Asynchronous cellular automata and logic for pomsets without auto-concurrency*

Manfred Droste
Institut für Algebra
Technische Universität Dresden
D-01062 Dresden
droste@math.tu-dresden.de

Paul Gastin
LIAFA
Université Paris 7
2, place Jussieu
F-75251 Paris Cedex 05
Paul.Gastin@liafa.ibp.fr

September 5, 1997

Abstract

This paper extends to pomsets without auto-concurrency the fundamental notion of asynchronous cellular automata (ACA) which was originally introduced for traces by Zielonka. We generalize to pomsets the notion of asynchronous mapping introduced by Zielonka and we show how to construct a deterministic ACA from an asynchronous mapping. Our main result generalizes Büchi's theorem for finite words to a class of pomsets without auto-concurrency which satisfy a natural axiom. This axiom ensures that an asynchronous cellular automaton works on the pomset as a concurrent read and exclusive owner write machine. More precisely, we prove the equivalence between non deterministic ACA, deterministic ACA and monadic second order logic for this class of pomsets.

1 Introduction

In a distributed system, some events may occur concurrently, meaning that they may occur in any order or simultaneously or even that their executions may overlap. This is the case in particular when two events use independent

*This research was partly carried out during a stay of the second author in Dresden.

resources. On the other hand, some events may causally depend on each other. For instance, the receiving of a message must follow its sending. Therefore, a distributed behavior may be abstracted as a pomset, that is a set of events together with a partial order which describes causal dependencies of events and with a labeling function. In this paper, we mainly deal with pomsets without auto-concurrency: concurrent events must have different labels. These pomsets are called semi-words in [14, 3]. For studies how general pomsets can be used to represent parallel processes and how they can be composed, we refer the reader e.g. to [13, 8].

There are several ways to describe the behaviors of a system. For instance, logic formulas are suited for specification purposes. Depending on the properties we have to express, we can use various logics such as temporal logics, first order logics or (monadic) second order logics. On the other hand, transition systems are often used to give more operational descriptions. In this paper, we will concentrate on these two kinds of descriptions of systems.

When dealing with distributed systems, it is natural to look for transition systems which faithfully reflect the concurrency. For instance, Petri nets are a widely studied class of such transition systems. Asynchronous cellular automata (ACA) form another fundamental class of transition systems with built-in concurrency. They were introduced for traces by Zielonka [16, 17]. Mazurkiewicz introduced traces in order to describe the behaviors of one-safe Petri nets [10, 11]. A trace is a pomset where the partial order is dictated by a static dependence relation over the actions of the system.

The primary aim of this work is to generalize the notion of ACA so that they can work on pomsets without auto-concurrency. There are several possible definitions for such ACA. In Section 3, we give a natural definition of ACA which intuitively run over the Hasse diagrams of pomsets. We investigate the closure of this class of automata under boolean operations. We also discuss possible alternative definitions.

Asynchronous mappings have proven to be a basic tool to construct ACA for traces [2]. In Section 4, we give a definition of asynchronous mappings for general pomsets. We show that a pomset language recognized by an asynchronous mapping can be accepted by a *deterministic* ACA.

The rest of this paper is devoted to the equivalence between ACA and monadic second order (MSO) logic for pomsets. We prove in Section 5 that from a (non deterministic) ACA one can construct a MSO formula which defines precisely the pomset language accepted by the automaton. In Section 6, we prove the converse for the special subclass of pomsets for which the ACA works as a concurrent read and exclusive owner write (CROW) machine. These pomsets are called CROW-pomsets. More precisely, from a MSO formula we construct a *deterministic* ACA which accepts precisely the

CROW-pomsets defined by the formula. Therefore, for CROW-pomsets, we have the equivalence between MSO logic, deterministic ACA and non deterministic ACA. This result is crucial since it opens the way of model checking for distributed systems whose behaviors are described as CROW-pomsets.

Note that the CROW assumption for pomsets is satisfied for a large class of systems. For instance, a stably concurrent automaton $\mathcal{A}$ [7] defines dynamic dependencies between actions of a system, whence it generalizes the static dependencies on actions used for traces. With the computation sequences of a stably concurrent automaton $\mathcal{A}$, one can associate dependence orders [1]. These dependence orders are pomsets which generalize traces but are special cases of pomsets without auto-concurrency. Under the assumption that the automaton $\mathcal{A}$ is stably concurrent and forwardly weakly preserves dependency, these dependence orders satisfy the CROW axiom. Therefore, as a consequence of our results, we obtain a new, Zielonka-type characterization of the recognizable languages of concurrency monoids studied in [7].

An extended abstract of this work already appeared in [6].

2 Preliminaries

2.1 Pomsets

Let Σ be a finite set, called alphabet. A pomset over Σ is (an isomorphism class of) a finite labeled partial order $t = (V, \leq, \lambda)$ where V is a finite set of vertices, $\leq$ is the partial order on V and $\lambda : V \longrightarrow \Sigma$ is the labeling function. The empty pomset $(\emptyset, \emptyset, \emptyset)$ will be denoted by 1.

Let $s = (V_s, \leq_s, \lambda_s)$ and $t = (V_t, \leq_t, \lambda_t)$ be two pomsets. We say that s is a prefix of t, denoted by $s \preceq t$ if s is (isomorphic to) a downward closed subpomset of t, that is, if V_s is a downward closed subset of V_t (i.e. $V_s \subseteq V_t$ and for all $u, v \in V_t$, $u \leq_t v$ and $v \in V_s$ imply $u \in V_s$) and $\leq_s$ is the restriction of $\leq_t$ to V_s (i.e. $\leq_s = \leq_t \cap V_s \times V_s$) and λ_s is the restriction of λ_t to V_s. The prefix order relation is a partial order on the set of pomsets. In the following, we will identify a downward closed subset of vertices with the corresponding prefix of the pomset. Let $s_1 = (V_{s_1}, \leq_{s_1}, \lambda_{s_1})$ and $s_2 = (V_{s_2}, \leq_{s_2}, \lambda_{s_2})$ be two prefixes of a pomset $t = (V_t, \leq_t, \lambda_t)$. Then, $V_{s_1} \cup V_{s_2}$ is a downward closed subset of V_t and the corresponding prefix of t is $s_1 \cup s_2 = (V_{s_1} \cup V_{s_2}, \leq_{s_1} \cup \leq_{s_2}, \lambda_{s_1} \cup \lambda_{s_2})$ where $\lambda_{s_1} \cup \lambda_{s_2}$ is the labeling which coincides with λ_{s_1} on V_{s_1} and with λ_{s_2} on V_{s_2} (note that λ_{s_1} and λ_{s_2} aggree on $V_{s_1} \cap V_{s_2}$).

Let $t = (V, \leq, \lambda)$ be a pomset. The downward closure of a vertex v is denoted by $\downarrow v = \{u \in V \mid u \leq v\}$. The strict downward closure of a vertex v is denoted by $\Downarrow v = \downarrow v \setminus \{v\}$. Since $\downarrow v$ and $\Downarrow v$ are downward closed subsets of V, we will identify these sets with the corresponding prefixes of t.

Let $\Sigma_1, \ldots, \Sigma_n$ be pairwise disjoint alphabets and let $\Sigma = \Sigma_1 \dot\cup \cdots \dot\cup \Sigma_n$. Intuitively, we can view $[n] = \{1, \ldots, n\}$ as a set of labels of sequential processes and $\Sigma_1, \ldots, \Sigma_n$ as the sets of actions of these sequential processes. Let $p : \Sigma \longrightarrow [n]$ be the mapping which associates with each letter $a \in \Sigma$ the process $p(a) \in [n]$ which executes the letter a, i.e. $a \in \Sigma_{p(a)}$.

Let $t = (V, \leq, \lambda)$ be a pomset. We say that a vertex v covers a vertex u, denoted by $u \prec v$, if $u < v$ and there is no vertex w such that $u < w < v$. We say that two vertices $u, v \in V$ are concurrent, denoted by $u \parallel v$, if $u \not\leq v$ and $u \not\geq v$. We may see the covering relation as the description of the interactions between the processes. More precisely, we consider that an event $v \in V$ reads the states of the processes $p \circ \lambda(\{u \mid u \prec v\})$ and writes in the process $p \circ \lambda(v)$. We will not allow concurrent writes, therefore two concurrent events $u \parallel v$ must write in different process $p \circ \lambda(u) \neq p \circ \lambda(v)$. This leads to the following definition.

A $(\Sigma_1, \ldots, \Sigma_n)$-pomset is a pomset $t = (V, \leq, \lambda)$ for which $\lambda^{-1}(\Sigma_i)$ is totally ordered for all $1 \leq i \leq n$. The set of $(\Sigma_1, \ldots, \Sigma_n)$-pomsets will be denoted by $\mathbb{P}(\Sigma_1, \ldots, \Sigma_n)$. Note that with this notation the set $\mathbb{P}(\Sigma)$ is the set of words over Σ. Another special case which will come into play later is when the sets $\Sigma_1, \ldots, \Sigma_n$ are all singletons.

For $A \subseteq \Sigma$, we denote by $\partial_A(t) = \downarrow\lambda^{-1}(A)$ the least prefix of a pomset t which contains all letters from A. Note that $\partial_A(t) = \bigcup_{v \mid \lambda(v) \in A} \downarrow v$. For $a \in \Sigma$ and $i \in [n]$, we will use the following simplified notations: $\partial_a(t) = \partial_{\{a\}}(t)$ and $\partial_i(t) = \partial_{\Sigma_i}(t)$.

Note that if $\lambda^{-1}(A)$ is totally ordered then $\partial_A(t)$ is either empty or has exactly one maximal vertex. In particular, if t is a $(\Sigma_1, \ldots, \Sigma_n)$-pomset then $\partial_i(t)$ is either empty or has exactly one maximal vertex.

2.2 Traces

We recall now basic definitions for Mazurkiewicz traces which will be needed in this paper. The reader is referred to [5] for a general presentation of traces.

A dependence alphabet is a pair (Σ, D) where Σ is a finite alphabet and $D \subseteq \Sigma \times \Sigma$ is a reflexive and symmetric relation over Σ called the dependence relation. Intuitively, two dependent actions $(a, b) \in D$ must be executed sequentially while two independent actions $(a, b) \notin D$ may occur concurrently. More formally, one considers the congruence relation $\sim$ over the free monoid Σ^* generated by the relation $\{(ab, ba) \mid (a, b) \notin D\}$. A trace is simply an equivalence class of words for the congruence $\sim$. The trace monoid is then the quotient $\mathbb{M}(\Sigma, D) = \Sigma^* / \sim$.

We give now an equivalent definition of traces which is more adequate in our context. Basically, a trace can be seen as a pomset which satisfies additional requirements. More precisely, we will see that a trace over the dependence alphabet (Σ, D) is a pomset $t = (V, \leq, \lambda)$ such that for all vertices $u, v \in V$,

$$(\lambda(u), \lambda(v)) \in D \implies u \leq v \vee v \leq u \tag{1}$$

$$u \prec\!\!\!\cdot\, v \implies (\lambda(u), \lambda(v)) \in D \tag{2}$$

Indeed, let $t = (V, \leq, \lambda)$ be a pomset satisfying conditions (1) and (2). Note that a linearization of t may be identified with a word of Σ^*. The set of linearizations of t is precisely a trace, that is, an equivalence class for $\sim$. Hence, with a pomset t satisfying (1) and (2), one can associate a trace $\varphi(t)$. Conversely, a word $u \in \Sigma^*$ defines a labeled linear order $(V_u, \leq_u, \lambda_u)$ over the occurrences of actions of u: $V_u = \{(a, i) \mid 1 \leq i \leq |u|_a\}$ ($|u|_a$ denotes the number of occurrences of a in u), $(a, i) \leq_u (b, j)$ if the i-th a occurs before the j-th b in u and $\lambda_u((a, i)) = a$. Since two equivalent words $u \sim v$ have the same set of occurrences of actions ($V_u = V_v$), we can associate with a trace $[u]$ the pomset $\psi([u]) = (V_u, \bigcap_{v \sim u} \leq_v, \lambda_u)$. One can check that $\psi([u])$ satisfies conditions (1) and (2) and that ψ and φ are inverse bijections. This explains why the two definitions are equivalent.

We will now define recognizable trace languages. A trace automaton is a quadruple $\mathcal{A} = (Q, T, I, F)$ where Q is a finite set of states, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states and $T \subseteq Q \times \Sigma \times Q$ is the set of transitions which verifies the diamond property: for all $(a, b) \in \Sigma \times \Sigma \setminus D$ and $q, q', q'' \in Q$, if $(q, a, q') \in T$ and $(q', b, q'') \in T$ then there exists some $\bar{q}' \in Q$ such that $(q, b, \bar{q}') \in T$ and $(\bar{q}', a, q'') \in T$. A word $w = a_1 \cdots a_n \in \Sigma^*$ is accepted by $\mathcal{A}$ if there is a run $q_0, a_1, q_1, \ldots, a_n, q_n$ such that $q_0 \in I$, $q_n \in F$ and $(q_{i-1}, a_i, q_i) \in T$ for all $1 \leq i \leq n$. A trace $t \in \mathbb{M}(\Sigma, D)$ is accepted by $\mathcal{A}$ if some linear extension of t is accepted by $\mathcal{A}$. Note that, thanks to the diamond property of a trace automaton, if some linear extension of a trace is accepted by $\mathcal{A}$ then all linear extensions of t are accepted by $\mathcal{A}$. A trace language $L \subseteq \mathbb{M}(\Sigma, D)$ is recognizable if it is the set of traces accepted by some trace automaton.

3 Asynchronous Cellular Automata

In this section, we introduce asynchronous cellular automata (ACA) over $(\Sigma_1, \ldots, \Sigma_n)$-pomsets and we study the closure properties of ACA under boolean operations. We give several examples of ACA and we discuss possible variations of the definition.

Definition 3.1 *A $(\Sigma_1, \ldots, \Sigma_n)$-ACA (asynchronous cellular automaton) is a tuple*

$$\mathcal{A} = ((Q_i)_{i \in [n]}, (\delta_{a,J})_{a \in \Sigma, J \subseteq [n]}, F)$$

where

1. *for all $i \in [n]$, Q_i is a finite set of local states for process i,*
2. *for all $a \in \Sigma$ and $J \subseteq [n]$, $\delta_{a,J} : \prod_{i \in J} Q_i \longrightarrow \mathcal{P}(Q_{p(a)})$ is a transition function,*
3. *$F \subseteq \bigcup_{J \subseteq [n]} \prod_{i \in J} Q_i$ is the set of accepting states.*

The automaton is deterministic if all transition functions are deterministic, i.e. if $|\delta_{a,J}((q_i)_{i \in J})| \leq 1$ for all $a \in \Sigma$, $J \subseteq [n]$ and $(q_i)_{i \in J} \in \prod_{i \in J} Q_i$.

In order to explain how a $(\Sigma_1, \ldots, \Sigma_n)$-ACA accepts a $(\Sigma_1, \ldots, \Sigma_n)$-pomset, we need to introduce first some new notations. Let $t = (V, \leq, \lambda)$ be a $(\Sigma_1, \ldots, \Sigma_n)$-pomset and let $v \in V$ be a vertex in t. We define the write domain of v by $W(v) = p \circ \lambda(v)$ and the read domain of v as the set of processes of vertices covered by v: $R(v) = p \circ \lambda(\{u \in V \mid u \prec v\})$. We also denote by $\max(t)$ the set of maximal vertices of t and by $F(t) = p \circ \lambda(\max(t))$ the set of processes corresponding to these maximal vertices.

Intuitively, in order to perform an event v, the ACA reads the label $\lambda(v)$ and the present states of the processes in $R(v)$ and according to its transition function $\delta_{\lambda(v),R(v)}$ determines the new state of the process $W(v)$. At the end of its run, the ACA collects the states of the maximal processes in $F(t)$ in order to decide whether the run is accepted or rejected. The formal definition is given below.

Note that if $i \in R(v)$ then $\partial_i(\Downarrow v)$ is non empty, whence has only one maximal vertex which will be identified with $\partial_i(\Downarrow v)$ in Definition 3.2. Similarly, if $i \in F(t)$ then $\partial_i(t)$ is non empty and will also be identified with its maximum vertex.

Definition 3.2 *Let $\mathcal{A} = ((Q_i)_{i \in [n]}, (\delta_{a,J})_{a \in \Sigma, J \subseteq [n]}, F)$ be a $(\Sigma_1, \ldots, \Sigma_n)$-ACA and let $t = (V, \leq, \lambda)$ be a $(\Sigma_1, \ldots, \Sigma_n)$-pomset. A run of $\mathcal{A}$ over t is a mapping $r : V \longrightarrow \bigcup_{i \in [n]} Q_i$ such that for all $v \in V$,*

$$r(v) \in \delta_{\lambda(v),R(v)} \left(r(\partial_i(\Downarrow v))_{i \in R(v)} \right)$$

The run r is accepted if its final maximal state $f(r) = r(\partial_i(t))_{i \in F(t)}$ is in the accepting set F.

Finally, a $(\Sigma_1, \ldots, \Sigma_n)$-pomset t is accepted by $\mathcal{A}$ if there is some accepting run of $\mathcal{A}$ on t. The set of $(\Sigma_1, \ldots, \Sigma_n)$-pomsets accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

Note that, there is a natural bijection between $\{\partial_i(\Downarrow v) \mid i \in R(v)\}$ and $\{u \in V \mid u \longrightarrow\!\!\!< v\}$. Hence, in order to lighten the notation, we may also write

$$r(v) \in \delta_{\lambda(v),R(v)}\left((r(u))_{u \prec v}\right)$$

Similarly, the acceptance condition can also be written $f(r) = (r(u))_{u \in \max(t)} \in F$. The notations adopted in Definition 3.2, also more complex, are more suitable to the discussion of alternative definitions given in Remark 3.3.

As a first example, we will give a deterministic $(\Sigma_1, \dots, \Sigma_n)$-ACA which accepts precisely the set of $(\Sigma_1, \dots, \Sigma_n)$-pomsets satisfying condition (2) of Section 2.2. More precisely, let (Σ, D) be a dependence alphabet with $\Sigma = \Sigma_1 \dot\cup \cdots \dot\cup \Sigma_n$. We define the $(\Sigma_1, \dots, \Sigma_n)$-ACA $\mathcal{A} = ((Q_i)_{i \in [n]}, (\delta_{a,J})_{a \in \Sigma, J \subseteq [n]}, F)$ where $Q_i = \Sigma_i \cup \{\bot\}$ for all $i \in [n]$ and

$$\delta_{a,J}((q_j)_{j \in J}) = \begin{cases} a & \text{if } q_j \neq \bot \text{ and } (a, q_j) \in D \text{ for all } j \in J \\ \bot & \text{otherwise} \end{cases}$$

for all $a \in \Sigma$ and $J \subseteq [n]$. Finally, the set of accepting states is $F = \bigcup_{J \subseteq [n]} \prod_{i \in J} \Sigma_i$. One can easily check that $L(\mathcal{A})$ is the set of $(\Sigma_1, \dots, \Sigma_n)$-pomsets $(V, \leq, \lambda)$ such that for all $u, v \in V$, if $u \prec v$ then $(\lambda(u), \lambda(v)) \in D$. For instance, if $(\Sigma, D) = a \text{ — } b \text{ — } c \text{ — } d$ with $\Sigma_1 = \{a, b\}$, $\Sigma_2 = \{c\}$ and $\Sigma_3 = \{d\}$, we give below a rejecting run and an accepting run of $\mathcal{A}$. In this picture, each vertex v is labeled by the pair $(\lambda(v), r(v))$. Note that, in order to obtain the states of minimal vertices, we apply transition functions of the form $\delta_{\lambda(v),\emptyset}$.

$$\begin{tikzcd}
\Sigma_1 & b,b \arrow[dr] & & a,\bot \arrow[r] & b,\bot \arrow[dr] & \\
\Sigma_2 & & c,c \arrow[ur] \arrow[dr] & & & c,\bot \\
\Sigma_3 & d,d \arrow[ur] & & d,d \arrow[urr] \arrow[rr] & & d,d
\end{tikzcd}$$

$$\begin{tikzcd}
\Sigma_1 & b,b \arrow[dr] & & b,b \arrow[r] & a,a \arrow[r] & b,b \\
\Sigma_2 & & c,c \arrow[ur] \arrow[dr] & & c,c \arrow[ur] & \\
\Sigma_3 & d,d \arrow[ur] & & d,d \arrow[ur] \arrow[rr] & & d,d
\end{tikzcd}$$

We will show now that the first condition for traces, stating that dependent events cannot occur concurrently, cannot be accepted by a deterministic $(\Sigma_1, \dots, \Sigma_n)$-ACA in general. We consider the dependence alphabet

$(\Sigma, D) = a \text{ --- } b \text{ --- } c$ and we will denote by $b^n ac$ the pomset which consists of a chain of n occurrences of b followed two concurrent events labeled a and c:

$$\begin{tikzcd} & & & & & a \\ b \arrow[r] & b \arrow[r] & b \cdots & b \arrow[r] & b \arrow[ur] \arrow[dr] & \\ & & & & & c \end{tikzcd}$$

Note that this pomset is actually a trace. Let $L = \{b^n ac \mid n > 0\}$ and assume that $L \subsetneq L(\mathcal{A})$ for some deterministic $(\{a, b\}, \{c\})$-ACA $\mathcal{A}$ (or $(\{a\}, \{b\}, \{c\})$-ACA $\mathcal{A}$). We denote by p_n, q_n, s_n the states associated respectively with the events a, c and with the last occurrence of b in the accepting run of $\mathcal{A}$ over $b^n ac$. Since $\mathcal{A}$ is deterministic, the run for $b^n ac$ must be

$$\begin{tikzcd} b, s_1 \arrow[r] & b, s_2 \arrow[r] & b, s_3 & \cdots b, s_{n-1} \arrow[r] & b, s_n \arrow[r] \arrow[dr] & a, p_n \\ & & & & & c, q_n \end{tikzcd}$$

Now, since there are only finitely many states, we must have $s_n = s_m$ for some $0 < n < m$. Then, necessarily $p_n = p_m$ and $q_n = q_m$ and we deduce that

$$\begin{tikzcd} b, s_1 \arrow[r] & b, s_2 & \cdots & b, s_n \arrow[r] \arrow[dr] & b, s_{n+1} \cdots & b, s_m \arrow[r] & a, p_m \\ & & & & c, q_n & & \end{tikzcd}$$

is an accepting run of $\mathcal{A}$. Therefore, $\mathcal{A}$ accepts a pomset which does not satisfy condition (1). This shows both that the language L and the set of pomsets satisfying the first condition of traces cannot be accepted by a deterministic $(\{a, b\}, \{c\})$-ACA (or $(\{a\}, \{b\}, \{c\})$-ACA).

As a third example, we give a non deterministic (Σ_1, Σ_2)-ACA which accepts L. Let $Q_1 = \{1, 2, 3\}$, $Q_2 = \{1\}$, $F = \{(3, 1)\}$ and define the transition functions by

$$\begin{array}{rr} \delta_{b,\emptyset} = \{1\} \qquad & \delta_{b,\{1\}}(1) = \{1, 2\} \\ \delta_{a,\{1\}}(2) = \{3\} \qquad & \delta_{c,\{1\}}(2) = \{1\} \end{array}$$

Then, the pomset $b^n ac$ is accepted by the run

$$\begin{tikzcd} b, 1 \arrow[r] & b, 1 \arrow[r] & b, 1 & \cdots\cdots b, 1 \arrow[r] & b, 2 \arrow[r] \arrow[dr] & a, 3 \\ & & & & & c, 1 \end{tikzcd}$$

We can easily check that this automaton accepts precisely the language L.

We do not know whether condition (1) for traces can be accepted by a non deterministic $(\Sigma_1, \dots, \Sigma_n)$-ACA for general dependence alphabets. We only expect the answer to be negative.

Nevertheless, for the simple dependence alphabet $(\Sigma, D) = a — b — c$, it is possible to accept precisely the set of traces as shown in our last example. We consider the processes $\Sigma_1 = \{a\}$, $\Sigma_2 = \{b\}$ and $\Sigma_3 = \{c\}$. The sets of states of the $(\Sigma_1, \Sigma_2, \Sigma_3)$-ACA $\mathcal{A}$ are $Q_1 = Q_2 = Q_3 = \{a, b\} \times \{b, c\}$ and all possible combination of states are accepting. Intuitively, a state (x, y) claims that among a and b (resp. c and b) the next event will be x (resp. y). The transition functions are the following:

$$
\begin{aligned}
\delta_{b,\emptyset} &= \delta_{b,\{1\}}((b,b)) = \delta_{b,\{2\}}((b,b)) = \delta_{b,\{3\}}((b,b)) \\
&= \delta_{b,\{1,3\}}((b,c),(a,b)) = \{(b,b),(a,b),(b,c),(a,c)\} \\
\delta_{a,\{1\}}((a,b)) &= \delta_{a,\{2\}}((a,b)) = \{(a,b),(b,b)\} \\
\delta_{a,\{1\}}((a,c)) &= \delta_{a,\{2\}}((a,c)) = \{(a,c),(b,c)\} \\
\delta_{c,\{2\}}((b,c)) &= \delta_{c,\{3\}}((b,c)) = \{(b,c),(b,b)\} \\
\delta_{c,\{2\}}((a,c)) &= \delta_{c,\{3\}}((a,c)) = \{(a,c),(a,b)\}
\end{aligned}
$$

Here is a run of this automaton:

$$
\begin{tikzcd}
 & a,(a,c) \arrow[r] & a,(b,c) \arrow[dr] & & & & \\
b,(a,c) \arrow[ur] \arrow[dr] & & & b,(b,b) \arrow[r] & b,(b,c) \arrow[dr] & & b,(a,b) \\
 & c,(a,b) \arrow[urr] & & & & c,(b,b) \arrow[ur] &
\end{tikzcd}
$$

It is easy to see that all traces starting with b can be accepted by $\mathcal{A}$. Although less trivial, the converse is also true. Therefore, $L(\mathcal{A})$ is the set of traces starting with b. By changing the initial condition of the automaton, we can recognize all traces starting with a or with c or with a and c. For instance, if we set $\delta_{b,\emptyset} = \emptyset$ and $\delta_{a,\emptyset} = \{(a,b),(b,b)\}$ we accept all traces starting with a.

As usual with automata, it is easy to see that deterministic $(\Sigma_1, \dots, \Sigma_n)$-ACA are closed under complement (one only has to complement the set of accepting states) and that deterministic or non deterministic $(\Sigma_1, \dots, \Sigma_n)$-ACA are closed under union and intersection (one uses classical direct product constructions, completing first the automata for the union).

We conclude this section with a few remarks. First, the covering of pomsets by the chains formed by the fixed sequential processes is crucial in the definition of asynchronous cellular automata. It allows us to use a fixed number of local states and to determine the read and write domains of the actions using the labeling and the covering relation. The weakest covering is when each Σ_i is a singleton. In this case we have a set of local states per letter as in the asynchronous cellular automata for traces [17, 2]. Note that, even with this trivial covering, our definition is not the same as that of Zielonka for traces. Mainly, in our definition, a run of the ACA is over the Hasse diagram of the pomset whereas with Zielonka's ACA for traces, a run is in fact over the dependence graph of the trace. A dependence graph is an intermediary representation of a trace between its Hasse diagram and its partial order. This intermediary representation is possible thanks to the existence of a static dependence relation over actions. More precisely, our definition of ACA for pomsets and that of Zielonka for traces differ in three respects. First, Zielonka's definition uses a global initial state which in our case is coded in the transition functions of the form $\delta_{x,\emptyset}$. Second, our transition functions only read the states of the processes covered by the current action whereas in Zielonka's definition a fixed set of processes is read even if the last executions of some of these processes are far below the current action. Third, we only read the states of maximal processes to determine whether a run is successful whereas in Zielonka's definition the states of all processes are collected globally to decide acceptance.

Remark 3.3 As discussed below, one could give several alternative definitions of accepting runs. One of these variants corresponds precisely to asynchronous cellular automata for traces.

1. First, one can change the set of processes read by transition functions. In the definition above, a transition only reads the processes covered by the current vertex. On the contrary, one can allow to read all processes which occur in the past of the current vertex by using $R_{\mathrm{occur}}(v) = p \circ \lambda(\{u \in V \mid u < v\})$ instead of $R(v)$. The run r should then satisfy the relation

$$r(v) \in \delta_{\lambda(v),R_{\mathrm{occur}}(v)}\left(r(\partial_i(\Downarrow v))_{i \in R_{\mathrm{occur}}(v)}\right)$$

   In the definition of ACA for traces, we use the trivial covering of Σ by singletons and we identify processes with letters of Σ. Then, we consider a fixed dependence (symmetric and reflexive) relation D on Σ and we use $R_{\mathrm{trace}}(v) = \{a \in \Sigma \mid (a, \lambda(v)) \in D\}$ instead of $R(v)$. This static approach enforces the use of an initial state $\perp$. Indeed, for some $a \in R_{\mathrm{trace}}(v)$, the prefix $\partial_a(\Downarrow v)$ may be empty and thus cannot be identified with a vertex of the trace. Thus we define $r(1) = \perp$ where 1 denotes the empty trace.

2. Second, one can change the acceptance condition. In the definition above, we only read the final state of the maximal processes. One can read the final state of all occurring processes by using $F_{\text{occur}}(t) = p \circ \lambda(V)$ instead of $F(t) = p \circ \lambda(\max(t))$. The accepting condition is then changed to $f_{\text{occur}}(r) = r(\partial_i(t))_{i \in F_{\text{occur}}(t)} \in F$. For traces, we read the final state of all processes by using $F_{\text{trace}}(t) = \Sigma$ instead of $F(t)$. The accepting condition is then $f_{\text{trace}}(r) = r(\partial_a(t))_{a \in \Sigma} \in F$. Again we need the initial state $\perp$ and the convention $r(1) = \perp$ when some prefix $\partial_a(t)$ is empty.

Note that all these variants are not necessarily equivalent. For instance, with $\Sigma_1 = \{a\}$ and $\Sigma_2 = \{b, c\}$, the set of pomsets consisting of an arbitrary sequence of a's followed by one b can be accepted with a deterministic (Σ_1, Σ_2)-ACA using Definition 3.2 whereas it can only be accepted by a non deterministic (Σ_1, Σ_2)-ACA if we read the final states of all occurring processes to determine acceptance. Nevertheless, we will see in Sections 4 and 6 that under some assumptions, these variants are equivalent. In particular, this is true for traces.

4 Asynchronous mapping

Asynchronous mappings were introduced in [2] in order to simplify the construction of ACA for traces. Here we generalize this notion to pomsets. The domain of an asynchronous mapping must be a prefix closed subset of pomsets, that is a subset $\mathbb{Q}$ of pomsets such that for all pairs of pomsets s and t, if $s \preceq t$ and $t \in \mathbb{Q}$ then $s \in \mathbb{Q}$. For instance, $\mathbb{P}(\Sigma_1, \ldots, \Sigma_n)$ and $\mathbb{M}(\Sigma, D)$ are prefix closed sets of pomsets.

Definition 4.1 *Let $\mathbb{Q}$ be a prefix closed set of pomsets and let S be a finite set. A mapping $\sigma : \mathbb{Q} \longrightarrow S$ is asynchronous if for all $t = (V, \leq, \lambda) \in \mathbb{Q}$,*

1. *for all vertices $v \in V$, the value $\sigma(\downarrow v)$ is uniquely determined by $\sigma(\Downarrow v)$ and $\lambda(v)$.*

2. *for all $A, B \subseteq \Sigma$, the value $\sigma(\partial_{A \cup B}(t))$ is uniquely determined by $\sigma(\partial_A(t))$ and $\sigma(\partial_B(t))$.*

A language $L \subseteq \mathbb{Q}$ is recognized by an asynchronous mapping $\sigma : \mathbb{Q} \longrightarrow S$ if $L = \sigma^{-1}(\sigma(L))$.

Remark 4.2 The definition of asynchronous mappings is valid for arbitrary pomsets and does not depend on the existence of sequential processes

$(\Sigma_1, \ldots, \Sigma_n)$ in the pomsets. The definition of asynchronous cellular automata requires at least that each set of vertices labeled with the same letter forms a chain. This simple and weak property is not even required for asynchronous mappings.

Proposition 4.3 *Let $\mathbb{Q} \subseteq \mathbb{P}(\Sigma_1, \ldots, \Sigma_n)$ be a prefix closed set of $(\Sigma_1, \ldots, \Sigma_n)$-pomsets. Let $L \subseteq \mathbb{Q}$ be a language of $(\Sigma_1, \ldots, \Sigma_n)$-pomsets recognized by some asynchronous mapping $\sigma : \mathbb{Q} \longrightarrow S$. Then there exists a deterministic $(\Sigma_1, \ldots, \Sigma_n)$-asynchronous cellular automaton $\mathcal{A}$ such that $L(\mathcal{A}) \cap \mathbb{Q} = L$.*

Proof: This proof follows the same ideas as the corresponding one for traces. Assume that $\sigma : \mathbb{Q} \longrightarrow S$ recognizes the language $L \subseteq \mathbb{Q}$. We define a deterministic $(\Sigma_1, \ldots, \Sigma_n)$-ACA $\mathcal{A}$ as follows.

1. For all $i \in [n]$, let $Q_i = S$,
2. For all $a \in \Sigma$, $J \subseteq [n]$ and $(q_i)_{i \in J} \in S^J$, let

$$\begin{aligned}\delta_{a,J}\left((q_i)_{i \in J}\right) = \{\sigma(t) \mid t \in \mathbb{Q}, t = {\downarrow} v \text{ for some } v \text{ such that}\\ \lambda(v) = a, R(v) = J \text{ and } \sigma(\partial_{\Sigma_i}({\Downarrow} v)) = q_i \text{ for all } i \in J\}\end{aligned}$$

3. $F = \{\sigma(\partial_{\Sigma_i}(t))_{i \in F(t)} \mid t \in L\}$.

Claim 1: $\mathcal{A}$ is deterministic.

Indeed, let $a \in \Sigma$, $J \subseteq [n]$ and $(q_i)_{i \in J} \in S^J$. Choose $t = {\downarrow} v$ and $t' = {\downarrow} v'$ in $\mathbb{Q}$ with $\lambda(v) = \lambda(v')$, $R(v) = J = R(v')$ and $\sigma(\partial_{\Sigma_i}({\Downarrow} v)) = q_i = \sigma(\partial_{\Sigma_i}({\Downarrow} v'))$ for all $i \in J$. We have ${\Downarrow} v = \partial_{(\cup_{i \in J} \Sigma_i)}({\Downarrow} v)$ and ${\Downarrow} v' = \partial_{(\cup_{i \in J} \Sigma_i)}({\Downarrow} v')$. Hence, using the definition of asynchronous mappings we deduce $\sigma({\Downarrow} v) = \sigma({\Downarrow} v')$ and since $\lambda(v) = \lambda(v')$ it follows $\sigma(t) = \sigma({\downarrow} v) = \sigma({\downarrow} v') = \sigma(t')$ which proves the claim.

Claim 2: Let $t = (V, \leq, \lambda) \in \mathbb{Q}$ be a $(\Sigma_1, \ldots, \Sigma_n)$-pomset. Then, the mapping $r : V \longrightarrow \bigcup_{i \in [n]} Q_i$ defined by $r(v) = \sigma({\downarrow} v)$ is the run of $\mathcal{A}$ on t.

One only has to check that this mapping r fullfils the condition of Definition 3.2. This follows directly from the definition of the transition functions of $\mathcal{A}$.

Claim 3: $L(\mathcal{A}) \cap \mathbb{Q} = L$.

Note first that $t = \partial_{(\cup_{i \in F(t)} \Sigma_i)}(t)$ for all $t \in \mathbb{Q}$. Now assume that $t \in L(\mathcal{A})$. Then for the unique run r of $\mathcal{A}$ on t, we have $f(r) \in F$ and there exists $t' \in L$

such that $F(t) = F(t')$ and $\sigma(\partial_{\Sigma_i}(t)) = \sigma(\partial_{\Sigma_i}(t'))$ for all $i \in F(t) = F(t')$. Since σ is asynchronous, it follows $\sigma(t) = \sigma(t')$. Therefore, $t \in \sigma^{-1}(\sigma(L)) = L$ which proves one inclusion. The converse is trivial. $\square$

Remark 4.4 Proposition 4.3 holds also for the alternative definitions of accepting runs discussed in Remark 3.3. The proofs remain essentially the same. One only has to change accordingly the definitions of the transition functions and of the final set.

Note that, for trace languages, the converse of Proposition 4.3 is also true implying that all alternative definitions of ACA are equivalent for traces. Indeed, it is easy to show that a trace language accepted by an ACA is a recognizable trace language, whatever alternative chosen for accepting runs. Moreover, if L is a recognizable trace language, the existence of an asynchronous mapping which recognizes L was proven in [2]. Finally, as mentioned above, from this asynchronous mapping one can easily get an ACA which accepts L, whatever alternative chosen for accepting runs.

The equivalence between alternative definitions of accepting runs will be extended to a more general class of pomsets in Section 6. On the other hand, in the general setting of $\mathbb{P}(\Sigma_1, \ldots, \Sigma_n)$, the converse of Proposition 4.3 is still open.

5 From ACA to MSOL

In this section, we will define monadic second order (MSO) formulas and their interpretations over pomsets. We will then prove that for all ACA $\mathcal{A}$ (deterministic or not), there exists a MSO formula which defines the language accepted by $\mathcal{A}$.

Let Σ be a finite alphabet. The MSO language over Σ that we consider consists of the unary predicates $(P_a)_{a\in\Sigma}$, a binary predicate R, first order variables $x, y, z, \ldots$, monadic second order variables $X, Y, Z, \ldots$, boolean connectives $\neg, \vee, \wedge, \longrightarrow, \longleftrightarrow$ and quantifiers $\exists, \forall$. A sentence is a formula without free variables. For instance, the following formulas are first order and MSO sentences respectively.

$$\begin{aligned}
\varphi_1 &::= \exists x(P_a(x) \wedge \forall y(R(x,y) \longrightarrow \neg P_b(y))) \\
\varphi_2 &::= \exists X \exists Y(\forall x(x \in X \vee x \in Y) \wedge \exists x\; x \in X \wedge \exists y\; y \in Y \\
&\qquad\qquad \wedge \forall x \forall y(x \in X \wedge y \in Y \longrightarrow \neg R(x,y) \wedge \neg R(y,x))
\end{aligned}$$

A pomset $t = (V, \leq, \lambda)$ can be seen as an interpretation of this MSO language as follows: the domain is the set V of vertices, i.e. first order variables range over vertices and MSO variables range over sets of vertices; for all $a \in \Sigma$, $P_a(x)$ means $\lambda(x) = a$ and $R(x, y)$ means $x \leq y$. We say that a pomset t satisfies a sentence φ, denoted by $t \models \varphi$, when φ is true for the interpretation defined by t. The set of pomsets which satisfy a sentence φ is denoted by $L(\varphi)$. For instance, $L(\varphi_1)$ is the set of pomsets which have a vertex labeled by a with no vertex labeled by b above and $L(\varphi_2)$ is the set of non connected pomsets.

In order to make the formulas more readable, we will write $x \leq y$ for $R(x, y)$ and $\lambda(x) = a$ for $P_a(x)$. For instance, the formula φ_1 will be written $\exists x(\lambda(x) = a \wedge \forall y(x \leq y \longrightarrow \lambda(y) \neq b))$. Moreover, we will use several abbreviations which can be easily translated in our MSO language. For instance, we will write

$$\begin{aligned}
x < y \quad &\text{for} \quad x \leq y \wedge \neg y \leq x \\
x \multimap\!\!< y \quad &\text{for} \quad x < y \wedge \neg \exists z(x < z \wedge z < y) \\
\lambda(x) \in A \quad &\text{for} \quad \bigvee_{a \in A} \lambda(x) = a \\
p \circ \lambda(x) = p \circ \lambda(y) \quad &\text{for} \quad \bigvee_{1 \leq i \leq n} (\lambda(x) \in \Sigma_i \wedge \lambda(y) \in \Sigma_i) \\
X \cap Y = \emptyset \quad &\text{for} \quad \neg \exists x(x \in X \wedge x \in Y)
\end{aligned}$$

Note that the language defined by a formula can contain pomsets with auto-concurrency (concurrent vertices with the same label). We do not need to put restrictions on the pomsets defined by a formula because all restrictions we need can be described by MSO formulas. For instance the set $\mathbb{P}(\Sigma_1, \ldots, \Sigma_n)$ of $(\Sigma_1, \ldots, \Sigma_n)$-pomsets is defined by the formula

$$\varphi_{(\Sigma_1, \ldots, \Sigma_n)} ::= \forall x \forall y (p \circ \lambda(x) = p \circ \lambda(y) \longrightarrow (x \leq y \vee y \leq x))$$

and the set $\mathbb{M}(\Sigma, D)$ of traces over a dependence alphabet is defined by the formula

$$\forall x \forall y \, [(\lambda(x), \lambda(y)) \in D \longrightarrow (x \leq y \vee y \leq x)] \wedge [x \multimap\!\!< y \longrightarrow (\lambda(x), \lambda(y)) \in D]$$

where $\lambda(x), \lambda(y)) \in D$ stands for the formula $\bigvee_{(a,b) \in D} (\lambda(x) = a \wedge \lambda(y) = b)$.

We are now ready to state

Theorem 5.1 *Let $\Sigma = \Sigma_1 \dot\cup \cdots \dot\cup \Sigma_n$ and let $\mathcal{A}$ be a possibly non deterministic $(\Sigma_1, \ldots, \Sigma_n)$-ACA. There exists an MSO formula φ over Σ such that*

$$L(\varphi) = L(\mathcal{A}).$$

Proof:

Let $\mathcal{A} = ((Q_i)_{i\in[n]}, (\delta_{a,J})_{a\in\Sigma,J\subseteq[n]}, F)$ be a $(\Sigma_1,\dots,\Sigma_n)$-ACA. We will construct a MSO formula which will be satisfied exactly by those $(\Sigma_1,\dots,\Sigma_n)$-pomsets accepted by $\mathcal{A}$. Let k be the number of states in $\bigcup_{i\in[n]} Q_i$. We may assume that $\bigcup_{i\in[n]} Q_i = [k] = \{1,\dots,k\}$. The following formula claims the existence of an accepting run of the automaton.

$$\psi ::= \exists X_1 \dots \exists X_k \left(\text{partition}(X_1,\dots,X_k) \wedge \left(\forall x \ \text{transition}(x)\right) \wedge \text{accepted}\right)$$

We will now explain this formula and give the sub-formulas *partition*, *transition* and *accepted*. A run over a pomset $t = (V, \leq, \lambda)$ is coded by the MSO variables $X_1,\dots,X_k$. More precisely, X_i stands for the set of vertices mapped on the state i by the run. The formula partition$(X_1,\dots,X_k)$ makes sure that the MSO variables $X_1,\dots,X_k$ do describe a mapping from V to $\bigcup_{i\in[n]} Q_i$.

$$\text{partition}(X_1,\dots,X_k) ::= \left(\forall x \bigvee_{i\in[k]} x \in X_i\right) \wedge \left(\bigwedge_{1\leq i\neq j\leq k} X_i \cap X_j = \emptyset\right)$$

Then, we have to claim that this labeling of vertices by states agrees with the transition functions of the automaton.

$$\begin{aligned}\text{transition}(x) ::= \bigvee \Big(&\lambda(x) = a \wedge x \in X_q \wedge \forall y \ (y \prec x \longrightarrow p \circ \lambda(y) \in J) \\ &\wedge \bigwedge_{i\in J} \exists y \ (y \prec x \wedge p \circ \lambda(y) = i \wedge y \in X_{q_i})\Big)\end{aligned}$$

where the disjunction ranges over all letters $a \in \Sigma$, states $q \in Q_{p(a)}$, subsets $J \subseteq [n]$ and tuples $(q_i)_{i\in J} \in \prod_{i\in J} Q_i$ such that $q \in \delta_{a,J}((q_i)_{i\in J})$.

It remains to state that the run reaches a final state of the automaton.

$$\begin{aligned}\text{accepted} ::= \bigvee_{(f_i)_{i\in J}\in F} \Big(&\forall x \ ((\neg\exists y \ x < y) \longrightarrow p \circ \lambda(x) \in J) \\ &\wedge \bigwedge_{i\in J} \exists x\big((\neg\exists y \ x < y) \wedge p \circ \lambda(x) = i \wedge x \in X_{f_i}\big)\Big)\end{aligned}$$

In fact, the formula ψ describes an accepting run of the automaton only for $(\Sigma_1,\dots,\Sigma_n)$-pomsets. Therefore, we need in addition the formula $\varphi_{(\Sigma_1,\dots,\Sigma_n)}$ described above. Finally, the theorem follows from the

Claim:

$$L(\varphi_{(\Sigma_1,\dots,\Sigma_n)} \wedge \psi) = L(\mathcal{A})$$

$\square$

Remark 5.2 Clearly, Theorem 5.1 holds also for the alternative definitions of accepting runs described in Remark 3.3. The proof is essentially the same. One only has to change the formulas *transition* and *accepted* accordingly.

The converse of Theorem 5.1 does not hold in general as shown by a simple example due to Dietrich Kuske [9]: the first order sentence stating that every a is covered by a b cannot be checked by a (non deterministic) $(\Sigma_1, \dots, \Sigma_n)$-ACA if a and b belong to different processes.

6 From MSOL to deterministic ACA

In this section, we prove that the converse of Theorem 5.1 holds for the special subclass of $(\Sigma_1, \dots, \Sigma_n)$-pomsets which satisfy the CROW axiom defined below.

Definition 6.1 *A $(\Sigma_1, \dots, \Sigma_n)$-pomset $t = (V, \leq, \lambda)$ satisfies the Concurrent Read and Exclusive Owner Write (CROW) axiom if for all $x, y, z \in V$,*

$$x \prec y \wedge x < z \wedge y \parallel z \quad \Longrightarrow \quad p \circ \lambda(x) \neq p \circ \lambda(z) \tag{3}$$

The set of $(\Sigma_1, \dots, \Sigma_n)$-pomsets which satisfy the CROW axiom is denoted by $\mathbb{CROW}(\Sigma_1, \dots, \Sigma_n)$.

A possible interpretation of this axiom is to think of the ACA as a Concurrent Read and Exlusive Owner Write (CROW) machine. More precisely, we consider n processes whose sets of actions are $\Sigma_1, \dots, \Sigma_n$ respectively. Each process has a memory which can be read by all actions but can be written by its own actions only (Owner Write). We allow concurrent reads of memories but no concurrent writes. As quoted in Section 2.1, this restriction is already enforced by the very definition of $(\Sigma_1, \dots, \Sigma_n)$-pomsets. Without further restrictions, two concurrent events may respectively read from and write to the same location. This is the case when there exist two concurrent events $y \parallel z$ such that z writes in the memory of some process i ($p \circ \lambda(z) = i$) and y reads the memory of this process i ($p \circ \lambda(x) = i$ for some $x \prec y$). This is precisely the situation which is forbidden by the CROW axiom.

Theorem 6.2 *Let $\Sigma = \Sigma_1 \dot\cup \cdots \dot\cup \Sigma_n$ and let φ be a MSO formula over Σ. There exists a deterministic $(\Sigma_1, \dots, \Sigma_n)$-ACA $\mathcal{A}$ such that*

$$L(\varphi) \cap \mathbb{CROW}(\Sigma_1, \dots, \Sigma_n) = L(\mathcal{A}) \cap \mathbb{CROW}(\Sigma_1, \dots, \Sigma_n).$$

In order to prove this theorem, one can use an induction on the structure of the formula. Disjunction and existential quantification are easily dealt with when non deterministic ACA are allowed. On the other hand, complement is easy for deterministic ACA. Whence the core of such an approach is the determinization of ACA. For this problem, starting from a non deterministic ACA $\mathcal{A}$, one can directly construct an asynchronous mapping which accepts the language $L(\mathcal{A})$ and then use Proposition 4.3. This construction is similar to that of [12] and uses the asynchronous time stamping ν of Cori, Metivier and Zielonka [2] but the proofs are more involved. In particular, it is known that for traces the mapping ν is asynchronous by itself [2, 4] but this is not the case for $\mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$-pomsets. Here we give a simpler proof which uses Zielonka's theorem. For this, we first map CROW-pomsets into traces by simply changing the labeling.

Let $\Sigma' = \Sigma \times \mathcal{P}([n])$ be a new set of labels and for all $i \in [n]$, let $\Sigma'_i = \Sigma_i \times \mathcal{P}([n])$ be the associated new processes. We define an embedding g from $\mathbb{P}(\Sigma_1, \ldots, \Sigma_n)$ into $\mathbb{P}(\Sigma'_1, \ldots, \Sigma'_n)$ by $g(V, \leq, \lambda) = (V, \leq, \lambda')$ where for all $v \in V, \lambda'(v) = (\lambda(v), R(v))$. Note that g is well defined, since for all $i \in [n]$, $\lambda'^{-1}(\Sigma'_i)$ is totally ordered. Let D' be the dependence relation defined on Σ' by

$$D' = \{((a, A), (b, B)) \mid p(a) = p(b) \vee p(a) \in B \vee p(b) \in A\}.$$

Proposition 6.3

$$\mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n) = g^{-1}(\mathbb{M}(\Sigma', D'))$$

Proof: We first prove that $\mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n) \subseteq g^{-1}(\mathbb{M}(\Sigma', D'))$. Let $t = (V, \leq, \lambda) \in \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$ and let $g(t) = (V, \leq, \lambda')$. Let $x, y \in V$ and assume that $x -\!\!< y$. Then, $p \circ \lambda(x) \in R(y)$ and it follows $(\lambda'(x), \lambda'(y)) \in D'$. Now, let $y, z \in V$ and assume that $(\lambda'(y), \lambda'(z)) \in D'$. If $p \circ \lambda(y) = p \circ \lambda(z)$ then $y \not\parallel z$ since t is a $(\Sigma_1, \ldots, \Sigma_n)$-pomset. Otherwise, we have for instance $p \circ \lambda(z) \in R(y)$. Hence, there exists $x \in V$ such that $x -\!\!< y$ and $p \circ \lambda(x) = p \circ \lambda(z)$. Therefore, x and z must be ordered. Since $x < z$ would contradict the CROW-axiom, it follows $z \leq x$, whence $z < y$. Therefore, $g(t) \in \mathbb{M}(\Sigma', D')$ (see Section 2.2) and it follows $t \in g^{-1}(\mathbb{M}(\Sigma', D'))$.

Conversely, let $t = (V, \leq, \lambda) \in g^{-1}(\mathbb{M}(\Sigma', D'))$ and let $g(t) = (V, \leq, \lambda')$. Let $x, y, z \in V$ be such that $x -\!\!< y \wedge x < z \wedge y \parallel z$. By definition, $p \circ \lambda(x) \in R(y)$ and $(\lambda'(y), \lambda'(z)) \notin D'$. Therefore, $p \circ \lambda(z) \notin R(y)$ and it follows $p \circ \lambda(x) \neq p \circ \lambda(z)$. $\square$

Proposition 6.4 *Let φ be a MSO formula over Σ. There exists a MSO formula φ' over Σ' such that*

$$L(\varphi) \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n) = g^{-1}(L(\varphi') \cap \mathbb{M}(\Sigma', D'))$$

Proof: Let φ be a MSO formula over Σ. Let φ' be the MSO formula over Σ' obtained from φ by substituting for atomic formulas of the form $x \in P_a$ the disjunction $\bigvee_{J \subseteq [n]} x \in P_{(a,J)}$:

$$\varphi' = \varphi \left[\bigvee_{J \subseteq [n]} x \in P_{(a,J)} \Big/ x \in P_a \right].$$

Let $t = (V, \leq, \lambda) \in L(\varphi) \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$. We have $g(t) = (V, \leq, \lambda') \in \mathbb{M}(\Sigma', D')$ by Proposition 6.3 and it remains to show that $g(t) \models \varphi'$. This is clear since $\lambda(x) = a$ if and only if $\lambda'(x) = (a, J)$ for some $J \subseteq [n]$. The converse can be shown similarly. $\square$

Proposition 6.5 *Let $\mathcal{A}'$ be a (deterministic) $(\Sigma'_1, \ldots, \Sigma'_n)$-ACA. There exists a (deterministic) $(\Sigma_1, \ldots, \Sigma_n)$-ACA $\mathcal{A}$ such that $L(\mathcal{A}) = g^{-1}(L(\mathcal{A}'))$.*

Proof: Let $\mathcal{A}' = ((Q_i)_{i \in [n]}, (\delta'_{a',J})_{a' \in \Sigma', J \subseteq [n]}, F)$ be a $(\Sigma'_1, \ldots, \Sigma'_n)$-ACA. For all $a \in \Sigma$ and $J \subseteq [n]$, let $\delta_{a,J} = \delta'_{(a,J),J}$. We claim that the automaton $\mathcal{A} = ((Q_i)_{i \in [n]}, (\delta_{a,J})_{a \in \Sigma, J \subseteq [n]}, F)$ is the required $(\Sigma_1, \ldots, \Sigma_n)$-ACA. Note that if $\mathcal{A}'$ is deterministic then so is $\mathcal{A}$.

We first show that in order to accept a pomset in $g(\mathbb{P}(\Sigma_1, \ldots, \Sigma_n))$ the ACA $\mathcal{A}'$ only uses transition functions of the form $\delta_{(a,J),J}$. Indeed, let $t = (V, \leq, \lambda) \in \mathbb{P}(\Sigma_1, \ldots, \Sigma_n)$ and let $g(t) = (V, \leq, \lambda')$. For all $v \in V$, we have $R'(v) = p \circ \lambda'(\{u \mid u \prec v\}) = p \circ \lambda(\{u \mid u \prec v\}) = R(v)$. Therefore, in a run of $\mathcal{A}'$ on $g(t)$ the transition functions used are of the form $\delta'_{\lambda'(v),R'(v)} = \delta'_{(\lambda(v),R(v)),R(v)} = \delta_{\lambda(v),R(v)}$.

It follows that a mapping $r : V \longrightarrow \bigcup_{i \in [n]} Q_i$ is an accepting run of $\mathcal{A}'$ on $g(t)$ if and only if it is an accepting run of $\mathcal{A}$ on t, that is,

$$t \in L(\mathcal{A}) \Longleftrightarrow g(t) \in L(\mathcal{A}') \Longleftrightarrow t \in g^{-1}(L(\mathcal{A}')).$$

The proposition follows. $\square$

Proof of Theorem 6.2: Let φ be a MSO formula over Σ. By Proposition 6.4, there exists a MSO formula φ' over Σ' such that

$$L(\varphi) \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n) = g^{-1}(L(\varphi') \cap \mathbb{M}(\Sigma', D')).$$

The language $L(\varphi') \cap \mathbb{M}(\Sigma', D')$ is a recognizable trace language [15]. Hence [2], there exists an asynchronous mapping σ from $\mathbb{M}(\Sigma', D')$ into a finite set which recognizes $L(\varphi') \cap \mathbb{M}(\Sigma', D')$. By Proposition 4.3, there exists a deterministic $(\Sigma'_1, \ldots, \Sigma'_n)$-ACA $\mathcal{A}'$ such that $L(\mathcal{A}') \cap \mathbb{M}(\Sigma', D') = L(\varphi') \cap \mathbb{M}(\Sigma', D')$. It follows by Proposition 6.5 that there exists a deterministic $(\Sigma_1, \ldots, \Sigma_n)$-ACA $\mathcal{A}$ such that $L(\mathcal{A}) = g^{-1}(L(\mathcal{A}'))$. Finally, applying Proposition 6.3 we obtain

$$\begin{aligned} L(\mathcal{A}) \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n) &= g^{-1}(L(\mathcal{A}')) \cap g^{-1}(\mathbb{M}(\Sigma', D')) \\ &= g^{-1}(L(\mathcal{A}') \cap \mathbb{M}(\Sigma', D')) \\ &= g^{-1}(L(\varphi') \cap \mathbb{M}(\Sigma', D')) \\ &= L(\varphi) \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n) \end{aligned}$$

□

As a corollary of Theorems 5.1 and 6.2 we obtain that MSO formulas, non deterministic $(\Sigma_1, \ldots, \Sigma_n)$-ACA and deterministic $(\Sigma_1, \ldots, \Sigma_n)$-ACA have the same expressive power for $\mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$-pomsets.

Theorem 6.6 *Let $L \subseteq \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$. The following are equivalent:*

1. *L is definable by a MSO formula,*
2. *there exists a non deterministic $(\Sigma_1, \ldots, \Sigma_n)$-ACA $\mathcal{A}$ such that*

$$L = L(\mathcal{A}) \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$$

3. *there exists a deterministic $(\Sigma_1, \ldots, \Sigma_n)$-ACA $\mathcal{A}$ such that*

$$L = L(\mathcal{A}) \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$$

Remark 6.7 Finally, we come back to the alternative definitions of accepting runs discussed in Remark 3.3. As a corollary of previous results, we obtain that these alternatives are equivalent for CROW-pomsets. Indeed, starting from a $(\Sigma_1, \ldots, \Sigma_n)$-ACA with one definition of accepting runs, one first builds a corresponding MSOL formula φ (Remark 5.2). Then, we use the proof of Theorem 6.2 except that when we build the ACA $\mathcal{A}'$ from an asynchronous mapping we choose another definition of accepting runs (Remark 4.4). Note that Proposition 6.5 holds whatever alternative definition chosen for accepting runs.

7 Conclusion

We believe that the CROW axiom is really natural if one sees an ACA as an abstract representation of a parallel machine where each process writes in its own memory and can read the memories of other processes. In this case, we have proved that deterministic ACA are closed under the boolean operations and have the same expressive power as the MSOL. Since emptiness for ACA is decidable when restricted to $\mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$ pomsets, we can use ACA to perform model checking: to check whether an implementation (an ACA) $\mathcal{A}$ satisfies a specification (a MSO formula) φ, one computes the deterministic ACA $\mathcal{B}$ such that $L(\varphi) \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n) = L(\mathcal{B}) \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$ and then checks for emptiness $L(\mathcal{A}) \cap \overline{L(\mathcal{B})} \cap \mathbb{CROW}(\Sigma_1, \ldots, \Sigma_n)$.

On the other hand, ACA may be seen as an abstract representation of distributed systems communicating asynchronously. In this setting, the covering relation $x \prec y$ can denote a message sent by x to y. If the communication is asynchronous, the process $p(x)$ may perform some actions concurrently to the reception (y) of the message sent (x). In this case, the CROW axiom is not natural anymore. Therefore, an important open problem is to know whether the properties proved in this paper for CROW-pomsets also hold in a more general setting.

Finally, we would like to point out that both the definition of asynchronous mappings and of MSOL can be extended without any change to pomsets with auto-concurrency. It would be very interesting to find a general model of ACA which allows auto-concurrency. Note that it is possible with our ACA to cope with bounded auto-concurrency by suitably relabeling auto-concurrent vertices.

References

[1] F. Bracho, M. Droste, and D. Kuske. Dependence orders for computations of concurrent automata. *Theoretical Computer Science*, 174:67–96, 1997.

[2] R. Cori, Y. Métivier, and W. Zielonka. Asynchronous mappings and asynchronous cellular automata. *Information and Computation*, 106:159–202, 1993.

[3] V. Diekert. A partial trace semantics for petri nets. *Theoretical Computer Science*, 113:87–105, 1994. Special issue of ICWLC 92, Kyoto (Japan).

[4] V. Diekert and A. Muscholl. Construction of asynchronous automata. In G. Rozenberg and V. Diekert, editors, *Book of Traces*, pages 249–267. World Scientific, Singapore, 1995.

[5] V. Diekert and G. Rozenberg, editors. *Book of Traces*. World Scientific, Singapore, 1995.

[6] M. Droste and P. Gastin. Asynchronous cellular automata for pomsets without auto-concurrency. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, number 1119 in Lecture Notes in Computer Science, pages 627–638. Springer Verlag, 1996.

[7] M. Droste and D. Kuske. Logical definability of recognizable and aperiodic languages in concurrency monoids. In *Proceedings of CSL'95*, number 1092 in Lecture Notes in Computer Science, pages 233–251. Springer Verlag, 1996.

[8] J.L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61:199–224, 1988.

[9] D. Kuske. A note on first order logic and asynchronous cellular automata for pomsets. Personal communication, 1997.

[10] A. Mazurkiewicz. Concurrent program schemes and their interpretations. Tech. rep. DAIMI PB 78, Aarhus University, 1977.

[11] A. Mazurkiewicz. Trace theory. In W. Brauer et al., editors, *Advances in Petri Nets'86*, number 255 in Lecture Notes in Computer Science, pages 279–324. Springer Verlag, 1987.

[12] A. Muscholl. On the complementation of Büchi asynchronous cellular automata. In S. Abiteboul and E. Shamir, editors, *Proceedings of the 21st International Colloquium on Automata, Languages and Programming (ICALP'94)*, number 820 in Lecture Notes in Computer Science. Springer Verlag, 1994.

[13] V.R. Pratt. Modelling concurrency with partial orders. *J. of Parallel Programming*, 15:33–71, 1987.

[14] P.H. Starke. Processes in petri nets. *EIK*, 17:389–416, 1981.

[15] W. Thomas. On logical definability of trace languages. In V. Diekert, editor, *Proceedings of a workshop of the ESPRIT BRA No 3166: Algebraic*

and Syntactic Methods in Computer Science (ASMICS), 1989, Report TUM-I9002, Technical University of Munich, pages 172–182, 1990.

[16] W. Zielonka. Notes on finite asynchronous automata. *R.A.I.R.O. — Informatique Théorique et Applications*, 21:99–135, 1987.

[17] W. Zielonka. Safe executions of recognizable trace languages by asynchronous automata. In A. R. Meyer et al., editors, *Proceedings of the Symposium on Logical Foundations of Computer Science (Logic at Botik'89)*, number 363 in Lecture Notes in Computer Science, pages 278–289. Springer Verlag, 1989.