_n$ where $M_i$ and $M_j$ do not share any processes if $i \neq j$) is executable at $K$ if and only if, $M_i$ is executable at $K$, for each $1 \leqslant i \leqslant n$. Furthermore, if $K \xrightarrow{M_i} K_i'$, then $K \xrightarrow{M} K'$ where $K' = \bigcup_i K_i'$.

We next adapt the above notions to the symbolic framework in order to characterize the symbolic counterparts of executable runs.

The edges of a symbolic run are labelled by partial MSCs over the register set $X$. From Remark 20, an awareness relation on $Free(M)$ is sufficient to analyse the executability of a partial MSC $M$. Consequently, for symbolic runs, we will employ symbolic awareness relations $\mathsf{SK} \subseteq X \times X$.

A symbolic awareness relation $SK$ for a partial MSC $M$ over a register set $X$ is just an awareness relation of $M$, that is a subset of $X \times X$. However, some care is needed when we define awareness relations for symbolic runs, as registers can be reused inside in a symbolic run and thus the same register might refer to different processes. Awareness can thus no longer be expressed by a simple reachability condition; instead “rewriting” of registers needs to be taken into acount. We are going to describe how the awareness relation for a symbolic run can be computed. We will then prove that it is exactly the same as the *induced symbolic awareness relation* $\mathsf{symb}_\nu(K)$, that is, the set of all pairs $(x,y) \in X \times X$ for which $(\nu(x), \nu(y)) \in K$, for a run with initial register assignment $\nu : X \mapsto \mathbb{P}$ and initial awareness relation $K \subseteq \mathbb{P} \times \mathbb{P}$.

For a single partial MSC $M$, the set of bound registers $Bnd(M)$ and the paths of $M$ ($\rightsquigarrow_M$) are precisely known. For MSCs resulting from a bHMSC, we do not need to keep track of all paths but it will be crucial to know which processes at the end of $M$ are aware of which processes at the beginning of $M$.

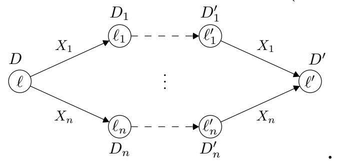
To this end, we first define, for every partial MSC $M$, the flow relation $\mathsf{flw}_M$ as the set of all pairs $(p, q)$ such that $p = q$ and $p \notin \mathit{Pids}(M)$, or $q \in \mathit{Free}(M)$ and there is a path from the minimal $q$ event of $M$ to the maximal $p$ event of $M$. That is, $\mathsf{flw}_M = \{(p, p) \in \mathbb{P} \times \mathbb{P} \mid p \notin \mathit{Pids}(M)\} \cup \{(p, q) \in \mathbb{P} \times \mathbb{P} \mid q \in \mathit{Free}(M)$ and $q \rightsquigarrow_M \max_q(M)\}$.

For a run $G$ with initial register assignment $\nu \in \mathcal{R}(X)$ and final register assignment $\nu' \in \mathcal{R}(X)$ we define $\mathsf{flw}_G \overset{\text{def}}{=} \{(x, y) \in X \times X \mid (\nu'(x), \nu(y)) \in \mathsf{flw}_{M(G)}\}$. Furthermore, we define the set $B_G$ of "rewritten registers" by $B_G \overset{\text{def}}{=} \{x \in X \mid \nu'(x) \neq \nu(x)\}$. Here, $\nu'(x) \neq \nu(x)$ holds in particular, if $\nu'(x)$ is defined but $\nu(x)$ is undefined.

Now we are prepared to define the *executability of symbolic runs*. Thanks to Lemma 8 we can do this by induction on the structure of an underlying run $G$. We also define the resulting "effect tuple" $(SK', \mathsf{flw}', B')$ simultaneously. We write $SK \xrightarrow{S} (SK', \mathsf{flw}', B')$ to denote that $S$ is executable at $SK$ and its effect is $(SK', \mathsf{flw}', B')$.

*Definition 23.* – Let $G = G(\delta, \nu, \nu')$ for register assignments $\nu, \nu' \in \mathcal{R}(X)$ and $\delta = \ell \xrightarrow{M} \ell'$ be a sequential transition and $SK_1, SK_2 \subseteq X \times X$ symbolic awareness relations. If $G$ is executable at $K_1$ with effect $K_2$ then we say that $S = \mathsf{symb}(G)$ is executable at $\mathsf{symb}_\nu(K_1)$ and define its effect as $(\mathsf{symb}_{\nu'}(K_2), \mathsf{flw}_G, B_G)$.

– Let $S_1$ and $S_2$ be symbolic runs, executable at $SK_1$ with effect $(SK_2, \mathsf{flw}_1, B_1)$ and at $SK_2$ with effect $(SK_3, \mathsf{flw}_2, B_2)$, respectivley. Then we say that $S_1 \circ S_2$ is executable at $SK_1$ with effect $(SK_3, \mathsf{flw}_2 \circ \mathsf{flw}_1, B_1 \cup B_2)$.

– Let $S =$ 

. $S$ is executable at $SK$ if for each $1 \leqslant i \leqslant n$, $S_i$ is executable at $\mathsf{SK}_{\restriction X_i}$.

Moreover, if $\mathsf{SK}_{\restriction X_i} \xrightarrow{S_i} (SK'_i, \mathsf{flw}'_i, B'_i)$ for each $1 \leqslant i \leqslant n$, then $SK \xrightarrow{S} (SK', \mathsf{flw}', B')$, where

- $B' = \bigcup_i (B'_i \cap X_i)$;
- $\mathsf{flw}' = \{(x, x) \mid SK(x, x), x \notin B'\} \cup \bigcup_i (\mathsf{flw}'_i)_{\restriction X_i}$;
- $SK'$ is the set of all pairs $(x, y)$, for which one of the following conditions holds:
  (i) $x \in X_0$, $(x, y) \in SK$ and $y \notin B'$;
  (ii) for some $i \in \{1, \ldots, n\}$, $x \in X_i$, for some $z$, $(x, z) \in \mathsf{flw}'_i$, $(z, y) \in SK$, and $y \notin B'$;

(iii) for some $i \in \{1, \ldots, n\}$, $x \in X_i$, $y \in X_i$, and $(x, y) \in SK'_i$;

We say that a symbolic run is executable if it is executable at $\{(x_0, x_0)\}$.

It should be noted that in the above definition the information whether a register is defined is implicitly given by the symbolic awareness relations: $x$ is defined if $SK(x, x)$. Furthermore, and crucial for the below algorithm, the definition makes use of concrete MSCs only in the base case.

Now, we can state the characterization of executable runs in terms of their symbolic runs.

**Lemma 24.** *A bHMSC $\mathcal{H}$ is executable if and only if every accepting symbolic run of $\mathcal{H}$ is executable.*

*Proof.* To show the lemma, we prove a slightly stronger statement.

*Claim.* Let $G = (V, R, loc, reg, \rho)$ with initial register assignment $\nu$ and final register assignment $\nu'$ and let $K$ be an awareness relation. Then the following statements hold.

(1) If $\mathsf{symb}(G)$ is executable at $\mathsf{symb}_\nu(K)$ then $G$ is executable at $K$.
(2) If $G$ is executable at $K$ then $\mathsf{symb}(G)$ is executable at $\mathsf{symb}_\nu(K)$.
(3) If $K \xrightarrow{G} K'$ then $\mathsf{symb}_\nu(K) \xrightarrow{\mathsf{symb}(G)} (\mathsf{symb}_{\nu'}(K'), \mathsf{flw}_G, B_G)$.

Statements (1) and (2) yield the statement of the lemma as follows.

$$\begin{aligned}
\mathcal{H} \text{ is executable} &\Leftrightarrow \text{every MSC from } L(\mathcal{H}) \text{ is executable} \\
&\Leftrightarrow \text{every accepting run } G \text{ of } \mathcal{H} \text{ with } reg(in(G)) = \{x_0 \mapsto p\} \\
&\quad\ \text{for some process } p \in \mathbb{P} \text{ is executable at } \{(p, p)\} \\
&\Leftrightarrow \text{every accepting symbolic run } S(G) \text{ of } \mathcal{H} \text{ is} \\
&\quad\ \text{executable at } \{(x_0, x_0)\} \\
&\Leftrightarrow \text{every accepting symbolic run } S(G) \text{ of } \mathcal{H} \text{ is executable}
\end{aligned}$$

We finally prove the claim by induction on $G$.

The base case is when $G = G(\delta, \nu, \nu')$ for register assignments $\nu, \nu' \in \mathcal{R}(X)$ and a sequential transition $\delta = \ell \xrightarrow{M} \ell'$ with $\nu \xrightarrow{M} \nu'$.

Let $K$ be an awareness relation. By definition of symbolic executability, $\mathsf{symb}(G)$ is executable at $\mathsf{symb}_\nu(K)$, if and only if $G$ is executable at $K$. Thus, (1) and (2) hold. Likewise, statement (3) holds by definition of symbolic effects.

For the inductive step, we first consider the case that there are runs $G_1, G_2$ such that $G = G_1 \circ G_2$. Here, we write $\nu_1$ for $\nu$, the initial register assignment of $G$, and $\nu_3$ for $\nu'$, its final register assignment. Likewise, we write $K_1$ for $K$. By definition of sequential composition, $\nu_1$ is also the initial assignment of $G_1$ and $\nu_3$ the final assignment of $G_2$. Furthermore, there is an intermediate assignment $\nu_2$, which is final for $G_1$ and initial for $G_2$.

Towards (1) let us assume that $S(G)$ is executable at $SK_1 = \mathsf{symb}_{\nu_1}(K_1)$, for some awareness relation $K_1$. By definition, we can conclude that $S(G_1)$ is executable at $SK_1$ with some effect $(SK_2, \mathsf{flw}_1, B_1)$ and $S(G_2)$ is executable at $SK_2$ with some effect $(SK_3, \mathsf{flw}_2, B_2)$. By induction, $G_1$ is executable at $K_1$ and $SK_2 = \mathsf{symb}_{\nu_2}(K_2)$, where $K_2$ is the effect of $G_1$ on $K_1$. Likewise, $G_2$ is executable at $K_2$, by induction. That $G$ is executable at $K_1$ follows by Remark 21.

On the other hand, again by Remark 21, if $G$ is executable at awareness relation $K_1$ then $G_1$ is executable at $K_1$ as well with some effect $K_2$ at which $G_2$ is executable. By induction, $S(G_1)$ is executable at $\mathsf{symb}_{\nu_1}(K_1)$ with effect $(\mathsf{symb}_{\nu_2}(K_2), \mathsf{flw}_{G_1}, B_{G_1})$ and $S(G_2)$ is executable at $\mathsf{symb}_{\nu_2}(K_2)$ with effect $(\mathsf{symb}_{\nu_3}(K_3), \mathsf{flw}_{G_2}, B_{G_2})$. Thus, by definition, $S(G)$ is executable at $\mathsf{symb}_{\nu_1}(K_1)$, yielding (2).

For (3), let us assume $K_1 \xrightarrow{G} K_3$. By induction, we have $\mathsf{flw}_1 = \mathsf{flw}_{G_1}$, $\mathsf{flw}_2 = \mathsf{flw}_{G_2}$, $B_1 = B_{G_1}$ and $B_2 = B_{G_2}$. By (2), $S(G)$ is executable at $SK_1$ and thus, by definition, $SK_1 \xrightarrow{S(G)} (SK_3, \mathsf{flw}_{G_2} \circ \mathsf{flw}_{G_1}, B_{G_1} \cup B_{G_2})$. Thus, to yield (3) it suffices to show that $\mathsf{flw}_G = \mathsf{flw}_{G_1} \circ \mathsf{flw}_{G_2}$ and $B_G = B_{G_1} \cup B_{G_2}$.

For the proof of $\mathsf{flw}_G = \mathsf{flw}_{G_2} \circ \mathsf{flw}_{G_1}$ first assume $(x, y) \in \mathsf{flw}_G$. By definition one of the following cases holds:

- (i) $\nu_1(x) = \nu_3(y)$ and $\nu_1(y)$ does not occur in $M(G)$
- (ii) $\nu_1(y) \in \mathit{Free}(M(G))$ and $\nu_1(y) \rightsquigarrow_{M(G)} \max_{\nu_3(x)}(M(G))$.

First, we look at the first case. By definition of runs it must hold $x = y$ and $\nu_1(x) = \nu_2(x) = \nu_3(x)$. As $\nu_1(x)$ does not occur in $M(G)$ at all, it follows $(x, x) \in \mathsf{flw}_{G_1}$ and $(x, x) \in \mathsf{flw}_{G_2}$, thus $(x, x) \in \mathsf{flw}_{G_2} \circ \mathsf{flw}_{G_1}$. Case (ii) has three subcases. Either

- (a) the path from $\nu_1(y)$ to $\max_{\nu_3(x)}$ starts end ends in $M(G_1)$, or
- (b) it starts and ends in $M(G_2)$, or
- (c) it starts in $M(G_1)$ and ends in $M(G_2)$.

We first look at case (a). As $\nu_1(y) \in \mathit{Free}(M(G))$ it holds also $\nu_1(y) \in \mathit{Free}(M(G_1))$. Moreover, as $\nu_3(x)$ occurs in $M(G_1)$, by definition of runs it must hold $\nu_2(x) = \nu_3(x)$. Thus, we get $\nu_1(y) \in \mathit{Free}(M(G_1))$ and $\nu_1(y) \rightsquigarrow_{M(G_1)} \max_{\nu_2(x)}(M(G_1))$ which means $(x, y) \in \mathsf{flw}_{G_1}$. As $\nu_2(x) = \nu_3(x)$ and $\nu_2(x)$ does not occur in $M(G_2)$ we get $(x, x) \in \mathsf{flw}_{G_2}$. Together with $(x, y) \in \mathsf{flw}_{G_1}$ it follows $(x, y) \in \mathsf{flw}_{G_2} \circ \mathsf{flw}_{G_1}$.

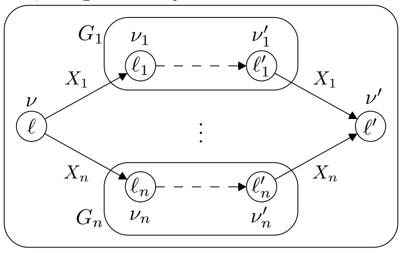
Case (b) is analogue. As $\nu_1(y)$ occurs in $M(G_2)$ it must hold $\nu_1(y) = \nu_2(y)$ and as it does not occur in $M(G_1)$ we get get $(y, y) \in \mathsf{flw}_{G_1}$. Moreover, observe that from $\nu_1(y) = \nu_2(y) \in \mathit{Free}(M(G))$ and $\nu_1(y) \in \mathit{Pids}(M(G_2))$ it follows $\nu_2(y) \in \mathit{Free}(M(G_2))$. Furthermore, as $\nu_1(y) = \nu_2(y)$ and the path $\nu_1(y) \rightsquigarrow_{M(G)} \max_{\nu_3(x)}(M(G))$ starts and ends in $M(G_2)$, we get $\nu_2(y) \rightsquigarrow_{M(G_2)} \max_{\nu_3(x)}(M(G_2))$, thus $(x, y) \in \mathsf{flw}_{G_2}$. Together with $(y, y) \in \mathsf{flw}_{G_1}$ it follows $(x, y) \in \mathsf{flw}_{G_2} \circ \mathsf{flw}_{G_1}$.

For case (c) observe that there has to be some $z$ such that $\nu_1(y) \rightsquigarrow_{M(G_1)} \max_{\nu_2(z)}(M(G_1))$, $\nu_2(z) \in \mathit{Free}(M(G_2))$ and $\nu_2(z) \rightsquigarrow_{M(G_2)} \max_{\nu_3(x)}(M(G_2))$. Thus, $(z, y) \in \mathsf{flw}_{G_1}$ and $(x, z) \in \mathsf{flw}_{G_2}$. It follows $(x, y) \in \mathsf{flw}_{G_2} \circ \mathsf{flw}_{G_1}$.

For the other direction, namely $(x,y) \in \mathsf{flw}_{G_2} \circ \mathsf{flw}_{G_1} \Rightarrow (x,y) \in \mathsf{flw}_G$, the main observation is that two pathes $\nu_1(y) \rightsquigarrow_{M(G_1)} \max_{\nu_2(z)}(M(G_1))$ and $\nu_2(z) \rightsquigarrow_{M(G_2)} \max_{\nu_3(x)}(M(G_2))$ can be put together to a single path $\nu_1(y) \rightsquigarrow_{M(G)} \max_{\nu_3(x)}(M(G))$.

To show $B_G = B_{G_1} \cup B_{G_2}$, one observes that a register is "rewritten" in $G$ if and only if it is rewritten in $G_1$ or in $G_2$.

It remains to prove the inductive step for the case that $G = FJ(\delta, \nu, G_1, \dots, G_n, \nu')$ for a fork-and-join transition $\delta$. Let, for each $i$, $\nu_i$ and $\nu'_i$ be the initial and final register assigments of $G_i$, respectively.



We first assume that $S(G)$ is executable at $\mathsf{symb}_\nu(K)$, for some awareness relation $K$. By definition, this is the case if each $S(G_i)$ is executable at $\mathsf{symb}_\nu(K_i)_{\upharpoonright X_i} = \mathsf{symb}_{\nu_i}(K)$. By induction, each $G_i$ is executable at $K_i \overset{\text{def}}{=} K_{\upharpoonright \mathit{Free}(M(G_i))}$. By definition, $\mathit{Free}(M(G_i)) \subseteq \mathrm{dom}(\nu_i)$, and thus $K_i \subseteq K$. Hence, by Remark 20, for every $i \in \{1, \dots, n\}$, $G_i$ is executable at $K$ and thus $G$ is executable at $K$, hence (1) holds.

Statement (2) follows similarly: let us assume that $G$ is executable at $K$. Then each subrun $G_i$ needs to be executable at $K$. By induction, each $S(G_i)$ is executable at $\mathsf{symb}_\nu(K)$ and therefore at $\mathsf{symb}_{\nu_i}(K_i) = \mathsf{symb}_\nu(K)_{\upharpoonright X_i}$. It follows by definition that $S(G)$ is executable at $\mathsf{symb}_\nu(K)$.

It remains to show (3). Let thus $K'$ be the effect of $G$ on $K$ and $(\mathsf{SK}', \mathsf{flw}', B')$ the effect of $S(G)$ on $SK \overset{\text{def}}{=} \mathsf{symb}_\nu(K)$. We first show that for two registers $x, y \in \mathrm{dom}(\nu')$, it holds $(\nu'(x), \nu'(y)) \in K'$ if and only if $(x,y) \in SK'$. In the following, let $M'$ denote $\nu'(M(G))$ and let $M'_i$ denote $\nu'_i(M(G_i))$, for $i \in \{1, \dots, n\}$.

Let thus, $x, y$ be registers with $(\nu'(x), \nu'(y)) \in K'$. We consider the three possible cases and conclude each time that $(x,y) \in SK'$:

- $(\nu'(x), \nu'(y)) \in K$: as $\nu'(x)$ and $\nu'(y)$ are already present in $K$, $x$ and $y$ are both not in $B'$. Thus, $\nu(x) = \nu'(x)$ and $\nu(y) = \nu'(y9$ and therefore $(x,y) \in SK$. We distinguish two subcases. If $x \in X_0$, then $(x,y) \in SK'$ by (i). Otherwise, $x \in X_i$, for some $i$ and, as $x \notin B'$, $(x,x) \in \mathsf{flw}'_i$, thus, $(x,y) \in SK'$ by (ii) with $z = x$.
- $\nu'(y) \rightsquigarrow_{M'} \max_{\nu'(x)}(M')$: in this case, the path is in some $M'_i$ and thus $(x,y) \in SK'_i$ and $(x,y) \in SK'$ by (iii).
- for some process $q'$, $K(q', \nu'(y))$ and $q' \rightsquigarrow_M \max_{\nu'(x)}(M)$: in this case there is a register $z$ with $\nu'(z) = q'$ such that $x, z$ are in some $X_i$ and $(\nu'(z), \nu'(y) \in K$. By definition $(x,z) \in \mathsf{flw}'_i$ and $(z,y) \in \mathsf{SK}$, and therefore $(x,y) \in \mathsf{SK}'$ by (ii).

Let us now assume $(x, y) \in SK'$. We distinguish three cases.

(i) $x \in X_0$, $(x, y) \in SK$ and $y \notin B'$: By definition of $SK$, $(\nu(x), \nu(y)) \in K$. However, as $x \in X_0$ and $y \notin B'$ we get $\nu'(x) = \nu(x)$ and $\nu'(y) = \nu(y)$ and therefore $(\nu'(x), \nu'(y)) \in K \subseteq K'$.

(ii) for some $i \in \{1, \ldots, n\}$, $x \in X_i$, $(x, z) \in \mathsf{flw}'_i$, for some $z$, $(z, y) \in SK$, and $y \notin B'$: $\nu(z) \rightsquigarrow_{M'_i} \max_{\nu'(x)}(M'_i)$ and $(\nu(z), \nu(y)) \in K$. As furthermore $\nu'(y) = \nu(y)$ we conclude $K'(\nu'(x), \nu'(y))$ by the definition of $K'$.

(iii) for some $i \in \{1, \ldots, n\}$, $x \in X_i$, $y \in X_i$, and $(x, y) \in SK'_i$: by induction it follows $(\nu'_i(x), \nu'(y)) \in K'_i$. By definition of $K'$ we get $K'(\nu'(x), \nu'(y))$.

We now show that $B' = B_G$, that is, $\bigcup_i (B'_i \cap X_i) = \{x \in X \mid \nu(x) \neq \nu'(x)\}$. By induction, for every $i$, $B'_i = B_{G_i}$ and thus, $x \in B'_i$ if and only if $\nu_i(x) \neq \nu'_i(x)$. This yields the desired equality, as $B_G = \bigcup_i \{x \in X_i \mid \nu_i(x) \neq \nu'_i(x)\}$.

It remains to show that $\mathsf{flw}_G = \mathsf{flw}' \overset{\text{def}}{=} \{(x, x) \mid SK(x, x), x \notin B'\} \cup \bigcup\limits_i (\mathsf{flw}'_i)_{\upharpoonright X_i}$. By definition, $(x, y) \in \mathsf{flw}_G$ if (a) $x = y$, $\nu'(x) = \nu(x)$ and $\nu(x)$ does not occur in $M(G)$ or (b) $\nu(y) \in \mathit{Free}(M(G))$ and there is a path from $\nu(y)$ to $\nu'(x)$ in $M(G)$. By definition, if $(x, x)$ fulfills (a) then $SK(x, x)$ and $x \notin B'$ hold. If $(x, y)$ fulfills (b), then the path from $\nu(y)$ to $\nu'(x)$ must be in $M(G_i)$, for some $i$, such that $x, y \in X_i$, and thus, by induction, $(x, y) \in \mathsf{flw}'_i$, hence in $\mathsf{flw}'$.

Pairs in $\mathsf{flw}'$ are either in $(\mathsf{flw}'_i)_{\upharpoonright X_i}$, for some $i$, or of the form $(x, x)$ with $x \notin B'$ and $SK(x, x)$. In the former case, by induction, $(x, y) \in \mathsf{flw}_{G_i}$ and therefore, by definition of $\mathsf{flw}_G$: $(x, y) \in \mathsf{flw}_G$. The latter case has two subcases: if $x \in X_0$ then $\nu(x) = \nu(y)$ and $\nu(x)$ does not occur in $M(G)$ (that is: has no event) and thus $(x, x) \in \mathsf{flw}_G$ by (a). Otherwise, $x \in X_i$, for some $i$, and thus $(x, y) \in (\mathsf{flw}'_i)_{\upharpoonright X_i}$. As in the former case, $(x, y) \in \mathsf{flw}_G$ follows. □

Now we can show the following.

**Lemma 25.** *Executability of bHMSCs can be tested in exponential time.*

*Proof.* Our algorithm is an extension of the algorithm of Lemma 10, but taking into account executability. The algorithm has to find out, given a bHMSC $\mathcal{H}$ whether it has an accepting run that is not executable.

To this end, the algorithm first computes the set $RE$ of all tuples

$$(\ell, SK, \ell', SK', \mathsf{flw}', B')$$

for which there is a symbolic run $S$ from location $\ell$ to location $\ell'$ with $SK \xrightarrow{S} (SK', \mathsf{flw}', B')$. This can be done inductively by a fixed point computation using the rules from Definition 23. It should be noted that

- only the base case of this definition involves processes which can be assigned appropriately, as the partial MSC $M$ is explicitly given;
- every awareness relation $SK$ implicity induces a set $D$ of defined registers as $\{x \mid SK(x, x)\}$, thus this part of the algorithm is a strict extension of the algorithm of Lemma 10.

Afterwards, it computes, again by a fixed point computation, the set $RN$ of all tuples $(\ell, SK, \ell', D')$, for which there is an accepting but non-executable symbolic run $S$ from location $\ell$ and awareness relation $SK$ to location $\ell'$ and set $D'$ of defined registers. The base case is as before but it additionally verifies that $M$ is *not* executable at $K_{\restriction Free(M)}$. For sequential composition the algorithm either combines a tuple $(\ell_1, SK_1, \ell_2, SK_2, \mathsf{flw}_2, B_2)$ from $RE$ with a tuple $(\ell_2, SK_2, \ell_3, D_3)$ from $RN$ to yield $(\ell_1, SK_1, \ell_3, D_3)$ or a tuple $(\ell_1, SK_1, \ell_2, D_2)$ from $RN$ with a tuple $(\ell_2, D_2, \ell_3, D_3)$ to yieled $(\ell_1, SK_1, \ell_3, D_3)$. For the fork and join step it combines any number of tuples $(\ell_i, SK_i, \ell'_i, SK'_i, \mathsf{flw}'_i, B'_i)$ from $RE$ with a set of (at least one) tuples $(\ell_j, SK_j, \ell'_j, D'_j)$ from $RN$ to yield a tuple $(\ell, SK, \ell', D')$ just as in the algorithm of Lemma 10.

Finally, the algorithm accepts if there is a tuple $(\ell_0, \{(x_0, x_0)\}, \ell_f, D')$ in $RN$ with an inital location $\ell_0$, a final location $\ell_f$ and an arbitrary set $D'$.

As there are only an exponetial number of tuples in the sets $RN$ and $RE$, and each iteration adds at least one tuple in exponential time, the overall time is exponential. □

Finally we show the corresponding lower bound.

**Lemma 26.** *Executability of bHMSCs is hard for EXPTIME.*

*Proof.* We argue that executability checking is as hard as nonemptiness checking. If we have an algorithm for executability checking, we use it as a black box to obtain an algorithm for nonemptiness checking as follows:

On input $\mathcal{H}$, if $\mathcal{H}$ is not executable, conclude $\mathcal{H}$ is nonempty. Otherwise obtain $\mathcal{H}'$ by adding a non-executable initial transition (similar to $M_1$ of Figure 5) using two new registers. If $\mathcal{H}'$ is not executable, conclude $\mathcal{H}$ is nonempty, otherwise conclude $\mathcal{H}$ is empty.

This argument shows that the lowerbound for nonemptiness checking is also a lowerbound for executability checking. Thus executability checking is Exptime hard even for bHMSCs that are join-free and guarded. □

Theorem 19 follows from Lemmas 25 and 26.

## 7 Implementing Guarded Join-Free bHMSCs

We will now identify a subclass of bHMSCs for which executability and implementabiliy coincide. The following definition is an adaptation of locality from [9]. *Guarded* bHMSCs are based on the notion of a leader process, which determines the next transition to be taken in a bHMSC.

*Definition 27 (guarded).* A join-free bHMSC $\mathcal{H} = (L, X, L_{\text{init}}, L_{\text{acc}}, x_0, T)$ is called *guarded* if there is a mapping $leader : L_{\text{seq}} \to X$ such that

1. for all partial MSCs $M = (E, \lhd, \lambda, \mu)$ that occur in $\mathcal{H}$, $(E, \lhd^*)$ has a unique minimal element $e$ and a unique maximal element $f$; we let $first(M) \overset{\text{def}}{=} pid(e)$ and $last(M) \overset{\text{def}}{=} pid(f)$,

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 $\mathit{leader}(\ell') = \mathit{last}(M)$, and
3. for all transition patterns $$\begin{tikzcd}[row sep=small] & & \ell_1 \\ \ell_0 \arrow[r, "M"] & \ell \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots \\ & & \ell_n \end{tikzcd}$$ it holds (i) for all $i \in \{1, \ldots, n\}$, both $\ell_i \in L_{\mathrm{seq}}$ and $\mathit{leader}(\ell_i) \in \mathit{Pids}(M) \cap X_i$, and (ii) there is $i \in \{1, \ldots, n\}$ such that $\mathit{leader}(\ell_i) = \mathit{last}(M)$.

*Example 28.* The bHMSCs from Examples 4 and 5 are both guarded.

**Theorem 29.** *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.*

Let $\mathcal{H} = (L, X, L_{\mathrm{init}}, L_{\mathrm{acc}}, x_0, T)$ be a guarded join-free bHMSC that is executable (i.e., implementable). The idea is to first enrich locations of $\mathcal{H}$ with awareness relations (in the same spirit as in the proof of Theorem 19). Then, we rely on techniques employed in the context of a bounded number of processes [10, 9], to build a DCA (together with a refinement) that recognizes $L(\mathcal{H})$.

For ease of the construction, we will assume that there are "pseudo transitions" from a fresh initial location $\ell_0$ to all locations from $L_{\mathrm{init}}$, which are labeled with the MSC that consists of one single start event on $x_0$.

Suppose $X = \{z_1, \ldots, z_N\}$ for $N \geqslant 1$ and pairwise distinct $z_i$. Let $\mathscr{M}$ denote the set of all partial MSCs appearing in $\mathcal{H}$. Each $M \in \mathscr{M}$ can be thought of as a unique label of a transition. We assume that the sets of events of the pMSCSs in $\mathcal{H}$ are pairwise disjoint. Let $\mathscr{E}$ be the union of the event sets of the partial MSC in $\mathscr{M}$. Thus, an event $e \in \mathscr{E}$ uniquely identifies the partial MSC $M_e$ to which it belongs and the sequential transition $\delta_e = \ell_e \xrightarrow{M_e} \ell'_e$ that $M_e$ labels.

The naive idea will be to keep the set $\mathscr{E}$ as the DCAstates and allowing each process to evolve to the next event while performing the intended action. However, the non-determinism in the bHMSC $\mathcal{H}$ can lead to potentially inconsistent runs if each process is allowed to choose the next transition. The guardedness property helps us to avoid this problem. We let the leader process non-deterministically choose the next transition. This decision can be communicated to all other active processes, thanks to the connectedness of the partial MSCs. Thus the DCA states will remember the current event as well as the next transition. We need as many local registers as the set of registers of bHMSC $\mathcal{H}$.

More formally, we define the DCA $\mathcal{D} = (S, X, S_{\mathrm{init}}, S_{\mathrm{acc}}, \Delta)$ along with a refinement $(B, h)$ of the message alphabet $A$ such that $h(L(\mathcal{D})) = L(\mathcal{H})$. Note that the set of registers of $\mathcal{D}$ is simply the set $X$ of registers in the bHMSC $\mathcal{H}$.

**States.** A state from $S$ is a pair $(K, e)$ where $K \subseteq X \times X$ is a (symbolic) awareness relation and $e \in \mathscr{E}$ such that $M_e$ is executable at $K$. Intuitively, $e$ is the event that has recently been executed and $K$ represents the global knowledge before executing $M_e$. Hereby, we consider $(\varnothing, e)$ to be a state as well if $e$ is one of the (start) events of the initial MSCs. In that case, we also define $\varnothing \xrightarrow{M_e} \{(x_0, x_0)\}$.

**Refinement.** The ranked alphabet $B$ is the set of pairs $b = (a, s)$ such that $a \in A$ and $s \in S$ is a state of $\mathcal{D}$. We let $\mathit{arity}(b) = \mathit{arity}(a) + N$ and define $h(b) = a$.

**Transitions.** The transition relation $\Delta$ contains a transition

$$((K, e), \alpha, (K', e'))$$

if the following hold:

(1) If $e' \lhd_{\mathsf{spawn}} f'$ for some event $f'$ (of $M_{e'}$), then

$$\alpha = \big(\mathit{pid}(f') := \mathsf{spawn}(s', \mathit{pid}(e'))\big)$$

where $s' = (K', f')$.

(2) If $e' \lhd_{\mathsf{msg}} f'$ and $\mu(e', f') = a(x_1, \ldots, x_n)$ for some event $f'$, then

$$\alpha = !_{\mathit{pid}(f')}((a, s')(x_1, \ldots, x_n, y_1, \ldots, y_N))$$

where $s' = (K', e')$ and, for $i \in \{1, \ldots, N\}$,

$$y_i = \begin{cases} \mathsf{self} & \text{if } y_i = \mathit{pid}(e') \\ z_i & \text{otherwise} \end{cases}$$

(3) If $f' \lhd_{\mathsf{msg}} e'$ and $\mu(f', e') = a(x_1, \ldots, x_n)$ for some event $f'$, then

$$\alpha = ?_*((a, s')(\bot, \ldots, \bot, y_1, \ldots, y_N))$$

where $s' = (K', f')$ and, for $i \in \{1, \ldots, N\}$,

$$y_i = \begin{cases} z_i & \text{if } z_i \in \mathit{Knowl}(K', f') \\ \bot & \text{otherwise} \end{cases}$$

where $\mathit{Knowl}(K', f')$ is the knowledge of $\mathit{pid}(f')$ at $f'$, i.e., $\mathit{Knowl}(K', f') \overset{\text{def}}{=} K'(\mathit{pid}(f')) \cup \{z \mid z \rightsquigarrow_{M_{f'}} f'$ or there is a $z'$ such that $K'(z', z)$ and $z' \rightsquigarrow_{M_{f'}} f'\}$.

(4) Either $e \lhd_{\mathsf{proc}} e'$ or $e$ is maximal on its process.

(5) If $e \lhd_{\mathsf{proc}} e'$, then $K' = K$.

(6) Suppose $e$ is maximal on its process, and $\ell'_e \in L_{\text{seq}}$.

  (i) If $\mathit{pid}(e) = \mathit{leader}(\ell'_e)$, then $K \xrightarrow{M_e} K'$ and there is a sequential transition $\ell'_e \xrightarrow{M} \ell'$ such that $e'$ is the minimal element of $M$ on process $\mathit{leader}(\ell)$.

  (ii) If $\mathit{pid}(e) \neq \mathit{leader}(\ell'_e)$, then there is $M \in \mathscr{M}$ such that $e'$ is both a receive event and the minimal element of $M$ on some process.

(7) Suppose $e$ is maximal on its process and $\ell'_e \in L_{\text{fork}}$ with (unique) outgoing transition

$$\begin{tikzcd} & \ell_1 \\ \ell'_e \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots \\ & \ell_n \end{tikzcd}$$

(i) If there is $i \in \{1, \ldots, n\}$ such that $\mathit{pid}(e) = \mathit{leader}(\ell_i)$, then there is $\hat{K}$ such that $K \xrightarrow{M_e} \hat{K}$ and $K' = \hat{K}_{\restriction X_i}$, and there is a sequential transition $\ell_i \xrightarrow{M} \ell'$ such that $e'$ is the minimal element of $M$ on process $\mathit{leader}(\ell_i)$.

(ii) If there is no $i \in \{1, \ldots, n\}$ such that $\mathit{pid}(e) = \mathit{leader}(\ell_i)$, then there is $M \in \mathscr{M}$ such that $e'$ is a receive event and the minimal element of $M$ on some process.

**Initial States.** A state $(K, e)$ is initial if $K = \emptyset$ and $e$ is one of the (start) events of the initial MSCs.

**Final States.** A state $(K, e)$ is final if the following hold:

- $e \in \max(M_e)$,
- if $\ell'_e \in L_{\text{seq}}$, then $\mathit{pid}(e) \neq \mathit{leader}(\ell'_e)$, and
- if $\ell'_e \in L_{\text{fork}}$ with outgoing transition

$$\begin{tikzcd} & \ell_1 \\ \ell'_e \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots \\ & \ell_n \end{tikzcd}$$

then $\mathit{pid}(e) \neq \mathit{leader}(\ell_i)$ for all $i \in \{1, \ldots, n\}$.

## 7.1 Correctness Proof

We claim $L(\mathcal{H}) = L(\mathcal{D})$. We prove this equality by showing

- $L(\mathcal{H}) \subseteq L(\mathcal{D})$: for every MSC generated by the executable guarded join-free bHMSC $\mathcal{H}$ there is a refinement that is generated by the DCA $\mathcal{D}$ and;
- $L(\mathcal{H}) \supseteq L(\mathcal{D})$: any MSC generated by the DCA $\mathcal{D}$ is also generated by the bHMSC $\mathcal{H}$ up to some refinement.

($\subseteq$) We first assume that $\mathcal{H}$ is executable. Let us consider an accepting run $G = (V, R, \mathit{loc}, \mathit{reg}, \rho)$ of $\mathcal{H}$ generating the MSC $M(G)$. We will consider a refinement $(B, h)$ of $M(G)$ as described in the construction, and show that $h^{-1}(M(G))$ is realizable by the DCA $\mathcal{D}$. By definition $M(G)$ is executable. Hence it is possible to label every node $u$ of $G$ with the corresponding awareness relation $K_u$ preserving executability. The initial awareness relation $K_{\mathit{in}(G)} = \{(x_0, x_0)\}$. The successive awareness relations can be computed as described in Definition 23. We then add a new (pseudo) initial state to the run $G$ with $\mathit{loc} = \ell_0$ and awareness relation $\emptyset$, and add an edge to the old initial state with an edge labelled by an MSC which consists of a single start event on $x_0$. This gives us a new (pseudo) run graph $G_{\mathit{in}}$.

Suppose a node $u$ of $G_{in}$ is labeled by a location $\ell \in L_{\text{seq}}$, an awareness relation $K$ and a register assignment $\nu$ (that is, $loc(u) = \ell \in L_{\text{seq}}$ and $reg(u) = \nu$). Since $G_{in}$ is accepting, $u$ has a successor $u'$ with a register assignment $\nu'$. The partial MSC labeling this edge will be $\nu'(M)$ for some $M$ labeling an outgoing transition of $\ell$ (that is $\rho(u, u') = \nu'(M)$). Since the accepting run $G_{in}$ is executable, $M$ is executable at $K$. Thus $(K, e)$ is a valid state of the DCA $\mathcal{D}$ for all events $e \in M$. Thus we can label the corresponding event in the partial MSC $h^{-1}(\nu'(M))$, which we denote by $e_u$, by the state $(K, e)$. We denote the labeling function by $\sigma$. That is, $\sigma(e_u) = (K, e)$.

We also define another labeling $\tau$ for the current register assignment. First, $\tau(init(h^{-1}(M(G_{in}))))$ is undefined everywhere. The values of $\tau$ at the successive nodes of the MSC $h^{-1}(M(G_{in}))$ can be computed in a way consistent to that in the definition of the run of a DCA. If $e$ is a maximal event of the partial MSC $M_e$ labeling an edge $(u, u')$ of the run $G_{in}$ (with $reg(u') = \nu'$), then $\tau(u_e)$ should agree with the awareness relation $K'$ at $v$. This is to say that, if $pid(e) = x$, then for all registers $y \in X$, $\tau_{\nu'(e)}(y) = \nu'(y)$ if $y \in K'(x)$. Such a consistent labeling with $\tau$ is always possible, thanks to the executability. Thus, given a run $G$ of the bHMSC $\mathcal{H}$, we can obtain a unique labeling $\sigma$ of the events of $h^{-1}(M(G_{in}))$ by the states of the DCA $\mathcal{D}$, and another unique labeling $\tau$ of the events of $h^{-1}(M(G_{in}))$ by the register assignments. It is sufficient to show that the labeling $(\sigma, \tau)$ of $h^{-1}(M(G_{in}))$ corresponds to an accepting run of $\mathcal{D}$.

First we notice that the labeling $\sigma$ allows consistent tiling of $h^{-1}(M(G))$ with the transitions of the DCA $\mathcal{D}$. Moreover, if a prefix of $h^{-1}(M(G_{in}))$ has been executed, the successive event can be executed in that configuration. Thus $(\sigma, \tau)$ is a valid run of $\mathcal{D}$ on $h^{-1}(M(G_{in}))$. Since $G$ was accepting, all the maximal nodes of $G$ are labeled by the state $\bot$ which does not have a leader. Thus all the maximal events of $M(G_{in})$ are labeled by accepting states of $\mathcal{D}$. Thus $(\sigma, \tau)$ is accepting as well.

($\supseteq$) However, the converse direction is more involved. Consider an accepting run $(\sigma, \tau)$ of the DCA $\mathcal{D}$ on an MSC $\mathcal{M}$. We describe an inductive construction of an *executable* accepting run $G$ of the bHMSC $\mathcal{H}$ generating the MSC $h(\mathcal{M})$.

We denote the partial initial run of the bHMSC $\mathcal{H}$ by $G'$. The inductive step appends transitions to $G'$, so that finally, when the procedure terminates, $G'$ will be the accepting run $G$ we are aiming for. A prefix of $h(\mathcal{M})$ will be generated by $G'$ (which is denoted $h(M(G')))$; the remaining part of $h(\mathcal{M})$ is a partial MSC which we denote by $M_{\mathsf{rem}}$. Thus after each iteration, the size of $G'$ increases and that of $M_{\mathsf{rem}}$ decreases.

To start with, $G'$ is $init(h(\mathcal{M}))$ and $M_{\mathsf{rem}}$ is $h(\mathcal{M}) \backslash init(h(\mathcal{M}))$. Consider the minimal event $m$ of $M_{\mathsf{rem}}$. This minimal event uniquely exists, and is the $\lhd_{\mathsf{proc}}$-successor of $init(h(\mathcal{M}))$ . Let $\sigma_m = (K, e)$, an initial state of the DCA $\mathcal{D}$. By definition of $\mathcal{D}$, $K = \{(x_0, x_0)\}$, $\ell_e \in L_{\text{init}}$, and $e = init(h^{-1}(M_e))$. The initial register assignment $\nu$ is defined only at $pid(e)$. We let $\nu(pid(e)) = pid(m)$. We store in $E_{\mathsf{curr}}$ the minimal set of events needed to trace out an occurrence of $h^{-1}(M_e)$ in $M_{\mathsf{rem}}$ with $m \in E_{\mathsf{curr}}$. This is always possible, as the run $(\sigma, \tau)$ is accepting, and the partial MSCs occuring in the bHMSC are guarded. Moreover, for each $m' \in E_{\mathsf{curr}}$, $\sigma(m') = (K, e')$ for some $e' \in M_e$. The awareness relation $K$ remains the same throughout the events in $E_{\mathsf{curr}}$. Consider $\ell_e$ and $\ell'_e$, the source and target locations of $M_e$. Necessarily $\ell_e \in L_{\text{init}}$. We can compute the awareness relation $K'$ as $K \xrightarrow{M_e} K'$. We can uniquely determine the register assignment $\nu'$ in the following way. $\nu'$ is defined only at $\mathit{Pids}(M_e)$. For all maximal events $m'$ in $E_{\mathsf{curr}}$ such that $m'$ corresponds to a maximal event $e' \in M_e$, we let $\nu'(\mathit{pid}(e')) = \mathit{pid}(m')$. This definition of $\nu'$ is consistent with $K'$ and $\tau$. That is, for all maximal events $m'$ in $E_{\mathsf{curr}}$ such that $m'$ corresponds to a maximal event $e' \in M_e$, if $\mathit{pid}(e') = x$ and $y \in K'(x)$, then $\nu'(y) = \tau_{m'}(y)$.

We have two cases depending on the target location of $M_e$.

– If $\ell'(e) \in L_{\text{seq}} \cup \{\perp\}$, then we update $G'$ to

$$\begin{tikzcd} \overset{\varnothing, \varnothing}{\ell_0} \arrow[r, "\nu(x_0)"] & \overset{\nu, K}{\ell_e} \arrow[r, "\nu'(M_e)"] & \overset{\nu', K'}{\ell'_e} \end{tikzcd}$$

and $M_{\mathsf{rem}}$ will be updated to $M_{\mathsf{rem}} \backslash E_{\mathsf{curr}}$.

– Otherwise, let $\begin{tikzcd}[row sep=small] & \ell_1 \\ \ell'_e \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots \\ & \ell_n \end{tikzcd}$. Then we update $G'$ to

$$\begin{tikzcd} & & & & \overset{\nu'_{\restriction X_1}, K'_{\restriction X_1}}{\ell_1} \\ \overset{\varnothing, \varnothing}{\ell_0} \arrow[r, "\nu(x_0)"] & \overset{\nu, K}{\ell_e} \arrow[r, "\nu'(M_e)"] & \overset{\nu', K'}{\ell'_e} \arrow[urr, "X_1"] \arrow[drr, "X_n"'] & & \vdots \\ & & & & \underset{\nu'_{\restriction X_n}, K'_{\restriction X_n}}{\ell_n} \end{tikzcd}$$

We update $M_{\mathsf{rem}}$ to $M_{\mathsf{rem}} \backslash E_{\mathsf{curr}}$.

Notice that $G'$ is executable at $\varnothing$ in both cases. This is because by definition $(K, e)$ is a state only if $M_e$ is executable at $K$. Notice that $M(G') \circ M_{\mathsf{rem}} = M$.

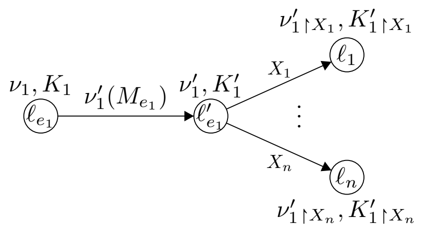
Now let us consider the inductive case. Consider $m$, a minimal event of the partial MSC $M_{\mathsf{rem}}$. Suppose $\sigma_m = (K, e)$. We trace out an occurrence of $h^{-1}(M_e)$ involving $m$ (this is always possible, thanks to guardedness). Let these events be denoted $E_{\mathsf{curr}}$. As before, the same awareness relation $K$ appears in the labels of all events in $E_{\mathsf{curr}}$. Since $m$ is a minimal event of $M_{\mathsf{rem}}$, $\mathit{pid}(e)$ is the $\mathit{leader}(M_e) = \mathit{leader}(\ell_e)$. As $m$ is not $\mathit{init}(\mathcal{M})$, $m$ has a $\lhd_{\mathsf{proc}}$-predecessor $m_1$ in $\mathcal{M}$ (that is, $m_1 \lhd_{\mathsf{proc}} m$). Suppose $\sigma_{m_1} = (K_1, e_1)$. Consider $\ell_e$ and $\ell'_e$ as well as $\ell_{e_1}$ and $\ell'_{e_1}$. By virtue of the construction of the DCA $\mathcal{D}$, either

– $\ell'_{e_1} \in L_{\text{seq}}$. We have

$$\begin{tikzcd} \overset{\nu_1, K_1}{\ell_{e_1}} \arrow[r, "\nu'_1(M_{e_1})"] & \overset{\nu'_1, K'_1}{\ell'_{e_1}} \end{tikzcd}$$

In this case, thanks to Condition (6) in the definition of the transitions of the DCA $\mathcal{D}$, $\ell'_{e_1} = \ell_e$ and $K_1 \xrightarrow{M_{e_1}} K'_1 = K$. We let $\nu = \nu'_1$.

– or $\ell'_{e'} \in L_{\text{fork}}$, say

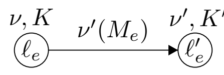


In this case, thanks to Condition (7)(i) in the definition of the transitions of the DCA $\mathcal{D}$, $\ell_e = \ell_i$ for some $i$ and $pid(e) \in X_i$. Moreover, $K = K'_{1\restriction X_i}$. We let $\nu = \nu'_{1\restriction X_i}$.

Thus we have the register assignment $\nu$ and an awareness relation $K$ at $\ell_e$. Indeed the awareness relation $K$ at $\ell_e$ is the same as that labeling $m$. As in the base case, we compute the awareness relation $K'$ as $K \xrightarrow{M_e} K'$. We can uniquely determine the register assignment $\nu'$ in the following way. $\nu'$ is defined only at $Pids(M_e) \cup \mathsf{Domain}(\nu)$. For all maximal events $m'$ in $E_{\mathsf{curr}}$ such that $m'$ corresponds to a maximal event $e' \in M_e$, we let $\nu'(pid(e')) = pid(m')$. For the remaining registers $y$, we let $\nu'(y) = \nu(y)$. Again, this definition of $\nu'$ is consistent with $K'$ and $\tau$. That is, for all maximal events $m'$ in $E_{\mathsf{curr}}$ such that $m'$ corresponds to a maximal event $e' \in M_e$, if $pid(e') = x$ and $y \in K'(x)$, then $\nu'(y) = \tau_{m'}(y)$.

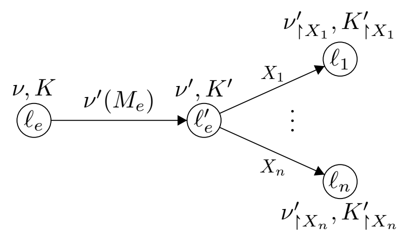
We update the partial initial run $G'$ and the partial MSC $M_{\mathsf{rem}}$ as in the base case. We have two cases depending on the target location of $M_e$.

– If $\ell'(e) \in L_{\text{seq}} \cup \{\bot\}$, then let $G''$ be the following.



We append $G''$ to $G'$ and $M_{\mathsf{rem}}$ will be updated to $M_{\mathsf{rem}} \backslash E_{\mathsf{curr}}$.

– Otherwise, let $$\begin{tikzcd} & \ell_1 \\ \ell'_e \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots \\ & \ell_n \end{tikzcd}$$ . Then let $G''$ be the following


$\nu'_{\restriction X_1}, K'_{\restriction X_1}$
$\nu, K$
$\nu'(M_e)$
$\nu', K'$
$X_1$
$\ell_1$
$\ell_e$
$\ell'_e$
$\vdots$
$X_n$
$\ell_n$
$\nu'_{\restriction X_n}, K'_{\restriction X_n}$

We append $G''$ to $G'$ and update $M_{\mathsf{rem}}$ to $M_{\mathsf{rem}} \backslash E_{\mathsf{curr}}$.

Notice that $G''$ is executable at $K$ in both cases. This is because by definition $(K, e)$ is a state only if $M_e$ is executable at $K$, and we have $(K, e)$ as a state of the DCA $\mathcal{D}$. By induction, $G'$ is executable at the initial awareness relation $\varnothing$. Notice that $M(G') \circ M_{\mathsf{rem}} = M$.

The procedure terminates as each step removes at least one node from the partial MSC $M_{\mathsf{rem}}$. Finally $G = G'$ if $M_{\mathsf{rem}}$ is empty. We now argue why $G$ must be accepting. Suppose not. This means that there is a node $v$ in $G$ which does not have a successor and the state labeling $v$ is not $\bot$. Let $\nu''$ be the register assignment and $\ell'' \neq \bot$ be the state label at the node $v$. Consider the neighborhood of $v$ in $G$. It is in one of the following forms:

– $\begin{tikzcd} \overset{\nu, K}{\ell} \arrow[r, "\nu'(M)"] & \overset{\nu', K'}{\ell'} \end{tikzcd} = v$ . In this case, $\ell' = \ell''$ and $\nu' = \nu''$. By definition, $\mathit{leader}(\ell'') \in \mathit{Pids}(M)$.

– or

$$\begin{tikzcd} & & \overset{\nu'_{\restriction X_1}, K'_{\restriction X_1}}{\ell_1} \\ \overset{\nu, K}{\ell} \arrow[r, "\nu'(M)"] & \overset{\nu', K'}{\ell'} \arrow[ur, "X_1"] \arrow[dr, "X_n"'] & \vdots \quad \ell_i = v \\ & & \underset{\nu'_{\restriction X_n}, K'_{\restriction X_n}}{\ell_n} \end{tikzcd}$$

In this case, $\ell_i = \ell''$ and $\nu'_{\restriction X_i} = \nu''$. By definition, $\mathit{leader}(\ell'') \in \mathit{Pids}(M) \cap X_i$.

Let $\mathit{leader}(\ell'') = x$. Let $e$ be the maximal $x$-event of $M$. By definition of the accepting states, $(K, e)$ is not accepting for any $K$. Thus the process $\nu'(x)$ is not in an accepting state. This contradicts the premise that the run $(\sigma, \tau)$ is accepting. Thus $G$ must be accepting. □

However, guardedness does not yield better complexities, since the lower-bound proofs work for the restricted class:

**Theorem 30.** *Nonemptiness and executability for guarded join-free bHMSCs are EXPTIME-complete.*

## 8 Conclusion

We proposed an extension of DCA as well as a new model for the specification of dynamic communicating systems, and we studied their implementability problem. We believe that with our techniques classes of bHMSC can be found, for which executability and implementability coincide, and that need not be join-free or guarded. To this aim, it may be worthwhile to consider restrictions of branching automata such as fork-acyclicity, and to transfer them to bHMSCs.

## References

1. R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of msc graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.
2. Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. *Logical Methods in Computer Science*, 7(4), 2011.
3. B. Bollig and L. Hélouët. Realizability of dynamic MSC languages. In *CSR'10*, volume 6072 of *LNCS*, pages 48–59, 2010.
4. J. Borgstroem, A. Gordon, and A. Phillips. A chart semantics for the pi-calculus. *Electronic Notes in Theoretical Computer Science*, 194(2):3–29, January 2008.
5. L. Bozzelli, S. La Torre, and A. Peron. Verification of well-formed communicating recursive state machines. *Theor. Comput. Sci.*, 403(2-3):382–405, 2008.
6. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2), 1983.
7. 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.
8. 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.
9. 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.
10. L. Hélouët and C. Jard. Conditions for synthesis of communicating automata from HMSCs. In *Proceedings of FMICS'00*, pages 203–224. Springer, 2000.
11. J. G. Henriksen, M. Mukund, K. Narayan Kumar, M. A. Sohoni, and P. S. Thiagarajan. A theory of regular MSC languages. *Inf. Comput.*, 202(1):1–38, 2005.
12. ITU-TS. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC).* ITU-TS, Geneva, February 2011.
13. 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.
14. K. Lodaya and P. Weil. Series-parallel languages and the bounded-width property. *Theoretical Computer Science*, 237(1-2):347 – 380, 2000.
15. K. Lodaya and P. Weil. Rationality in algebras with a series operation. *Information and Computation*, 171(2):269 – 293, 2001.
16. M. Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theoretical Computer Science*, 309(1-3):529–554, 2003.
17. R. Meyer. On boundedness in depth in the $\pi$-calculus. In *Proceedings of IFIP TCS'08*, volume 273 of *IFIP*, pages 477–489. Springer-Verlag, 2008.
18. M. L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.
19. R. Morin. Recognizable sets of message sequence charts. In *Proceedings of STACS 2002*, volume 2285 of *Lecture Notes in Computer Science*, pages 523–534. Springer, 2002.
20. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL 2006*, volume 4207 of *LNCS*, pages 41–57. Springer, 2006.
21. Helmut Seidl. Haskell overloading is dexptime-complete. *Inf. Process. Lett.*, 52(2):57–60, 1994.