Adding Reconfiguration to Zielonka's Asynchronous Automata

Citation for the original published paper (version of record):
Lehaut, M., Piterman, N. (2024). Adding Reconfiguration to Zielonka's Asynchronous Automata. Electronic Proceedings in Theoretical Computer Science, EPTCS, 409: 88-102.
http://dx.doi.org/10.4204/EPTCS.409.10

Adding Reconfiguration to Zielonka's Asynchronous Automata *

Mathieu Lehaut
University of Gothenburg, Gothenburg, Sweden
lehaut@chalmers.se

Nir Piterman
University of Gothenburg, Gothenburg, Sweden
piterman@chalmers.se

We study an extension of Zielonka's (fixed) asynchronous automata called reconfigurable asynchronous automata where processes can dynamically change who they communicate with. We show that reconfigurable asynchronous automata are not more expressive than fixed asynchronous automata by giving translations from one to the other. However, going from reconfigurable to fixed comes at the cost of disseminating communication (and knowledge) to all processes in the system. We then show that this is unavoidable by describing a language accepted by a reconfigurable automaton such that in every equivalent fixed automaton, every process must either be aware of all communication or be irrelevant.

1 Introduction

In recent years, computation devices have become so widely available that they are now everywhere. They are lighter, cheaper, prevalent, and, ultimately, mobile. Sensor networks, multi-agent systems, and robot teams use mobile and ad-hoc networks. In such networks, participants/agents/processes come and go and change the communication configuration based on need, location, and various restrictions. These systems force us to consider how communication changes when participants are numerous, mobile, and required to collaborate.

We consider a canonical formalism in language theory for distributed systems with a fixed communication structure – Zielonka's *asynchronous automata*. These are a well known model that supports distribution of language recognition under a fixed communication topology. In this model, a number of processes are each connected to some fixed set of channels. They can then communicate with others processes by synchronizing on a channel. During such a communication, all processes involved share their local states with each other, and then progress to a new state. Another feature of this model is that a communication can only happen if all processes involved are ready for it; if even a single process does not accept then the communication is blocked. The model is especially interesting due to Zielonka's seminal result on the ability to distribute languages in this model [9]. Zielonka's result starts from a given regular language and a target (fixed) distribution of the alphabet. He then shows that if the deterministic automaton for the language satisfies a simple condition about independence of communications then the language can be distributed and accepted by a distributed team of processes. Zielonka's quite involved construction has been revisited and optimized several times, let us cite e.g. [8, 5, 4] for the general construction and [6] for an example of a simpler construction in a restricted case. This result lead to several applications notably in synthesis [7], further establishing the usefulness of this model.

The aim of this paper is to extend the power of asynchronous automata by giving them reconfigurability. To this end, processes comprising a system are extended with the ability to dynamically connect and

*Supported by the ERC Consolidator grant D-SynMA (No. 772459).

A. Achilleos and A. Francalanza (Eds.): Fifteenth International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 88–102, doi:10.4204/EPTCS.409.10

disconnect from channels after a communication. As before, a communication can only occur on a channel if all the processes that are connected to the channel agree on it, and otherwise the communication is blocked. This is the exact notion of communication of asynchronous automata, except that processes can now connect and disconnect to channels dynamically during an execution. In order to allow more than mere synchronization on the channel, communications are extended by a data value, which correspond to the state sharing of asynchronous automata. We call this variant *reconfigurable asynchronous automata*. They are inspired by attribute-based communication calculus [2, 1] and channeled transition systems [3], though they are much simpler than those and adapted to the context of asynchronous automata. To prevent confusions, we sometimes refer to the base variant as *fixed* (as opposed to reconfigurable) asynchronous automata.

With the definition of this new extension, the first natural question is whether reconfigurable asynchronous automata are more expressive than the fixed variant. To this we answer negatively by showing how to translate from one model to the other. Going from fixed to reconfigurable is easy. We also show that if the fixed asynchronous automaton has local transitions, i.e. the next state of a process only depends on its own current state and not on the state of others, then it corresponds to a reconfigurable asynchronous automaton that does not use data values during communications. The other direction is also relatively easy, however with an important caveat: every process in the fixed automaton participates in every communication, autonomously deciding which communications to ignore and which to act upon.

With the two models being equi-expressive, the second natural question is what does adding reconfigurability actually bring? It is well known that non-deterministic finite automata are as expressive as deterministic ones, but can be exponentially smaller in size. In the case of fixed versus reconfigurable asynchronous automata, the gain is not in size, but in the communication structure. As explained just before, our translation from reconfigurable to fixed asynchronous automata results in essentially sharing all information to all parts of the system, and letting each process decide whether that information is actually needed. This is undesirable for many reasons. First, in real systems, each communication takes time and costs energy to process, so one should not waste resources sending information that would be useless to some process. Second, for privacy reasons, it is obviously not desirable that every process in the system has access to every communication; we would rather that a process only receives information based on its "need-to-know". Thirdly, it implies that every process is always connected to every other process in the system, which is not good in systems with a high number of participants that only require a small number of connections at each time.

We then show that this sharing is, in general, unavoidable. We suggest a language that can be recognized by reconfigurable asynchronous automata but for which any equivalent fixed asynchronous automaton has the pitfall described above. In this language, using reconfigurable communication, processes actively connect and disconnect from channels and keep themselves informed only about crucial information. Throughout, processes are connected to a very small number of channels that is independent of system size. However, some (changing) channels are used for coordination of how to connect and disconnect from the other channels. We show that for asynchronous automata to recognize the same language, some processes must be connected to the full set of channels and be informed of everything. What's more, every process that is not connected to the full set of channels can be made trivial by accepting unconditionally all possible communication on channels that they are connected to. We also show that the system contains a subsystem performing the same computation in which the processes that are not fully connected are completely trivial: the system does not need them at all to perform exactly the same computation.

The rest of the paper is organized as follows. In Section 2 we recall the definition of Zielonka's

asynchronous automata and give the definition of reconfigurable asynchronous automata. In Section 3 we give the translations between the models and show that the data of reconfigurable asynchronous automata correspond to the global transitions of fixed asynchronous automata. We then show in Section 4 that in every translation that removes the reconfigurability, all processes either know everything or are trivial. Finally, we conclude and discuss our results in Section 5.

2 Definitions

2.1 Fixed Communication Structure

2.1.1 Distributed Alphabets

We fix a finite set $\mathbb{P}$ of processes. Let Σ be a finite alphabet, and $dom : \Sigma \to 2^{\mathbb{P}}$ a domain function associating each letter with the subset of processes listening to that letter. The pair (Σ, dom) is called a distributed alphabet. We let $dom^{-1}(p) = \{a \in \Sigma \mid p \in dom(a)\}$. It induces an independence binary relation I in the following way: $(a,b) \in I \Leftrightarrow dom(a) \cap dom(b) = \emptyset$. Two words $u = u_1 \dots u_n$ and $v = v_1 \dots v_n$ are said to be equivalent, denoted by $u \sim v$, if one can start from u, repeatedly switch two consecutive independent letters, and end up with v. Let us denote by $[u]$ the equivalence class of a word u. Let $\mathscr{A} = (Q, \Sigma, q_0, \Delta, F)$ be a deterministic automaton over Σ. We say that $\mathscr{A}$ is I-diamond if for all pairs of independent letters $(a,b) \in I$ and all states $q \in Q$, we have $\Delta(q, ab) = \Delta(q, ba)$. If $\mathscr{A}$ has this property, then a word u is accepted by $\mathscr{A}$ if and only if all words in $[u]$ are accepted. Zielonka's result states that an I-diamond automaton can be distributed to processes who are connected to channels according to dom [9].

2.1.2 Asynchronous Automata

An *asynchronous automaton* (in short: AA) [9] over distributed alphabet (Σ, dom) and processes $\mathbb{P}$ is a tuple

$$\mathscr{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma}, \mathrm{Acc})$$

such that:

- S_p is the finite set of states for process p, and $s_p^0 \in S_p$ is its initial state,
- $\delta_a : \prod_{p\in dom(a)} S_p \to \prod_{p\in dom(a)} S_p$ is a partial transition function for letter a that only depends on the states of processes in $dom(a)$ and leaves those outside unchanged,
- $\mathrm{Acc} \subseteq \prod_{p\in\mathbb{P}} S_p$ is a set of accepting states.

A global state of the automaton is of the form $\mathbf{s} = (s_p)_{p\in\mathbb{P}}$, giving the state of each process. For every such global state and every subset $P \subseteq \mathbb{P}$, we denote by $\mathbf{s} \downarrow_P = (s_p)_{p\in P}$ the subset of $\mathbf{s}$ of states from processes in P. Then a run of $\mathscr{B}$ is a sequence $\mathbf{s}_0 a_1 \mathbf{s}_1 a_2 \dots \mathbf{s}_n$ where for all $0 < i \leq n$, $\mathbf{s}_i \in \prod_{p\in\mathbb{P}} S_p$, $a_i \in \Sigma$, satisfying $\mathbf{s}_0 = (s_p^0)_{p\in\mathbb{P}}$ and the following relation:

$$\mathbf{s}_i \downarrow_{dom(a_i)} = \delta_{a_i}(\mathbf{s}_{i-1} \downarrow_{dom(a_i)}) \text{ and } \mathbf{s}_i \downarrow_{\mathbb{P}\setminus dom(a_i)} = \mathbf{s}_{i-1} \downarrow_{\mathbb{P}\setminus dom(a_i)}$$

A run is accepting if $\mathbf{s}_n$ belongs to Acc. The word $a_1 a_2 \dots$ is accepted by $\mathscr{B}$ if such an accepting run exists (note that automata are deterministic but runs on certain words may not exist). The language of $\mathscr{B}$, denoted by $\mathscr{L}(\mathscr{B})$, is the set of words accepted by $\mathscr{B}$. For the rest of this paper, we will drop the Acc component as we focus more on the runs themselves over whether they can reach a certain target. That is, we assume that $\mathrm{Acc} = \prod_{p\in\mathbb{P}} S_p$. This restricts the languages that can be recognized by asynchronous automata but still allows us to prove all our results.

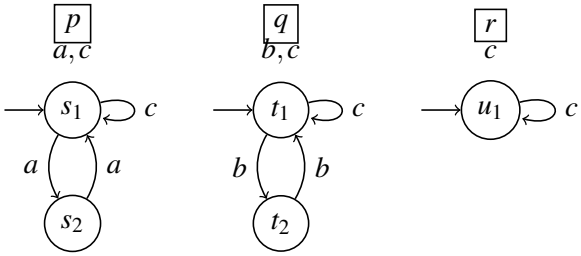

Figure 1: An asynchronous automaton $\mathscr{B}$ over three processes.

Example 1. We give an example of an asynchronous automaton $\mathscr{B}$ in Figure 1. There are three letters $\Sigma = \{a,b,c\}$ distributed over three processes $\mathbb{P} = \{p,q,r\}$ with the domain: $dom(a) = \{p\}, dom(b) = \{q\}, dom(c) = \mathbb{P}$. An example of a run is the following sequence:

$$(s_1,t_1,u_1)\ a\ (s_2,t_1,u_1)\ b\ (s_2,t_2,u_1)\ b\ (s_2,t_1,u_1)\ a\ (s_1,t_1,u_1)\ c\ (s_1,t_1,u_1)$$

which gives *abbac* as a word in $\mathscr{L}(\mathscr{B})$. More generally, $\mathscr{B}$ accepts all words where all occurrences of c are such that there are an even number of a's and an even number of b's in the prefix before the c occurrence. That is,

$$\mathscr{L}(\mathscr{B}) = \left\{ v_0 \dots v_n \in \{a,b,c\}^* \;\middle|\; \begin{array}{r} \forall i \,.\, v_i = c \text{ implies } a_\sharp(v_0 \dots v_i) =_{mod 2} 0 \\ \text{and } b_\sharp(v_0 \dots v_i) =_{mod 2} 0 \end{array} \right\},$$

where $\sigma_\sharp(w)$ is the number of occurrences of letter σ in word w.

2.1.3 Local Asynchronous Automata

We also define a weaker version of asynchronous automata, called *local* asynchronous automata (short: LAA or local AA), in which the transition function is local to each process, and therefore independent with respect to the states of all other processes. To avoid confusion, we sometimes refer to normal asynchronous automata as defined earlier as *global* asynchronous automata (or global AA), though by default AA refers to global AA.

A *local asynchronous automaton* over (Σ, dom) and $\mathbb{P}$ is a tuple

$$\mathscr{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_p)_{p\in\mathbb{P}}),$$

where S_p and s_p^0 are defined as before, and $\delta_p : S_p \times dom^{-1}(p) \to S_p$ is the transition function of process p. A run of $\mathscr{B}$ is a sequence $\mathbf{s}_0 a_1 \mathbf{s}_1 a_2 \dots \mathbf{s}_n$ where $\mathbf{s}_0 = (s_p^0)_{p\in\mathbb{P}}$ and for all $0 < i \le n$, $\mathbf{s}_i = (s_i^p)_{p\in\mathbb{P}} \in \prod_{p\in\mathbb{P}} S_p$, $a_i \in \Sigma$, satisfying the following relation:

$$s_i^p = \begin{cases} \delta_p(s_{i-1}^p, a_i) & \text{if } p \in dom(a_i), \\ s_{i-1}^p & \text{otherwise.} \end{cases}$$

Observe that a local AA is a syntactic restriction of global AA. There are languages recognizable by global AA that cannot be recognized by local AA, because intuitively it would be impossible to make a process react differently to the same communication based on differences observed by another process. For example, take $\Sigma = \{a,\bar{a},b,c,\bar{c}\}$ and two processes p,q such that p listens to $a,\bar{a},b$ and q listens to

$c, \bar{c}, b$. Then take language $L = \{abc, \bar{a}b\bar{c}\}$. One can easily see that L can be recognized by a global AA but by no local AA.

In particular, Zielonka's distribution result [9] no longer holds for local AA. Note that the automaton given in Figure 1 is local.

2.2 Reconfigurable Communication

Let us consider here a model where the communication structure is not fixed, and can be modified dynamically during a run. As before, let us fix a finite set $\mathbb{P}$ of processes. Let us as well fix a finite set C of channels, with a role similar to the alphabet Σ of the previous section. Here, the function *dom* is replaced by a state-dependent listening function through which processes reconfigure their communication interfaces depending on their current state. Finally, let T be a finite set of message contents. The intuition behind T is to abstract the state-sharing part of a communication to allow us to define each process' transition function independently of other processes. We emphasize that this has nothing to do with reconfigurability and is just a way to have nicer definitions.

2.2.1 Reconfigurable Asynchronous Automata

A *reconfigurable asynchronous automaton* (in short: RAA) over C is a tuple $\mathscr{A} = (S, s^0, \Delta, L)$ where:

- S is a set of states, $s^0 \in S$ being the initial state,
- $\Delta : S \times (T \times C) \to S$ is the partial transition function, where $\Delta(s,(t,c)) = s'$ means going from state s to s' after having a message on channel c with content t. We write $(s,(t,c),s') \in \Delta$ for $\Delta(s,(t,c)) = s'$,
- $L : S \to 2^C$ is a listening function such that $c \in L(s)$ if there is a transition of the form $(s,(t,c),s') \in \Delta$, i.e. state s must be listening to channel c if there is some transition from s involving a message on c.

A run of $\mathscr{A}$ is a sequence $s_0 m_1 s_1 m_2 \dots s_n$ starting from the initial state $s_0 = s^0$ and where for all $0 < i \leq n, m_i \in T \times C$ and $\Delta(s_{i-1}, m_i) = s_i$. The language of $\mathscr{A}$, denoted by $\mathscr{L}(\mathscr{A})$, is the set of words over C of the form $c_0 c_1 \dots$ such that there exists a run of the form $s_0(t_0,c_0)s_1(t_1,c_1)\dots$, i.e. we focus only on the sequence of channels where messages are sent, and drop the states and message contents.

Intuitively this definition represents the behavior of a single process, communicating with the outside on channels from C. In order to be able to reconstruct the whole system, we now define the parallel composition of RAA.

Given a sequence of RAA $(\mathscr{A}_p)_{p\in\mathbb{P}}$ with $\mathscr{A}_p = (S_p, s_p^0, \Delta_p, L_p)$, one can define their parallel composition $\mathscr{A}_{\|\mathbb{P}} = (S, s^0, \Delta, L)$:

- $S = \prod_{p\in\mathbb{P}} S_p$ and $s_0 = (s_p^0)_{p\in\mathbb{P}}$,
- $L((s_p)_{p\in\mathbb{P}}) = \bigcup_{p\in\mathbb{P}} L_p(s_p)$,
- $\Delta((s_p)_{p\in\mathbb{P}},(t,c)) = (s'_p)_{p\in\mathbb{P}}$ if the following conditions are met:
  1. $\exists p$ s.t. $c \in L_p(s_p)$,
  2. $\forall p$ s.t. $c \in L_p(s_p), (s_p,(t,c),s'_p) \in \Delta_p$, and
  3. $\forall p$ s.t. $c \notin L_p(s_p), s'_p = s_p$.

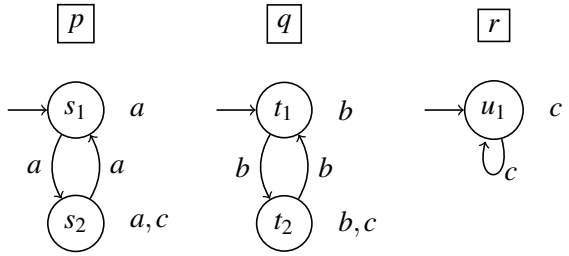

Figure 2: An RAA $\mathscr{A}$ over three processes. The listening function is given to the right of each state.

In plain words, there is a transition if all processes listening to the corresponding channel have a transition with the *same* message content, with at least one process listening to the channel, whereas those that do not listen are left unchanged. Note that if some process listens to that channel but does not implement the transition, then that transition is blocked.

By convention, an RAA over C and $\mathbb{P}$ refers to an RAA of the form $\mathscr{A}_{\|\mathbb{P}}$ as described above.

Example 2. Figure 2 shows an example of RAA over channels $C = \{a,b,c\}$ and three processes $\mathbb{P} = \{p,q,r\}$. Here we take $T = \{t\}$ as the set of message contents, so for readability purposes it is omitted from the transitions. Note that when process p is in state s_2, it is listening to channel c but no c-transition is implemented, therefore a communication on c is impossible (similarly for q and t_2). So the only way a communication happens on c is when p and q are in s_1 and t_1 respectively, which means only process r listens to c. It is then easy to see that this RAA accepts the same language as the AA given in Figure 1. Note that it does so without p or q ever taking part in a communication on c, contrary to the previous example.

3 From Fixed to Reconfigurable and Back

We now focus on comparing the expressive power of these two formalisms. For the rest of this section, we fix a finite set $\mathbb{P}$ of processes.

3.1 Fixed AA to Reconfigurable AA

Let (Σ, dom) be a distributed alphabet, and let $\mathscr{B}$ be an AA over it. One can construct an RAA $\mathscr{A}_{\|\mathbb{P}}$ with Σ as set of channels that recognizes the same language as $\mathscr{B}$.

The intuition is as follows. The listening function of each process is the same for all states: each process always listens to the channels that have this process in their domain. The only part that is not straightforward to emulate is that a transition of an AA depends on the states of all processes in the domain of the corresponding letter. Therefore each process in the RAA needs to share their states via message contents to all others when emulating a transition.

Theorem 1. *Every language recognized by an AA over (Σ, dom) and $\mathbb{P}$ can be recognized by an RAA with set of channels Σ and processes $\mathbb{P}$.*

Proof. Let $\mathscr{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$ be an AA as described earlier. For the set of messages, we take $T = \bigcup_{a\in\Sigma}(\prod_{p\in dom(a)} S_p)$.

Then let $\mathscr{A}_p = (S_p, s_p^0, \Delta_p, L_p)$ be a RAA for process p where:

- $L_p(s) = \{a \in \Sigma \mid p \in dom(a)\}$ for all $s \in S_p$,
- $\Delta_p(s_p,(t,a)) = (\delta_a(t))\downarrow_{\{p\}}$ if $s_p = t\downarrow_{\{p\}}$

i.e. an a-transition is possible if and only if the message t is the tuple comprising the current states of all processes in $dom(a)$, and all processes then update their state according to δ_a.

By construction, one can show inductively that for each run of $\mathscr{B}$, there is a corresponding run of $\mathscr{A}_{\|\mathbb{P}}$ where at each point, the state of each process p is the same in both runs. It follows that $\mathscr{L}(\mathscr{B}) \subseteq \mathscr{L}(\mathscr{A}_{\|\mathbb{P}})$ and conversely $\mathscr{A}_{\|\mathbb{P}}$ can only emulate runs of $\mathscr{B}$, showing the reverse inclusion. □

Note that the size of the constructed RAA lies almost entirely in the size of T, the message contents set, which is $\prod_{p\in\mathbb{P}} S_p$.

For local AA the translation is even more straightforward, as no message content is required (i.e. T can be reduced to a singleton).

Corollary 2. *Every language recognized by an LAA over (Σ, dom) can be recognized by an RAA with set of channels Σ and where $|T| = 1$.*

Proof. In the case of LAA, the transition δ_p does not depend on the states of other processes. Let $T = \{t\}$. We replace the transition Δ_p in the proof of Theorem 1 by $\Delta_p(s_p,(t,a)) = \delta_p(s_p,a)$. □

3.2 Reconfigurable to Fixed

Let us now focus on the reverse direction. Let $(\mathscr{A}_p)_{p\in\mathbb{P}}$ be a sequence of RAA over $\mathbb{P}$ with set of channels C, and let $\mathscr{A}$ be their parallel composition. Our goal is to create an AA with alphabet C that recognizes the same language. The question that arises is: what should *dom* be defined as for the distributed alphabet (C, dom)?

The solution is to define it as the complete domain function *Fdom*: $Fdom(a) = \mathbb{P}$ for all channels. In that case, it is simple to build an AA over $(C, Fdom)$ that emulates $\mathscr{A}$, as each process can simply stutter when they are not supposed to listen to a channel.

Theorem 3. *Every language recognized by an RAA over set of channels C and processes $\mathbb{P}$ can be recognized by an AA over $(C, Fdom)$ and the same set of processes.*

Proof. Consider $(\mathscr{A}_p)_{p\in\mathbb{P}}$, where $\mathscr{A}_p = (S_p, s_p^0, \Delta_p, L_p)$, with $\mathscr{A} = (S, s^0, \Delta, L)$ being their parallel composition over message contents T. We build $\mathscr{B} = ((Q_p)_{p\in\mathbb{P}}, (q_p^0)_{p\in\mathbb{P}}, (\delta_c)_{c\in C})$ as follows:

- for all $p \in \mathbb{P}$, $Q_p = S_p$, and $q_p^0 = s_p^0$
- For channel $c \in C$ we have δ_c defined as follows.

$$\delta_c = \left\{ ((q_p)_{p\in\mathbb{P}}, (q'_p)_{p\in\mathbb{P}}) \;\middle|\; \begin{array}{l} \exists p \in \mathbb{P}. c \in L_p(q_p) \text{ and} \\ \exists t \in T. \forall p \in \mathbb{P}. \\ \quad \text{if } c \in L_p(q_p), (q_{p'}, (t,c), q'_{p'}) \in \Delta_{p'} \text{ and} \\ \quad \text{if } c \notin L_p(q_p), q_p = q'_p \end{array} \right\}$$

Similarly to Theorem 1, the construction makes it so that any run from $\mathscr{A}$ has a corresponding run in $\mathscr{B}$ where the state are identical for each process, and the same in the other direction. □

Note that having global transitions is necessary to ensure all processes share the same message content t. However if we assume that T is a singleton, then local transitions suffice. Additionally, notice that the construction would still work with a set T of infinite size, so we could consider RAA where processes synchronize by agreeing on, say, an integer.

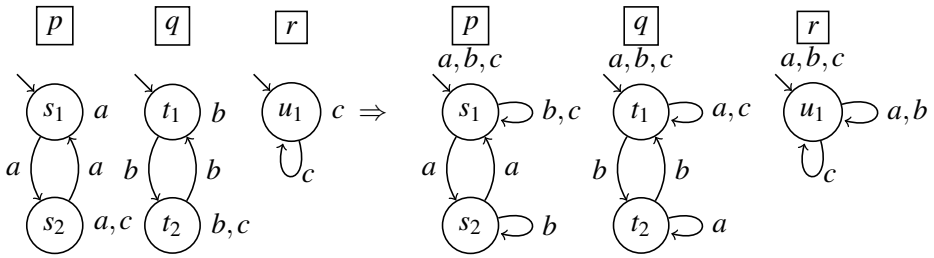

Figure 3: On the left, the RAA from Example 2. On the right, its translation to an AA.

Corollary 4. *Every language recognized by an RAA over C and $\mathbb{P}$, where $|T| = 1$, can be recognized by an LAA over $(C, Fdom)$ and $\mathbb{P}$.*

We illustrate this construction in Figure 3. Note that for this particular example the general construction is not optimal. For example, process p is made to listen to b but can never block a communication on this channel with all states having a self-loop on reading b. Thus, one could safely remove the letter b from the alphabet of p. By doing similarly on other processes, one can get back the AA from Example 1.

There is an alternative construction that does not require *all* processes to listen to *all* channels. If one process does while also storing the state information of every other processes, then it can simulate the original automaton by itself; meanwhile every other process can listen to an arbitrary set of channels as long as they accept every communication. In other words, one process serves as a centralized executor of the simulation, while others simply need to be non-blocking. With a centralized executor there is no point in having computation done anywhere but in the centralized executor. By abuse of definition we still refer to such a domain function as a complete domain.

In the next section we show that there is no hope of finding a transformation that does not require a complete domain.

4 Trivializable, Fully Listening, and Trivial

The method described above is a general method to transform a reconfigurable asynchronous automaton into an equivalent fixed asynchronous automaton, with the cost of needing a complete domain function. It is of course possible that for some particular examples such a heavy construction is not needed, and a translation with a much smaller domain could be possible. However, we show that there is no better (in terms of channel domain) *general* translation by giving an example of an RAA such that every equivalent AA relies on a complete domain.

The idea is to allow every possible subset of channels to be either fully independent, that is every one of those channels can be used in parallel, or make them sequentially dependent, that is they can only be used in a certain order. This status can be switched by a communication on a separate channel (that all processes listen to), called the switching channel. Moreover, after enough switches, a different channel will serve as the switching channel. That way, all channels have the opportunity to serve as the switching channel, given enough switches. Our construction does not use data values during communications. Thus, already the weakest form of RAA is enough for this example.

4.1 Description of the switching RAA

Let $\mathbb{P} = \{p_1, \ldots, p_n\}$. We fix $C = \{c_1, \ldots, c_n, c_{n+1}\}$, that is, we have one channel per process and one additional channel to be used as switching channel (dynamically).

For all $\mathrm{sc} \in C$ (sc stands for *switching channel*), fix $<_{\mathrm{sc}}$ an arbitrary total order over $2^{C\setminus\{\mathrm{sc}\}}$, with the only requirement that $\emptyset$ be the minimal element. Intuitively, a set in $2^{C\setminus\{\mathrm{sc}\}}$ will represent the set of dependent channels, and a switch will go to the next one with respect to $<_{\mathrm{sc}}$. Let us denote by $inc_{<_{\mathrm{sc}}} : 2^{C\setminus\{\mathrm{sc}\}} \to 2^{C\setminus\{\mathrm{sc}\}} \cup \{\bot\}$ the function that returns the next set according to $<_{\mathrm{sc}}$ or $\bot$ for the maximal element.

Additionally, for every subset $D \subseteq C$, we fix $+^D 1 : C \to C$ a function that cycles through all elements of D and is the identity on $C \setminus D$. For convenience we write $d +^D 1$ for $+^D 1(d)$. We also define $-^D 1 : D \to D$ the inverse function and use the same notation. Namely, for every $d \in D$ we have $(d -^D 1) +^D 1 = d$ and $(d +^D 1) -^D 1 = d$. We denote by $c_D \in D$ an arbitrary element of D.

Finally, we set $T = \{t\}$, and omit the message content component in transitions.

We build $\mathscr{A}_p = (S_p, s_p^0, \Delta_p, L_p)$ for $p = p_k$ as follows:

- $S_p = \{(c, \mathrm{sc}, D, d) \mid c, \mathrm{sc} \in C, D \subseteq C \setminus \{\mathrm{sc}\}, d \in D \cup \{c\}\}$, and $s_p^0 = (c_k, c_{n+1}, \emptyset, c_k)$.
  The first component is the channel assigned to this process, initially c_k for process p_k, but may change if c_k becomes the switching channel. The second component is the current switching channel, initialized to c_{n+1} for all processes. Component D represents the set of channels that are currently dependent, and d is the next channel that $\mathscr{A}_k$ is listening to on which it is expecting communication.
- All processes listen to the switching channel and their assigned channel, plus the previous one if D contains the assigned channel:

$$L_p(c, \mathrm{sc}, D, d) = \begin{cases} \{\mathrm{sc}, c, c -^D 1\} & \text{if } c \in D \\ \{\mathrm{sc}, c\} & \text{if } c \notin D \end{cases}$$

- The transition Δ_p is the union of the following sets .

$$\{((c, \mathrm{sc}, D, c), c, (c, \mathrm{sc}, D, c -^D 1))\} \tag{1}$$
$$\{((c, \mathrm{sc}, D, c -^D 1), c -^D 1, (c, \mathrm{sc}, D, c))\} \tag{2}$$

  The first two kinds of transitions handle the independence of all channels in $C \setminus D$ and the cycling through the channels of D. If $c \notin D$ then $c = c -^D 1$. In this case, the first two sets simply say that a transition on c is always possible. If $c \in D$, then the process awaits until it gets a message on $c -^D 1$ and then is ready to interact on c. After interaction on c it awaits another interaction on $c -^D 1$. It follows that all the processes owning the channels in D enforce together the cyclic order on the messages in D. This part is further illustrated in Figure 4.

  Remaining transitions describe what happens when a switch occurs.

$$\{((c, \mathrm{sc}, D, d), \mathrm{sc}, (c, \mathrm{sc}, D', c)) \mid D' = inc_{<_{\mathrm{sc}}}(D) \neq \bot \text{ and } c = c_{D'}\} \tag{3}$$
$$\{((c, \mathrm{sc}, D, d), \mathrm{sc}, (c, \mathrm{sc}, D', c -^{D'} 1)) \mid D' = inc_{<_{\mathrm{sc}}}(D) \neq \bot \text{ and } c \neq c_{D'}\} \tag{4}$$

  Sets three and four describe what happens when the next set according to $<_{\mathrm{sc}}$ is defined. In this case, the next set becomes the new set of dependent channels D. Set three handles the case of the process that is in charge of the channel becoming the first channel to communicate on the new set

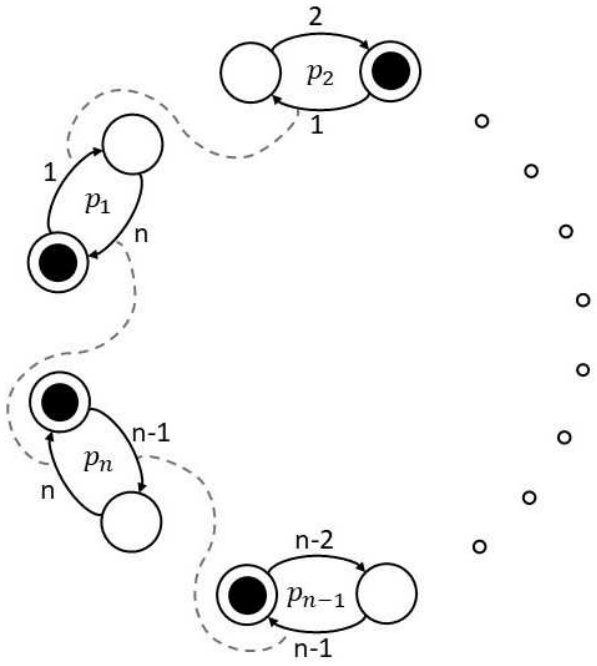

Figure 4: Illustration of how the order on the channels in D is maintained. We consider the case where $D = \{1,\dots,n\}$ and p_i is in charge of channel i. The order between the channels is the natural order on $\{1,\dots,n\}$. The black token indicates the current state for each process. Transitions that are on the same channel are connected with a dashed line. The system is set up for next communication on channel 1 and all other channels are blocked. Indeed, both processess listening to channel 1 are ready to interact on 1 (p_1 in state $(1,n+1,D,1)$ and p_2 in state $(2,n+1,D,1)$) and for every other channel $i > 2$ process i is awaiting communication on $i-1$ (p_i in state $(i,n+1,D,i-1)$) so channel i is not enabled.

$inc_{<_{\text{sc}}}(D)$. This process is ready for communication on this first channel. The fourth set handles the case of all other processes. All other processes are either in charge of channels in D', in which case they set themselves to await a communication on the previous in D' or they are in charge of channels not in D' in which case, c and $c \dashv^{D'} 1 = c$, and the process is ready to communicate on c.

$$\{((c,\text{sc},D,d),\text{sc},(c,\text{sc}\dashv^{C}1,\emptyset,c)) \mid inc_{<_{\text{sc}}}(D) = \bot \text{ and } c \neq \text{sc}\dashv^{C}1\} \tag{5}$$

$$\{((c,\text{sc},D,d),\text{sc},(\text{sc},\text{sc}\dashv^{C}1,\emptyset,\text{sc})) \mid inc_{<_{\text{sc}}}(D) = \bot \text{ and } c = \text{sc}\dashv^{C}1\} \tag{6}$$

Finally, sets five and six describe what happens when the next set according to $<_{\text{sc}}$ is undefined. In this case, the next dependent set becomes $\emptyset$. Most processes just set the dependent set to $\emptyset$ and allow communication on "their" channel (set 5). The process that was in charge of the new switching channel $\text{sc}\dashv^{C}1$ takes over the old switching channel sc and is ready to communicate on it (set 6). Notice that communications on the switching channel affect all processes. The change in D and the change of the switching channel is further illustrated in Figure 5.

An illustration of the whole construction for $n = 2$ (i.e. 2 processes and 3 channels) is given in Figure 6. There we have processes $\mathbb{P} = \{p,q\}$ and channels $C = \{1,2,3\}$. Initially p is assigned channel 1 and q channel 2, while channel 3 is the switching channel. We chose as order for the dependent sets the following order: $\emptyset <_3 \{1\} <_3 \{2\} <_3 \{1,2\}$. The blue states illustrate the moment when the dependent

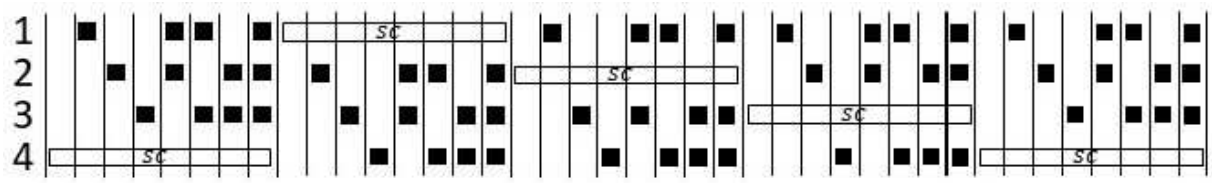

Figure 5: Illustration of how the set D and the switching channel sc change whenever there is a communication on sc. We consider the case where there are three processes and four channels. Each column corresponds to the status after one more communication on sc. Each channel is in turn the switching channel starting with 4. The channels in D at a certain time/column are marked with a black box. We cycle through the states in $\{1,\ldots,4\}\setminus\{\text{sc}\}$ according to set size first and then lexigographically on the sorted set.

set has two channels that must be used in the correct order ($1 \to 2 \to 1 \to ...$). The red transitions lead to a change in the switching channel. At that point, 1 becomes the new switching channel. Thus p gets a new assigned channel (3, i.e. the previous switching channel), while q keeps its old assigned channel. After enough changes of the switching channel, the state cycles back to the initial state for both.

Let $\mathscr{A}$ be the parallel composition of $(\mathscr{A}_p)$. A state for one process keeps track of 3 channels and one set of channels, so its size is in $O(n^3.2^n)$. Therefore, the size of the state space of $\mathscr{A}$ is in $O((n^3.2^n)^n)$.

4.2 Asynchronous Automata Construction

We show that an AA that recognizes the same language as $\mathscr{A}$ has the following property: for each process p, either p listens to every channel ($dom^{-1}(p) = C$), or from every reachable state there is a path to a bottom strongly connected component that is complete w.r.t. $dom^{-1}(p)$. That is, for every state s in this bottom SCC and for every channel in $dom^{-1}(p)$ the transition $\delta(s,c)$ is defined. In the former case, we call p *fully-listening*. In the latter case, we say that p is *trivializable*, as once it is in this bottom SCC it always includes transitions for all the channels it listens to. Thus, p becomes irrelevant to the rest of the computation.

Theorem 5. *Let $\mathscr{B}$ be an AA such that $\mathscr{L}(\mathscr{B}) = \mathscr{L}(\mathscr{A})$. Each process in $\mathscr{B}$ is either fully-listening or trivializable.*

Proof. Let $\mathscr{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$, and let $p \in \mathbb{P}$. Assume that p is not fully-listening, so let $C_p = dom^{-1}(p) \subsetneq C$. In particular, let $c \in C \setminus C_p$ be a channel that p does not listen to.

Let s_p be a reachable state for p in $\mathscr{B}$. Then there is w a computation in $\mathscr{L}(\mathscr{B})$ such that p reaches state s_p after w. Consider the same computation in $\mathscr{A}$, and let sc be the current switching channel in $\mathscr{A}$ at the end of w. Let $\text{sc} \cdot c_1 \cdot ... \cdot c_{n-1} \cdot c$ be the sequence of channels from sc to c according to the order from $\lightning$1. Then there is a continuation w' of w of the form $\text{sc}^{k_0} \cdot c_1^{k_1} \cdot ... \cdot c_{n-1}^{k_{n-1}}$ with $k_0,\ldots,k_{n-1} \in \mathbb{N}$ such that:

- $ww' \in \mathscr{L}(\mathscr{A})$,
- after w', c is the current switching channel and the dependent set D is $\emptyset$.

From this, every continuation $w'' \in (C\setminus\{c\})^*$ is still in $\mathscr{L}(\mathscr{A})$ and does not change the switching channel or the dependent set. In particular, every $w'' \in (dom^{-1}(p))^*$ maintains that $w \cdot w' \cdot w''$ is also in $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{B})$. Therefore, from the state reached in $\mathscr{B}$ after ww', there is a path to a strongly connected component that will implement all transitions in $dom^{-1}(p)$, i.e. a complete one. □

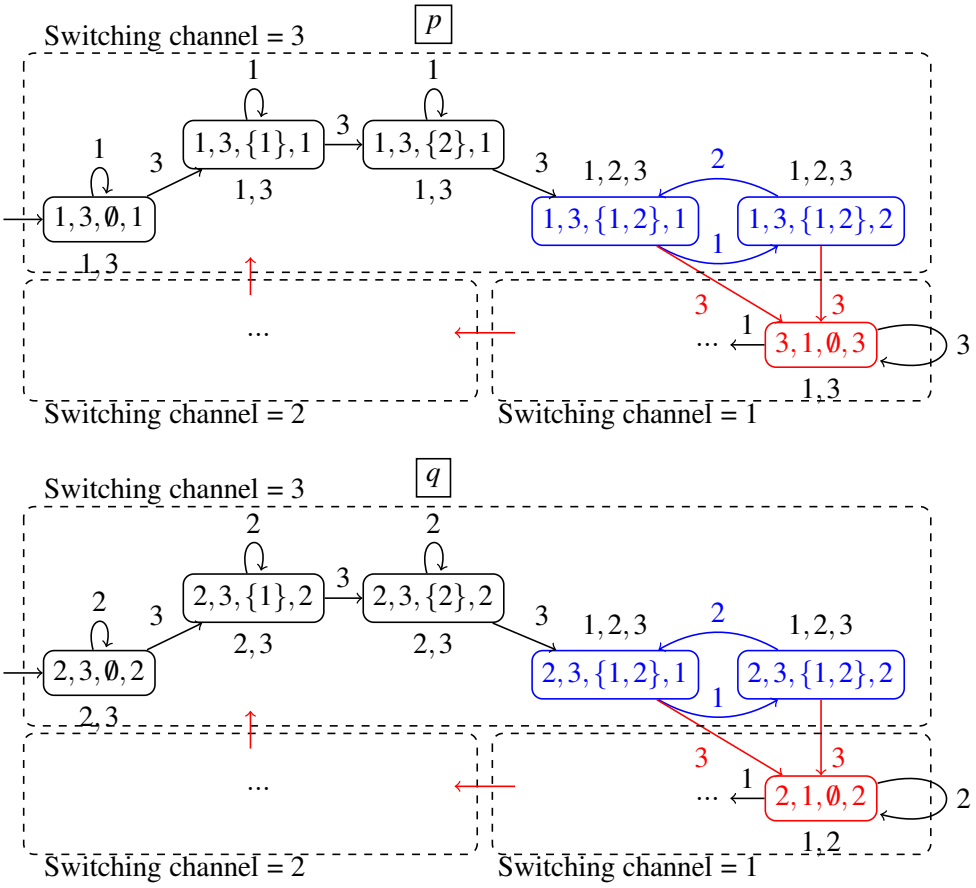

Figure 6: Illustration of the switching RAA for $n = 2$.

A process that is trivializable may become irrelevant. This means that there are pathological runs where only fully listening processes are active in the computation while others passively accept everything. However, trivializable processes *may* still initially participate in the computation. Nevertheless, for the languages given in this section, we show that there exists an alternative initial configuration of the system where trivializable processes actually start trivialized. This means that, in essence, all the machinery required for doing the entire computation is present within the remaining fully listening processes only.

Given a language $\mathscr{L}$ and a word w let $w\backslash\mathscr{L} = \{w' \mid ww' \in \mathscr{L}\}$ and let $pref(\mathscr{L}) = \{w \mid \exists w' \,.\, ww' \in \mathscr{L}\}$. A language $\mathscr{L}$ is *repetitive* if for every word $w \in pref(\mathscr{L})$ there exists a word w' such that $ww'\backslash\mathscr{L} = \mathscr{L}$.

Lemma 6. *The language $\mathscr{L}(\mathscr{A})$ is repetitive.*

Proof. Consider a word w and the configuration of $\mathscr{A}$ reachable after reading w. All processes in $\mathscr{A}$ agree on the set D and the channel sc. Every communication on sc increases the set of dependent channels in the order $<_{\text{sc}}$ until reaching the set D' such that $inc_{<_{\text{sc}}}(D') = \bot$. An additional communication on sc then leads to the switching channel being updated to $\text{sc}\dot{+}1$.

So after at most 2^n communications on sc the switching channel becomes $\text{sc}\dot{+}1$. Let w_0 be the word that leads to the switching channel changing.

For every channel, c_i, when the dependent set is $\emptyset$ the sequence $(c_i)^{2^n}$ leads to the change of the switching channel from c_i to $c_i\dot{+}1$.

Let $\text{sc} = c^0, c^1, \ldots, c^k$ be the sequence of switching channels ending $c^k\dot{+}1 = c_{n+1}$.

It follows that $w_0 \cdot (c^1)^{2^n} \cdots (c^k)^{2^n}$ leads $\mathscr{A}$ to setting the switching channel to c_{n+1}. At that point all processes in $\mathscr{A}$ are in their initial states except for their assigned channel, which is shifted by one. Namely, process p_k ends up in state $(c_k\dot{-}^{(C\setminus\{c_{n-1}\})}1, c_{n+1}, \emptyset, c_k)$.

Now let $w_{\text{loop}} = c_{n+1}^{2^n} \cdot c_1^{2^n} \cdots c_n^{2^n}$. Each application of w_{loop} again shifts assigned channels by one. So after $n-1$ applications, each process finishes in its initial state. From this configuration the residue language is $\mathscr{L}(\mathscr{A})$. □

Using repetitiveness we can strengthen our result as follows. A process is *trivial* if its initial state lies in a bottom strongly connected component that is complete w.r.t. $dom^{-1}(p)$. Given an AA $\mathscr{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$ and an alternative initial configuration $\vec{t} = (t_p^0)_{p\in\mathbb{P}}$ we denote by $\mathscr{B}(\vec{t})$ the AA $\mathscr{B}(\vec{t}) = ((S_p)_{p\in\mathbb{P}}, (t_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$.

Theorem 7. *Let $\mathscr{B}$ be an AA such that $\mathscr{L}(\mathscr{B}) = \mathscr{L}(\mathscr{A})$. There exists an alternative initial configuration $\vec{t} = (t_0)_{p\in\mathbb{P}}$ such that $\mathscr{L}(\mathscr{B}(\vec{t})) = \mathscr{L}(\mathscr{A})$ and each process in $\mathscr{B}(\vec{t})$ is either fully-listening or trivial.*

Proof. Let $\mathscr{B}$ be an AA equivalent to $\mathscr{A}$. By Theorem 5 there exists a word w such that after reading w all processes in $\mathscr{B}$ that are not fully listening reached a bottom SCC, where they accept all communications on all channels they are listening to. By Lemma 6, there exists a word w' such that $ww'\backslash\mathscr{L}(\mathscr{B}) = \mathscr{L}(\mathscr{B})$. Let $\vec{t} = (t_p^0)_{p\in\mathbb{P}}$ be the states that processes in $\mathscr{B}$ reach after reading ww'. Then $\mathscr{L}(\mathscr{B}(\vec{t})) = \mathscr{L}(\mathscr{B})$. The theorem follows. □

5 Conclusion and Discussion

We study the addition of reconfiguration of communication to asynchronous automata. We show that in terms of expressiveness, the addition does not change the power of the model: every language recognized distributively by automata with reconfigurable communication can be recognized essentially by

the same automata with fixed communication. For deterministic automata this also means that the two are bisimilar. The same is (obviously) true in the other direction. However, the cost of conversion is in disseminating widely all the information and leaving it up to the processes whether to use it or not. We also show that this total dissemination cannot be avoided. Processes who do not get access to the full information about the computation become irrelevant and in fact do not participate in the distributed computation.

The issues of mobile and reconfigurable communication raise a question regarding "how much" communication is performed in a computation. Given a language recognized by an asynchronous automaton (distributively), the independence relation between letters is fixed by the language. It follows that two distributed systems in the form of asynchronous automata accepting (distributively) the same language must have the same independence relation between letters. However, this does not mean that they agree on the distribution of the alphabet. In case of two different distributed alphabets, what makes one better than the other? This question becomes even more important with systems with reconfigurable communication interfaces. Particularly, in reconfigurable asynchronous automata, the connectivity changes from state to state, which makes comparison even harder. How does one measure (and later reduce or minimize) the amount of communication in a system while maintaining the same behavior? We note that for the system in Section 4, the maximal number of channels a process is connected to is four regardless of how many channels are in the system. Dually, the asynchronous automaton for the same language requires every process that participates meaningfully in the interaction to have number of connections equivalent to the parameter n. Is less connectivity better than more connectivity?

The issues of "who is connected" and "with whom information is shared" also have implications for security and privacy. Reconfiguration allowed us to share communication only with those who "need to know". Fixed topology forced us to disseminate information widely. If we intend to use language models and models of concurrency in applications that involve security and privacy we need a way to reason about dissemination of information and comparing formalisms also based on knowledge and information.

Acknowledgments
We are grateful to Y. Abd Alrahman and L. Di Stefano for fruitful discussions and suggestions.

References

[1] Yehia Abd Alrahman, Rocco De Nicola & Michele Loreti (2019): *A calculus for collective-adaptive systems and its behavioural theory*. Information and Computation 268, p. 104457, doi:10.1016/j.ic.2019.104457.

[2] Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi & Roberto Vigo (2015): *A calculus for attribute-based communication*. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 1840–1845, doi:10.1145/2695664.2695668.

[3] Yehia Abd Alrahman & Nir Piterman (2021): *Modelling and verification of reconfigurable multi-agent systems*. Auton. Agents Multi Agent Syst. 35(2), p. 47, doi:10.1007/s10458-021-09521-x.

[4] Blaise Genest, Hugo Gimbert, Anca Muscholl & Igor Walukiewicz (2010): *Optimal Zielonka-type construction of deterministic asynchronous automata*. In: Automata, Languages and Programming: 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II 37, Springer, pp. 52–63, doi:10.1007/978-3-642-14162-1_5.

[5] Blaise Genest & Anca Muscholl (2006): *Constructing exponential-size deterministic Zielonka automata*. In: Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II 33, Springer, pp. 565–576, doi:10.1007/11787006_48.

[6] Siddharth Krishna & Anca Muscholl (2013): *A quadratic construction for Zielonka automata with acyclic communication structure*. Theoretical Computer Science 503, pp. 109–114, doi:10.1016/j.tcs.2013.07.015.

[7] Madhavan Mukund, K Narayan Kumar & Milind Sohoni (2000): *Synthesizing distributed finite-state systems from MSCs*. In: International Conference on Concurrency Theory, Springer, pp. 521–535, doi:10.1007/3-540-44618-4_37.

[8] Madhavan Mukund & Milind Sohoni (1997): *Keeping track of the latest gossip in a distributed system*. Distributed Computing 10, pp. 137–148, doi:10.1007/s004460050031.

[9] Wieslaw Zielonka (1987): *Notes on Finite Asynchronous Automata*. RAIRO Theor. Informatics Appl. 21(2), pp. 99–135, doi:10.1051/ita/1987210200991.