Logical Methods in Computer Science
Volume 22, Issue 4, 2026, pp. 9:1–9:27
https://lmcs.episciences.org/

Submitted Apr. 08, 2025
Published Oct. 27, 2026

ADDING RECONFIGURATION TO ZIELONKA'S ASYNCHRONOUS AUTOMATA

MATHIEU LEHAUT AND NIR PITERMAN

University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden

Abstract. 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.

This paper is part of the special issue *Selected Papers of the 15th International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2024).*

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. Those channels enable a symmetric, multiprocess kind of communication called synchronization. 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

Key words and phrases: asynchronous automata, reconfiguration, distribution.

An earlier version appeared in Gandalf 2024.

Supported by the ERC Consolidator grant D-SynMA (No. 772459), by the Swedish research council (VR) project (No. 2020-04963) and by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. M. Lehaut was also partially supported by NCN grant 2021/41/B/ST6/00535.

DOI:10.46298/LMCS-22(4:9)2026

© M. Lehaut and N. Piterman
Creative Commons

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 [Zie87]. 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 saying that independent communications can commute, 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. [MS97, GM06, GGMW10] for the general construction and [KM13] for an example of a simpler construction in a restricted case. This result leads to several applications notably in synthesis [MKS00], 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 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 corresponds to the state sharing of asynchronous automata. We call this variant *reconfigurable asynchronous automata*. They are inspired by attribute-based communication calculus [ADNL$^+$15, AADNL19] and channeled transition systems [AP21], 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". Third, 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. Fourth, in several cases, algorithms applied to asynchronous automata rely (or use) sparse communication structure for efficiency or simplicity [KM13, MW14]. In one case, we have shown that efficiency is preserved under "sparse" reconfiguration [HLP24]. This may be lost if we saturate communication to simulate reconfiguration.

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 (AA) and give the definition of reconfigurable asynchronous automata (RAA). 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. Section 5 introduces a discussion on the cost of communication and gives an alternative construction. Section 6 discusses some other approaches to reconfiguraiton. Finally, we conclude and discuss our results in Section 7.

The paper extends our earlier publication in GandALF 2024 [LP23]. We extend the results by introducing a measure for the cost of communications, analyzing the cost of communication in the different models, and introducing an alternative construction of an RAA that challenges AA, and showing that it changes the balance in the trade-off between passive and active communication.

2. Definitions

In this section we introduce the two models of interest for this work, asynchronous automata and their reconfigurable variant.

2.1. **Fixed Communication Structure.**

We introduce the classic definition of asynchronous automata and give some examples. We distinguish between the *global* and *local* versions differing on the amount of data transferred when communication occurs.

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 a binary independence 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 $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ be a deterministic automaton over Σ. We say that $\mathcal{A}$ 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 $\mathcal{A}$ has this property, then a word u is accepted by $\mathcal{A}$ 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 [Zie87].

2.1.2. *Asynchronous Automata.* An *asynchronous automaton* (in short: AA) [Zie87] over distributed alphabet (Σ, dom) and processes $\mathbb{P}$ is a tuple

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

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 global states, as defined below.

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 $\mathcal{B}$ 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 $\mathcal{B}$ if such an accepting run exists (note that automata are deterministic but runs on certain words may not exist). The language of $\mathcal{B}$, denoted by $\mathcal{L}(\mathcal{B})$, is the set of words accepted by $\mathcal{B}$. 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.

Example 2.1. We give an example of an asynchronous automaton $\mathcal{B}$ 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 $\mathcal{L}(\mathcal{B})$. More generally, $\mathcal{B}$ 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,

$$\mathcal{L}(\mathcal{B}) = \left\{ v_0 \dots v_n \in \{a,b,c\}^* \;\middle|\; \begin{array}{r} \forall i \,.\, v_i = c \text{ implies } a_\sharp(v_0 \dots v_i) =_{mod2} 0 \\ \text{and } b_\sharp(v_0 \dots v_i) =_{mod2} 0 \end{array} \right\},$$

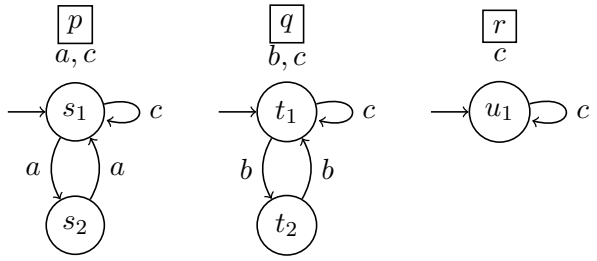

Figure 1: An asynchronous automaton $\mathcal{B}$ over three processes p, q, and r. Below each process is the list of letters that contain this process in their domain, and below that is the automaton associated with the process.

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

$$\mathcal{B} = ((S_p)_{p\in\mathbb{P}}, (s^0_p)_{p\in\mathbb{P}}, (\delta_p)_{p\in\mathbb{P}}),$$

where S_p and s^0_p 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 $\mathcal{B}$ is a sequence $\mathbf{s}_0 a_1 \mathbf{s}_1 a_2 \dots \mathbf{s}_n$ where $\mathbf{s}_0 = (s^0_p)_{p\in\mathbb{P}}$ and for all $0 < i \le n$, $\mathbf{s}_i = (s^p_i)_{p\in\mathbb{P}} \in \prod_{p\in\mathbb{P}} S_p$, $a_i \in \Sigma$, satisfying the following relation:

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

In other words, a run in a local AA is a run on the direct product of the automata of each process.

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. Similarly, we could modify the AA in Example 2.1 so that it accepts the language

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

We do this by replacing the transitions on c as follows. We have $\delta((s_1, t_1, u_1), c) = (s_1, t_1, u_1)$ and $\delta((s_2, t_2, u_1), c) = (s_2, t_2, u_2)$. Namely, allow c transitions only when both p and q agree on the parity of the number of as and bs that they have seen so far. In particular, Zielonka's

distribution result [Zie87] no longer holds for local AA. Note that the automaton given in Example 2.1 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. Namely, in AA, the alphabet and its distribution are given. This fixes a connection between processes and certain letters. Contrarily, here, processes come with some initial connectivity, but are aware of the existence of all letters / communication means. During execution, processes may decide to change the letters they react to, either by adding further connections or removing them.

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 using message contents to exchange data rather than including, for example, a definition of a transition that is parameterized with the processes that are connected to it, has nothing to do with reconfigurability and is just a way to simplify definitions. The reconfigurability comes only from the previously mentionned listening function.

2.2.1. *Reconfigurable Asynchronous Automata.* A *reconfigurable asynchronous automaton* (in short: RAA) over C is a tuple $\mathcal{A} = (S, s^0, \Delta, L)$ 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 $\mathcal{A}$ 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 $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, 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 $(\mathcal{A}_p)_{p\in\mathbb{P}}$ with $\mathcal{A}_p = (S_p, s^0_p, \Delta_p, L_p)$, one can define their parallel composition $\mathcal{A}_{\|\mathbb{P}} = (S, s^0, \Delta, L)$:

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

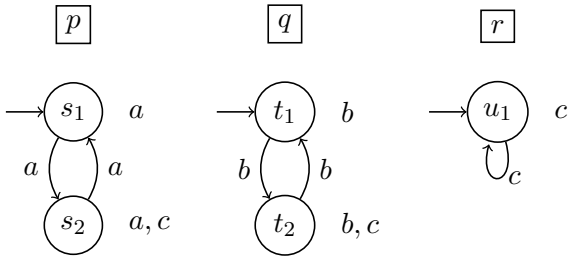

Figure 2: An RAA $\mathcal{A}$ over three processes. The listening function is given to the right of each state.

(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. $c \notin L_p(s_p), s'_p = s_p$.

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 $\mathcal{A}_{\|\mathbb{P}}$ as described above. We say that $\mathcal{A}_{\|\mathbb{P}}$ is *message-deterministic* if, for any global state $s \in S$ and channel $c \in C$, for all message contents $t, t' \in T$ such that $\Delta(s, (t, c))$ and $\Delta(s, (t', c))$ are defined, they both lead to the same successor state, i.e. $\Delta(s, (t, c)) = \Delta(s, (t', c))$. That is, from one global state of the system there can not be two different data values that lead to different states. This restriction becomes important when we consider translation from RAA to AA, as without message determinism, AA would have to support nondeterministic transitions. We believe that this is a reasonable restriction as the contents of the message jointly "arises" from the states of all processes. In fact, we would not expect the same global state to support multiple data values.

Example 2.2. 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. We show translations in both directions. The translation from fixed to reconfigurable asynchronous automata corresponds to a change of syntax. The translation from reconfigurable to fixed asynchronous automata introduces a dissemination of the data throughout the system. 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 $\mathcal{B}$ be an AA over it. One can construct an RAA $\mathcal{A}_{\|\mathbb{P}}$ with Σ as set of channels that recognizes the same language as $\mathcal{B}$.

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 simulate 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 state via message content to all others when simulating a transition.

Theorem 3.1. *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 $\mathcal{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$ 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 $\mathcal{A}_p = (S_p, s_p^0, \Delta_p, L_p)$ be an 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. Remark that this means $\mathcal{A}_{\|\mathbb{P}}$ is message-deterministic.

By construction, one can show inductively that for each run of $\mathcal{B}$, there is a corresponding run of $\mathcal{A}_{\|\mathbb{P}}$ where at each point, the state of each process p is the same in both runs. It follows that $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A}_{\|\mathbb{P}})$ and conversely $\mathcal{A}_{\|\mathbb{P}}$ can only simulate runs of $\mathcal{B}$, 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 3.2. *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 3.1 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 $(\mathcal{A}_p)_{p\in\mathbb{P}}$ be a sequence of RAA over $\mathbb{P}$ with set of channels C, and let $\mathcal{A}$ 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 simulates $\mathcal{A}$, as each process can simply stutter when they are not supposed to listen to a channel.

Theorem 3.3. *Every language recognized by a message-deterministic 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.*

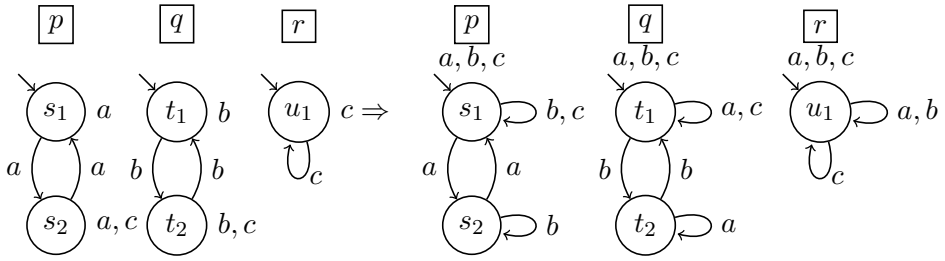

Figure 3: On the left, the RAA from Example 2.2. On the right, its translation to an AA.

Proof. Consider $(\mathcal{A}_p)_{p\in\mathbb{P}}$, where $\mathcal{A}_p = (S_p, s_p^0, \Delta_p, L_p)$, with $\mathcal{A} = (S, s^0, \Delta, L)$ being their parallel composition over message contents T. We build $\mathcal{B} = ((Q_p)_{p\in\mathbb{P}}, (q_p^0)_{p\in\mathbb{P}}, (\delta_c)_{c\in C})$ 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 3.1, the construction makes it so that any run of $\mathcal{A}$ has a corresponding run of $\mathcal{B}$ where the states are identical for each process, and the same in the other direction. □

Note that message-determinism is necessary for the transition function to be well defined. Without that assumption, we would obtain non-deterministic AA, which have not been defined in this paper. Furthermore, 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.

Corollary 3.4. *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 2.1.
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 message contents during communications. Thus, already the weakest form of RAA (T is a singleton) is enough for this example.

Technically, this is implemented using the following notions:

- At every given moment of time there is a set of channels on which communication is unrestricted and (the complement) set of channels on which communication happens only in a fixed (cyclic) order. Each process stores their current view of what is this set as part of their state space. We store the restricted set rather than (its complement) the unrestricted set. The coordination between all processes ensures that all processes have a common view of what is this set at all times.
- At every given moment of time, every process is in charge of one channel. This process "initiates" communication on this channel. If this channel needs to respect a given order, this process waits for the communication that enables this channel and only then initiates. Thus, the state space of each process memorizes the channel they are in charge of. They need to memorize whether they are awaiting communication on another channel in order to initiate communication on their channel. In order to do the latter, we add to the state space of a process the channel on which they await the next communication.
- At every given moment of time, one of the channels (switching channel) serves as a global coordination channel. All processes need to maintain the identity of this channel in their state space. A communication on this channel changes the set of ordered channels according to some agreed upon order between all possible sets. Once we have gone through all possible sets of ordered channels, the switching channel itself changes. The coordination between all processes ensures that all processes have a common view of which channel is the switching channel.

4.1. Description of the switching RAA.

Let $\mathbb{P} = \{p_1, \ldots, p_n\}$. We fix the set of channels $C = \{c_1, \ldots, c_n, c_{n+1}\}$, 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 $\mathcal{A}_p = (S_p, s_p^0, \Delta_p, L_p)$ 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 $\mathcal{A}_k$ is listening to on which it is expecting communication. Note that if D is the empty set or any singleton, it means there are no constraints on communication for every non-switching channel. Thus, the behavior of $\mathcal{A}_p$ is the same for all sets D such that $|D| \leq 1$.
- 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 six sets.

$$\{((c, \mathrm{sc}, D, c), c, (c, \mathrm{sc}, D, c-^D 1))\} \tag{4.1}$$

$$\{((c, \mathrm{sc}, D, c-^D 1), c-^D 1, (c, \mathrm{sc}, D, c))\} \tag{4.2}$$

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.

$$\{((c, \mathrm{sc}, D, d), \mathrm{sc}, (c, \mathrm{sc}, D', c)) \mid D' = inc_{<_{\mathrm{sc}}}(D) \neq \bot \text{ and } c = c_{D'}\} \tag{4.3}$$

$$\{((c, \mathrm{sc}, D, d), \mathrm{sc}, (c, \mathrm{sc}, D', c-^{D'} 1)) \mid D' = inc_{<_{\mathrm{sc}}}(D) \neq \bot \text{ and } c \neq c_{D'}\} \tag{4.4}$$

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 $inc_{<_{\mathrm{sc}}}(D)$. 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 $c \dot{-}^{D'} 1 = c$, and the process is ready to communicate on c.

$$\{((c, \mathrm{sc}, D, d), \mathrm{sc}, (c, \mathrm{sc}\dot{+}^{\mathcal{C}}1, \emptyset, c)) \mid inc_{<_{\mathrm{sc}}}(D) = \bot \text{ and } c \neq \mathrm{sc}\dot{+}^{\mathcal{C}}1\} \tag{4.5}$$

$$\{((c, \mathrm{sc}, D, d), \mathrm{sc}, (\mathrm{sc}, \mathrm{sc}\dot{+}^{\mathcal{C}}1, \emptyset, \mathrm{sc})) \mid inc_{<_{\mathrm{sc}}}(D) = \bot \text{ and } c = \mathrm{sc}\dot{+}^{\mathcal{C}}1\} \tag{4.6}$$

Finally, sets five and six describe what happens when the next set according to $<_{\mathrm{sc}}$ 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 (4.5). The process that was in charge of the new switching channel $\mathrm{sc}\dot{+}^{\mathcal{C}}1$ takes over the old switching channel sc and is ready to communicate on it (4.6). 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.

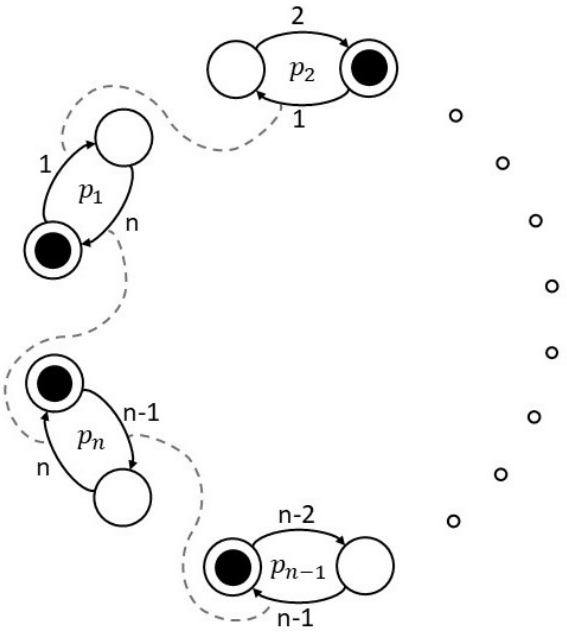

Figure 4: Illustration of how the order on the channels in D is maintained. We consider the case where $D = \{1, \ldots, n\}$ and p_i is in charge of channel i. The order between the channels is the natural order on $\{1, \ldots, n\}$. The black token indicates the current state for each process. Transitions 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 processes 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.

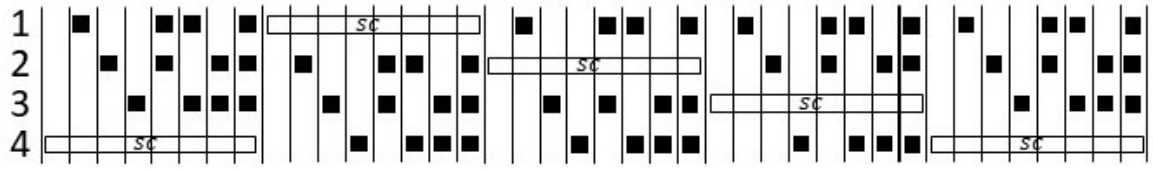

Figure 5: Illustration of how the set D and the switching channel sc change whenever there is a communication on sc. We consider the case where there are three processes and four channels. Each column corresponds to the status after one more communication on sc. Each channel is in turn the switching 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 $2^{\{1,\ldots,4\}\setminus\{\mathrm{sc}\}}$ according to set size first and then lexicographically on the sorted set.

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 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 $\mathcal{A}$ be the parallel composition of $(\mathcal{A}_p)$. A state for one process keeps track of 3 channels and one set of channels. However, the channel on which they "await" communication is either equivalent to their main channel or is the previous channel in the order. Thus a single bit is required to memorize this information. It follows that the size of a single process is in $O(n^2.2^n)$. The set of channels is maintained uniform across all processes and so is the identity of the switching channel. Furthermore, once we know which is the switching channel and we know which is the process in charge of the least other channel (i.e., channel 1 or in case that the switching channel is 1, channel 2), the rest of the association between the channels and processes respects a cyclic order. Therefore, the size of the state space of $\mathcal{A}$ is in $O(n^3.2^n)$ (rather than $(n^2.2^n)^{O(n)}$).

The language $\mathcal{L}(\mathcal{A})$ is complex, but can be decomposed in smaller parts. First, let $\mathcal{L}^{\mathrm{sc}}_D$ be the language over $C \setminus \{\mathrm{sc}\}$ respecting D: $w \in \mathcal{L}^{\mathrm{sc}}_D$ if the first occurrence of a letter in D in w is c_D, and for all $d \in D$ that occur in w, the next occurrence of D in w (if there is one) is $d \not\perp^D 1$. Note the absence of constraints for letters not in D. Second, let $\mathcal{L}^{\mathrm{sc}} = \mathcal{L}^{\mathrm{sc}}_{D_0} \cdot \mathrm{sc} \cdot \mathcal{L}^{\mathrm{sc}}_{D_1} \cdot \mathrm{sc} \cdot \ldots \cdot \mathrm{sc} \cdot \mathcal{L}^{\mathrm{sc}}_{D_{2^n-1}}$, with $D_0 = \emptyset$ and $D_{i+1} = inc_{<_{\mathrm{sc}}}(D_i)$ for all $0 \leq i < 2^n - 1$. This is the language of all words when sc is the current switching channel. Finally, we have $\mathcal{L}(\mathcal{A}) = (\mathcal{L}^{c_{n+1}} \cdot c_{n+1} \cdot \mathcal{L}^{c_1} \cdot c_1 \cdot \ldots \cdot \mathcal{L}^{c_n} \cdot c_n)^*$: we start with the language where c_{n+1} is the switching channel, then after the last switch c_1 assumes this role, and so on until the last switch using c_n at which point we end up back at the beginning.

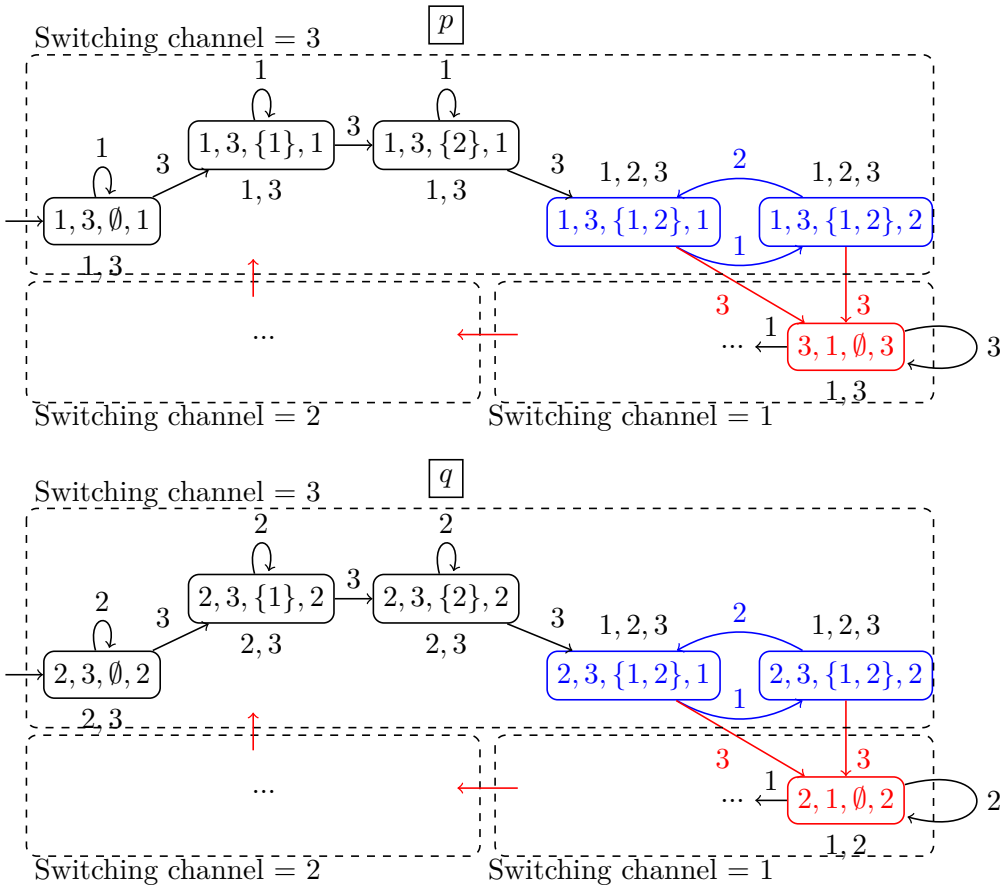

Figure 6: Illustration of the switching RAA for $n = 2$.

4.2. **Asynchronous Automata Construction.** We show that an AA that recognizes the same language as $\mathcal{A}$ 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 4.1. *Let $\mathcal{B}$ be an AA such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A})$. Each process in $\mathcal{B}$ is either fully-listening or trivializable.*

Proof. Let $\mathcal{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$, 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 $\mathcal{B}$. Then there is w a computation in $\mathcal{L}(\mathcal{B})$ such that p reaches state s_p after w. Consider the same computation in $\mathcal{A}$, and let sc be the current switching channel in $\mathcal{A}$ at the end of w. Let $\text{sc} \cdot c_1 \cdot \ldots \cdot c_{n-1} \cdot c$ be the sequence of channels

from sc to c according to the order from $\dotplus^{C}1$. Then there is a continuation w' of w of the form $\mathrm{sc}^{k_0} \cdot c_1^{k_1} \cdot \ldots \cdot c_{n-1}^{k_{n-1}}$ with $k_0, \ldots, k_{n-1} \in \mathbb{N}$ such that:

- $ww' \in \mathcal{L}(\mathcal{A})$,
- 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 $\mathcal{L}(\mathcal{A})$ 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 $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$. Therefore, from the state reached in $\mathcal{B}$ 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. □

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

Lemma 4.2. *The language $\mathcal{L}(\mathcal{A})$ is repetitive.*

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

So after at most 2^n communications on sc the switching channel becomes $\mathrm{sc}\dotplus^{C}1$. 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 $c_i\dotplus^{C}1$.

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

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

Now let $w_{\mathrm{loop}} = c_{n+1}^{2^n} \cdot c_1^{2^n} \cdots c_n^{2^n}$. 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 $\mathcal{L}(\mathcal{A})$. □

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 $\mathcal{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$ and an alternative initial configuration $\vec{t} = (t_p^0)_{p\in\mathbb{P}}$ we denote by $\mathcal{B}(\vec{t})$ the AA $\mathcal{B}(\vec{t}) = ((S_p)_{p\in\mathbb{P}}, (t_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$.

Theorem 4.3. *Let $\mathcal{B}$ be an AA such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A})$. There exists an alternative initial configuration $\vec{t} = (t_0)_{p\in\mathbb{P}}$ such that $\mathcal{L}(\mathcal{B}(\vec{t})) = \mathcal{L}(\mathcal{A})$ and each process in $\mathcal{B}(\vec{t})$ is either fully-listening or trivial.*

Proof. Let $\mathcal{B}$ be an AA equivalent to $\mathcal{A}$. By Theorem 4.1 there exists a word w such that after reading w all processes in $\mathcal{B}$ that are not fully listening reached a bottom SCC, where they accept all communications on all channels they are listening to. By Lemma 4.2, there exists a word w' such that $ww'\backslash\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B})$. Let $\vec{t} = (t_p^0)_{p\in\mathbb{P}}$ be the states that processes in $\mathcal{B}$ reach after reading ww'. Then $\mathcal{L}(\mathcal{B}(\vec{t})) = \mathcal{L}(\mathcal{B})$. The theorem follows. □

5. The Cost of Communication

In the previous section, we have shown that although they are equivalent in terms of language expressivity, reconfigurable automata allow for a much lower communication footprint than fixed automata. In this section, we introduce a measure that allows us to further study the differences. Our measure computes the cost of communication over an execution. We then use this measure to compare the costs of different executions in the two models clarifying what is the price of full dissemination and when (just in terms of communication cost) it becomes worthwhile to support reconfigurability. Finally, we give an alternative construction of an RAA (requiring full distribution) that suggests a different balance of costs than the one from Section 4.

5.1. **A Measure for Communication Cost.**

5.1.1. *Measure Definition.* First, let us define a measure that reflects how expensive it is to maintain channels and communicate on them. We distinguish two different costs, *passive* and *active*. The passive cost is defined as the number channels a process is connected to at the same time; the fewer there are the lower the cost. For a fixed automaton, this cost is constant over the whole execution. For a reconfigurable automaton, this cost is computed at each action and summed up over the execution. The second kind is the active cost, which refers to the cost incurred by a communication. The cost of a communication is defined as the number of participants in it.

Formally, let us fix some set of channels C and set of processes $\mathbb{P}$, an AA $\mathcal{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$ over dom, and an RAA $\mathcal{A} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\Delta_p)_{p\in\mathbb{P}}, (L_p)_{p\in\mathbb{P}})$. Given an execution $w = c_0c_1\cdots \in C^*$, we define the passive costs $\gamma_p^{\mathcal{A}}(w)$, $\gamma_p^{\mathcal{B}}(w)$ and the active costs $\gamma_a^{\mathcal{A}}(w)$, $\gamma_a^{\mathcal{B}}(w)$ as follows:

$$\gamma_p^{\mathcal{A}}(w) = \sum_{0\le i<|w|}\sum_{p\in\mathbb{P}} |dom^{-1}(p)| = |w|\cdot\sum_{c\in C}|dom(c)|$$

$$\gamma_p^{\mathcal{B}}(w) = \sum_{0\le i<|w|}\sum_{p\in\mathbb{P}} |L_p(s_p^i)|$$

$$\gamma_a^{\mathcal{A}}(w) = \sum_{0\le i<|w|} |dom(c_i)|$$

$$\gamma_a^{\mathcal{A}}(w) = \sum_{0\le i<|w|} |\{p \mid c_i \in L_p(s_p^i)\}|$$

where s_p^i denotes the state reached by process p in $\mathcal{A}$ after the prefix $c_0 \dots c_{i-1}$ of w. We define the costs $\gamma^{\mathcal{A}}(w)$ and $\gamma^{\mathcal{B}}(w)$ as their sums:

$$\gamma^{\mathcal{A}}(w) = \gamma_p^{\mathcal{A}}(w) + \gamma_a^{\mathcal{A}}(w)$$
$$\gamma^{\mathcal{B}}(w) = \gamma_p^{\mathcal{B}}(w) + \gamma_a^{\mathcal{B}}(w)$$

Our notion of active cost is related to the concept of *total communication cost* applied to measure performance of distributed algorithms. Total communication cost measures the number of bits transferred in order to complete a given protocol. We simplify this notion and adapt it to the theoretical concepts as follows. Noting that the constructions use local asynchronous automata and trivial message content, as active cost, we simply count the number of participants in each communication. Thus we abstract away from the details of information distribution through a network, routing information, and similar. A more thorough evaluation of communication costs will have to consider these aspects. For example, for asynchronous automata, (the log of) the product of the state spaces of the processes participating in a communication needs to be considered. Without a more detailed connection to actual implementations of these abstract notions of interaction it is hard to consider the actual costs of dissemination of information. In particular, disconnecting from a channel does not incur further cost in our model. But in a real implementation it would have to incur some communication costs.

Our passive cost is intended to capture the attention given by the processes to the communication. Thus, for every process and every channel that this process is connected to, we count a fixed cost. This could correspond, for example, to analyzing several types of header information on a single network to identify what to do with messages, or connection to several networks.

5.1.2. *Costs of Fixed and Reconfigurable Automata.* Now let us use this measure to calculute the costs of the RAA $\mathcal{A}$ built in Section 4 and its fixed AA equivalent $\mathcal{B}$ where all processes are fully-listening. Let $n = |\mathbb{P}|$. We fix an "average" execution $w_{\mathcal{A}}$ that first sets the dependent set from $\emptyset$ to some D, followed by k communications on D. Since there are 2^n possible choices for D, and the RAA has to cycle through all of them in order, we choose for our average execution the D in the middle, i.e. the dependent set reached after 2^{n-1} communications on sc. Let $D = \{c_1, \dots, c_d\}$. After setting the dependent set to D, we cycle through elements of D for some (large) number k of communications. In the end, the execution is $w_{\mathcal{A}} = \mathrm{sc}^{2^{n-1}} \cdot (c_1 \dots c_d)^{k/d}$. On the other hand, all the machinery used to set the dependent set to D in $\mathcal{A}$ is not relevant for $\mathcal{B}$. Therefore, in order to be fair, we omit this part of the execution when computing the cost for $\mathcal{B}$, and set $w_{\mathcal{B}} = (c_1 \dots c_d)^{k/d}$.

Let us now compute $\gamma^{\mathcal{A}}(w_{\mathcal{A}})$ and $\gamma^{\mathcal{B}}(w_{\mathcal{B}})$. First we start with $\mathcal{B}$. Every process in $\mathcal{B}$ is fully-listening, so $\gamma_p^{\mathcal{B}}(w_{\mathcal{B}}) = kn^2$. Moreover every communication involves all processes, so $\gamma_a^{\mathcal{B}}(w_{\mathcal{B}}) = kn$. Therefore the total cost for $\mathcal{B}$ is $\gamma^{\mathcal{B}}(w_{\mathcal{B}}) = k(n^2 + n)$.

Now let us turn to $\mathcal{A}$. At any moment, each process listens to either 2 or 3 channels: the current switching channel, the channel assigned to the process, and the channel previous to the assigned one according to the order of the dependent set if the assigned channel belongs to it. The exact number does not matter too much, only that it is a small (i.e. less than $O(n)$), so for the computation of the passive cost we just assume that each process is always connected to 3 channels. With this assumption, we get that $\gamma_p^{\mathcal{A}}(w_{\mathcal{A}}) = 3n(2^{n-1} + k)$. For the active cost, each communication on the switching channel in the first half of $w_{\mathcal{A}}$

involves all processes, and all communications in the second half only concern 2 processes, so $\gamma_a^{\mathcal{A}}(w_{\mathcal{A}}) = n2^{n-1} + 2k$. This gives a total cost of $\gamma^{\mathcal{A}}(w_{\mathcal{A}}) = k(3n+2) + 4n2^{n-1}$.

5.1.3. *Costs Analysis.* We can now compare for different values of k and n which model performs better in terms of communication costs. First, we remark that there is a large overhead in the machinery used to create the dependent set in $\mathcal{A}$, which means that if k is not much larger than n the fixed automaton $\mathcal{B}$ is cheaper, simply due to not including the machinery in its cost. Specifically, the cost for $\mathcal{A}$ starts being smaller when k is in $O(2^n/n)$, and will stay smaller for even higher values of k. Moreover, when k is so much larger that the second half of $\gamma^{\mathcal{A}}(w_{\mathcal{A}})$ is irrelevant compared to the first half, what remains is $k(n^2+n)$ for $\mathcal{B}$ and $k(3n+2)$ for $\mathcal{A}$. In other words, while both are understandably linear in k, we lower the dependency in n from squared to linear when using the reconfigurable automaton.

In the next section, we give an alternative construction of an RAA whose language requires full distribution that slightly increases the passive costs in exchange for a reduced machinery cost. We then re-analyze the cost/benefit calculation of using each of the constructions.

5.2. **Different Trade-off between Passive and Machinery Costs.** We outline a second construction that allows for a finer control on the set of dependent channels using an additional switching channel. Aside from the two switching channels, there is always one *highlighted* non-switching channel. The first switching channel, tg (for *toggling channel*), is used to add the highlighted channel to the dependent set. The second switching channel, hl (for *highlighting channel*), changes the highlighted channel to the next non-switching channel. With that setup, it is always possible to go from an empty dependent set to every set D in a linear number of steps by iterating on D: switch the highlighted channel to the next one in D, then toggle it, repeat. This takes at most n uses of hl to cycle through all channels and n uses of tg to add them. In the previous construction, it could take up to 2^n steps to reach a target set. Finally, after enough uses of hl, the two switching channels change so that every channel can eventually be one of the switching channels, and the dependent set is also reset to the empty set.

Technically, this is implemented using the following notions.

- As before, at every given moment of time there is a set of channels on which communication is unrestricted and (the complement) set of channels on which communication happens only in a fixed (cyclic) order.
- As before, at every given moment of time, every process is in charge of one channel. This process "initiates" communication on this channel.
- At every given moment of time, there is one channel that is a candidate to be included in the set of restricted channels. Each process keeps a record of this channel in their state space. The coordination between all processes ensures that all processes have a common view of which channel this is at all times.
- At every given moment of time, there are two global coordination channels: a toggling channel and a highlighting channel. Each process keeps a record of these channels in their state space. The coordination between all processes ensures that all processes have a common view of these channels.
  - A communication on the toggling channel adds the current highlighted channel to the set of restricted channels.

– A communication on the highlighting channel either tells all processes that they should jointly move to highlight the next possible channel. Once all channels have been considered as possible channel to be added, the identity of the toggling channel and the highlighting channel is changed.

Formally, let us first introduce some useful notations. We let $\mathbb{P} = \{p_1, \dots, p_n\}$ and $C = \{c_1, \dots, c_{n+2}\}$. As in the previous construction, 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 $+^D 1(d)$. As before we also use $-^D 1 : D \to D$ the inverse function and use the same notation. When the two switching channels tg and hl are known, we denote by $C^* = C \setminus \{\mathrm{tg}, \mathrm{hl}\}$ the set of non-switching channels. Finally, let $c^{1^\star}$ and $c^{0^\star}$ be two channels in C^* such that $c^{1^\star} = c^{0^\star} +^{C^*} 1$. Those two are used to check when we have cycled through every non-switching channel.

We build $\mathcal{A}_p^2 = (S_p, s_p^0, \Delta_p, L_p)$ for $p = p_k$ as follows:

- $S_p = \{(c, \mathrm{tg}, \mathrm{hl}, h, D, d) \mid c, \mathrm{tg}, \mathrm{hl}, h \in C, D \subseteq C \setminus \{\mathrm{tg}, \mathrm{hl}\}, d \in D \cup \{c\}\}$ is the set of states and $s_p^0 = (c_k, c_{n+1}, c_{n+2}, c^{1^\star}, \emptyset, c_k)$ is the initial state. As in the first construction, c is the channel currently assigned to p, initialized to c_k, which may change when the switching channels are replaced. The channels tg and hl are those two switching channels. Channel h is the highlighted non-switching channel. Finally, D is the current dependent set and d the next channel available for a communication for p. As in the previous construction, the behavior for all D such that $|D| \leq 1$ is the same.
- All processes listen to both switching channels and their assigned channel, plus, potentially, the previous channel in D if D contains the assigned channel:

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

- The transition Δ_p is the union of the following seven sets:

$$\{(c, \mathrm{tg}, \mathrm{hl}, h, D, c), c, (c, \mathrm{tg}, \mathrm{hl}, h, D, c -^D 1)\} \tag{5.1}$$

$$\{(c, \mathrm{tg}, \mathrm{hl}, h, D, c -^D 1), c -^D 1, (c, \mathrm{tg}, \mathrm{hl}, h, D, c)\} \tag{5.2}$$

The first two transitions are similar to the first two of the previous construction.

$$\{(c, \mathrm{tg}, \mathrm{hl}, h, D, d), \mathrm{tg}, (c, \mathrm{tg}, \mathrm{hl}, h, E, c -^E 1) \mid c \neq h \wedge E = D \cup \{h\}\}\} \tag{5.3}$$

$$\{(c, \mathrm{tg}, \mathrm{hl}, h, D, d), \mathrm{tg}, (h, \mathrm{tg}, \mathrm{hl}, h, D \cup \{h\}, h) \mid c = h\} \tag{5.4}$$

These transitions are the toggling mechanism, which adds the currently highlighted channel h to the dependent set. If the followed channel is not h then update the previous channel according to the set $D \cup \{h\}$ (5.3). If the followed channel is h then expect the next communication on h (5.4).

The remaining are the highlighting mechanism. When a communication on hl occurs, the highlighted channel changes to the next one.

$$\{(c, \mathrm{tg}, \mathrm{hl}, h, D, d), \mathrm{hl}, (c, \mathrm{tg}, \mathrm{hl}, h +^{C^*} 1, D, d) \mid h \neq c^{0^\star}\} \tag{5.5}$$

If the highlighted channel is not maximal then go to the next highlighted channel.

$$\{(c, \mathrm{tg}, \mathrm{hl}, c^{0^\star}, D, d), \mathrm{hl}, (c, \mathrm{hl}, \mathrm{hl} +^C 1, c^{1^\star}, \emptyset, c) \mid c \neq \mathrm{hl} +^C 1\} \tag{5.6}$$

$$\{(c, \mathrm{tg}, \mathrm{hl}, c^{0^\star}, D, d), \mathrm{hl}, (\mathrm{tg}, \mathrm{hl}, \mathrm{hl} +^C 1, c^{1^\star}, \emptyset, \mathrm{tg}) \mid c = \mathrm{hl} +^C 1\} \tag{5.7}$$

If we have already cycled through all non-switching channels once (the current highlighted channel is maximal), then new channels assume the roles of toggling and highlighting, and

the highlighted channel and dependent set are both reset. We may also have to update the first component if the channel assigned to this process becomes a switching channel. That is, either the assigned channel is untouched by the change (5.6), or it is assigned to hl (5.7).

We give an illustration of this construction in Figure 7. As in the previous example, we have two processes p and q whose initial assigned channels are 1 and 2 respectively. Channels 3 and 4 serve as the toggling and highlighting channels. Following the upper path toggles both 1 and 2 and adds them to the dependent set, eventually reaching the blue states where those two have to be played in the correct order. The lower path skips toggling 1 and therefore does not reach a state with more than one channel in D. In any case, after the highlighting channel is used twice, the two switching channels change. This leads to the state in red where 4 and 1 are the new toggling and highlighting channels respectively. For p this means that its assigned channel becomes 3, while q's does not change. The path then continues, crossing into new areas of the automaton with each switching channels change, and eventually leads back to the initial state.

Let $\mathcal{A}^2$ be the parallel composition of $(\mathcal{A}^2_p)$. As with the previous construction we show that for every equivalent AA, processes are either listening to all channels or can be led to a trivial component.

Lemma 5.1. *Let $\mathcal{B}$ be an AA such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A}^2)$. Each process in $\mathcal{B}$ is either fully-listening or trivializable.*

Proof. Similarly to the previous construction, let $\mathcal{B} = ((S_p)_{p\in\mathbb{P}}, (s^0_p)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$, and let $p \in \mathbb{P}$. Again, assume that p is not fully-listening, so let $C_p = dom^{-1}(p) \subsetneq C$ and let $c \in C \setminus C_p$ be a channel that p does not listen to.

Start from some state s_p reachable by p in $\mathcal{B}$. By definition, there is a computation w in $\mathcal{L}(\mathcal{B})$ such that p reaches state s_p after w. Consider the same computation w in $\mathcal{A}^2$. The computation ends in some configuration where $(\mathrm{tg}, \mathrm{hl})$ are the current toggling and highlighting channels respectively. We want to find a continuation of this computation where c is the highlighting channel afterwards. To that end, let $\mathrm{hl} \cdot c_1 \cdot ... \cdot c_{n-1} \cdot c$ be the sequence of channels from hl to c according to the order from $\circlearrowleft 1$. Then there exists a continuation w' of the form $w' = \mathrm{hl}^{k_0} \cdot c_1^{k_1} \cdot ... \cdot c_{n-1}^{k_{n-1}}$ with $k_0, \ldots, k_{n-1} \in \mathbb{N}$ such that:

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

Note that we do not make assumptions regarding the toggling channel tg.

We show that every possible continuation $w'' \in (C \setminus \{c\})^*$ must be possible in p. Consider such an arbitrary continuation $w'' \in (C \setminus \{c\})^*$. If w'' involves no communication on tg, then the dependent set stays empty and none of the switching channels change, so $ww'w''$ still belongs to $\mathcal{L}(\mathcal{A})$. In particular, p cannot block communications in C_p. If w'' does involve communications on tg, then after the first such communication the dependent set is updated to $\{c^{1^\star}\}$. Afterwards, the dependent set does not change anymore and neither do the switching channels. Since the dependent set is only a singleton, all communications on non-switching channels are allowed to go through in every possible order. Therefore in that case as well $ww'w'' \in \mathcal{L}(A)$. Again, p cannot block communications in C_p.

Since it holds for every such continuation, it holds *a fortiori* for all continuations $w'' \in C_p^*$. Therefore p is indeed in a strongly connected component implementing all transitions in its domain. □

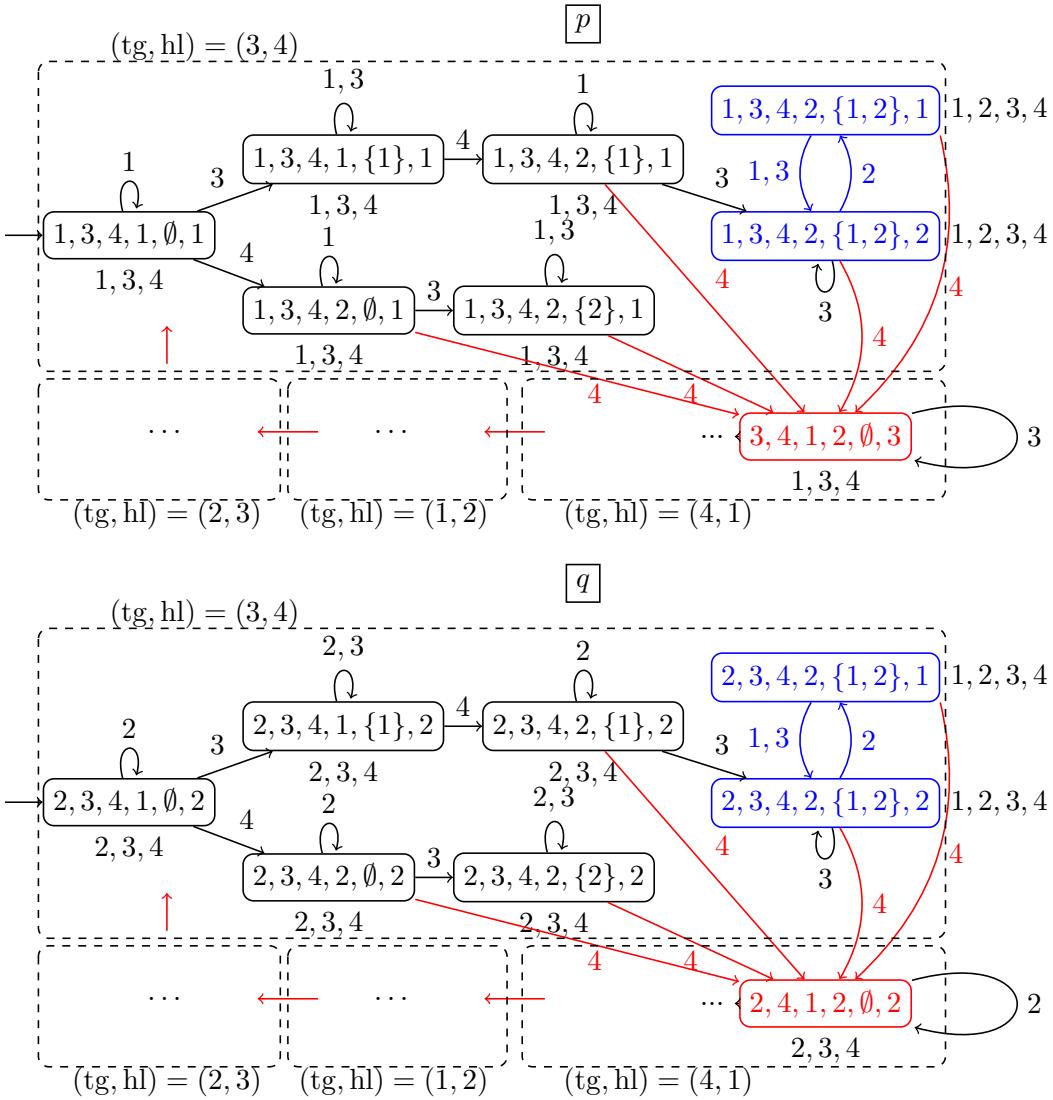

Figure 7: Illustration of the second construction for $n = 2$.

We note that it is possible to modify the last construction so that only the hl channel is dynamic and the tg channel is fixed. This, however, complicates the notations further.

As before, the AA $\mathcal{B}$ has an alternative initial configuration making all non fully listening processes trivial.

Lemma 5.2. *The language $\mathcal{L}(\mathcal{A}^2)$ is repetitive.*

Proof. The proof is similar to the proof of Lemma 4.2. By repeatedly communicating on the highlighting channel hl, the highlighting and toggling channels change. Repeated communication on the next highlighting channel leads to the change occurring again. After $n + 2$ changes of the toggling and the highlighting channel, the same channels become the highlighting and toggling channels again. We call the highlighting and toggling channels returning to their initial values *a loop*. If during a loop a process is "in charge" of channel

c, then in the next loop the same process is "in charge" of $(c \overset{C}{-} 1) \overset{C}{-} 1$. So, if $|C|$ is even, after $|C|/2$ loops all processes are back in charge of their original channels. If $|C|$ is odd, after $|C|$ loops all processes are back in charge of their original channels. In both cases, all processes return to their original initial states. □

Lemma 5.3. *Let $\mathcal{B}$ be an AA such that $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{A}^2)$. There exists an alternative initial configuration $\vec{t} = (t_0)_{p \in \mathbb{P}}$ such that $\mathcal{L}(\mathcal{B}(\vec{t})) = \mathcal{L}(\mathcal{A}^2)$ and each process in $\mathcal{B}(\vec{t})$ is either fully-listening or trivial.*

Proof. The proof is identical to the proof of Lemma 4.3. □

Communication Costs. Finally, we compute the communication cost of this new automaton $\mathcal{A}^2$ and compare it to the previous one. As explained previously, we only need a linear amount of operations to set the dependent set to any D, as opposed to exponential in the previous construction. Therefore, we set $w_2 = (\text{tg} \cdot \text{hl})^{n/2} \cdot (c_1 \dots c_d)^{k/d}$ to represent an "average" execution. With a similar computation as with the first construction, we get that $\gamma_p^{\mathcal{A}^2}(w_2) = 4n(n+k)$ and $\gamma_a^{\mathcal{A}^2}(w_2) = n^2 + 2k$, for a total cost of $\gamma^{\mathcal{A}^2}(w_2) = k(4n+2) + 5n^2$.

Compared with the first construction's cost of $\gamma^{\mathcal{A}}(w_{\mathcal{A}}) = k(3n+2) + 4n2^{n-1}$, we see a small increase in the factor of k, due to the additional switching channel increasing the passive cost. In exchange, we obtain a massive reduction in the cost of the machinery needed to set up the dependent set, going from exponential to simply squared. First we compare $\mathcal{A}^2$ with the equivalent fixed automaton $\mathcal{B}$. Remember that the cost for $\mathcal{B}$ was $\gamma^{\mathcal{B}}(w_{\mathcal{B}}) = k(n^2+n)$. For very small values of n, meaning up to 3, $\mathcal{B}$ will be more efficient for any value of k. Starting from $n = 4$, $\mathcal{A}^2$ starts being more efficient when k is high enough:

Value of n	1	2	3	4	5	6	7	8	9 – 11	12 – 14
$\gamma^{\mathcal{A}^2}(w_2)$ is lower for k at least	—	—	—	40	16	12	10	9	8	7

Specifically, $\mathcal{A}^2$ starts being more efficient than $\mathcal{B}$ when $k \geq 5n^2/(n^2 - 3n - 2)$, so for high values of n only ~5 communications are enough to offset the set-up cost. Now comparing $\mathcal{A}$ and $\mathcal{A}^2$, the slightly larger factor for k in $\mathcal{A}^2$ makes it less efficient when k is really large compared to n, specifically when $k \geq 4n2^{n-1} - 5n^2$. Still, in both cases the factor of k is linear in n, so we argue that the difference between the two is minimal.

6. Other Reconfiguration Approaches

While the connection to Zielonka / Asynchronous automata has been discussed in the introduction and, in a sense, is the main topic of this paper, we would like discuss briefly reconfiguration in a more general context. Here we discuss other approaches to reconfiguration and the general issues that arise when considering reconfiguration in the larger sense of the word.

First, in general formal methods "model-based approaches", the concepts of reconfiguration and dynamism appear slightly differently than in our work. We include several examples below.

Dynamic Timed Automata are timed automata that change their transitions during run time. This does not affect the communication [TGH+23].

In order to reason about them the automata are translated directly into timed automata, where model checking and other analysis can be applied. This is similar to our work in the sense that the advanced features can be compiled away. That is, in the same way we can translate RAA to AA.

In their case, the justification for using dynamism is convenience of modelling but analysis is done on the original model. In our case, the justification for using reconfiguration is in measuring the different communication costs. But, throughout our work, we do the analysis on top of the more expressive model claiming that it does not makes sense to just reduce it to AA and the like.

Reconfigurable Hierarchical Timed Automata are similar to dynamic timed automata but the approach is derived from software engineering [BKKL19]. As before, they introduce modelling convenience through high level syntactical elements that can be compiled away into the original model.

Reconfigurable Asynchronous Logic Automata are a variant of cellular automata [GDC+10]. The grid and communication between different cells is fixed so reconfiguration is used in a very different sense than in the current work.

Dynamic Communication Automata and Dynamic Reactive Modules include two types of automata that are allowed to create new automata. Dynamic Communication Automata are automata that communicate over unbounded buffers [BCH+13]. All problems about the general model are undecidable and here also creation of automata is added, making the problem even more infinite. There are some complexity results on analysis in restricted cases. As before, the notion of dynamism is quite different from that discussed in the current work. Dynamic Reactive Modules combine the creation of new automata with the transfer of pointers to the variables of different automata [FHN+11]. As before, the issue of creation leads to a model on which reasoning is undecidable and the main purpose of this work is to show coherent definitions and opportunities. The usage of pointers and pointer dereferencing allows, in principle, to change during runtime the connection between different parts of the system. However, the communication is done strictly by variable sharing and the ideas of changeable communication is obfuscated by the creation of new automata. Reconfiguration per-se is not studied.

Reconfigurable Automata Networks are related to interface automata [dAH01] but include also behavior of the components (cf. [GKS17] for examples). The reconfiguration relates to the abilities of the formalism to reuse components and interfaces in different ways and does not support reconfiguration during runtime.

Second, looking further afield from "model-based approaches", we would like to recognize a relation between temporal graphs and graph rewriting systems to our ideas of reconfiguration.

Temporal Graphs are graphs in which edges are present only at specific times, typically modelled either as time-stamped edge sets or as sequences of graph snapshots [CFQS12]. They incorporate the timing of interactions directly into the graph structure and, thus, the dynamism is exogenous. The central focus is on efficiently computing graph properties (such as connectivity or reachability) over the evolution. Our reconfigurable systems can be seen as generative models of temporal graphs, where connectivity evolves according to

local rules. Our interest is in properties that hold over all evolution rather than efficient computation over a fixed instance.

Graph Rewriting Systems are formal models in which graphs evolve by the application of local rewriting rules that replace a subgraph matching a given pattern with another graph, sometimes under constraints [EEPT06]. The studies of such systems concentrate on reachability of graph configurations or termination. Dually, our reconfiguration is more operational state-based change and the analysis targets the evolution of the entire system and interested in the computation path.

Our analysis of translation between models would be akin to considering a global structure in which analysis questions about the formalism can be expressed. While our transformation shows that the translation between AA and RAA is possible and effective and that its cost is in the dissemination of information, which we try to quantify, embedding "dynamism" into a single structure in other contexts might involve a huge blow up of the artefact.

7. 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. Thus, the conversion leads to an increase in communications that can be avoided with reconfigurability. We compute a cost measure for communications and show when reconfigurable automata perform better than their fixed counterpart.

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? To start answering these questions, we introduced a measure of cost of communication that is partitioned between passive and active costs. The passive cost corresponds to the machinery that is used and the active costs corresponds to the actual interaction. For this notion of cost, we show some tradeoffs between AA and different RAA for the same languages. We leave further studies of the *cost of communication* as an interesting question for further research.

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 would like to thank Y. Abd Alrahman and L. Di Stefano for fruitful discussions and suggestions; A. Muscholl for highlighting the nondeterminism of RAAs; and an anonymous reviewer for critisizing the notion of trivializable leading to the results on full triviality.

References

[AADNL19] Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. A calculus for collective-adaptive systems and its behavioural theory. *Information and Computation*, 268:104457, 2019. doi:10.1016/j.ic.2019.104457.

[ADNL+15] Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi, and Roberto Vigo. A calculus for attribute-based communication. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1840–1845, 2015. doi:10.1145/2695664.2695668.

[AP21] Yehia Abd Alrahman and Nir Piterman. Modelling and verification of reconfigurable multi-agent systems. *Auton. Agents Multi Agent Syst.*, 35(2):47, 2021. doi:10.1007/s10458-021-09521-x.

[BCH+13] Benedikt Bollig, Aiswarya Cyriac, Loïc Hélouët, Ahmet Kara, and Thomas Schwentick. Dynamic communicating automata and branching high-level mscs. In Adrian-Horia Dediu, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, pages 177–189, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BKKL19] Roufaida Bettira, Laid Kahloul, Mohamed Khalgui, and Zhiwu Li. Reconfigurable hierarchical timed automata: Modeling and stochastic verification. In *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, pages 2364–2371, 2019. doi:10.1109/SMC.2019.8913890.

[CFQS12] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012. doi:10.1080/17445760.2012.668546.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In A Min Tjoa and Volker Gruhn, editors, *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pages 109–120. ACM, 2001. doi:10.1145/503209.503226.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Graph Transformation Systems*, pages 37–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi:10.1007/3-540-31188-2_3.

[FHN+11] Jasmin Fisher, Thomas A. Henzinger, Dejan Nickovic, Nir Piterman, Anmol V. Singh, and Moshe Y. Vardi. Dynamic reactive modules. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, Lecture Notes in Computer Science, pages 404–418. Springer, 2011. doi:10.1007/978-3-642-23217-6_27.

[GDC+10] Neil Gershenfeld, David Dalrymple, Kailiang Chen, Ara Knaian, Forrest Green, Erik D. Demaine, Scott Greenwald, and Peter Schmidt-Nielsen. Reconfigurable asynchronous logic automata: (rala). In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 1–6, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1706299.1706301.

[GGMW10] Blaise Genest, Hugo Gimbert, Anca Muscholl, and Igor Walukiewicz. 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*, pages 52–63. Springer, 2010. doi:10.1007/978-3-642-14162-1_5.

[GKS17] Alessandro Gianola, Stefano Kasangian, and Nicoletta Sabadini. Cospan/span(graph): an algebra for open, reconfigurable automata networks. In Filippo Bonchi and Barbara König, editors, *7th Conference on Algebra and Coalgebra in Computer Science, CALCO 2017, Ljubljana, Slovenia, June 12-16, 2017*, volume 72 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.CALCO.2017.2.

[GM06] Blaise Genest and Anca Muscholl. 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*, pages 565–576. Springer, 2006. doi:10.1007/11787006_48.

[HLP24] Daniel Hausmann, Mathieu Lehaut, and Nir Piterman. Distribution of reconfiguration languages maintaining tree-like communication topology. In S. Akshay, Aina Niemetz, and Sriram Sankaranarayanan, editors, *Automated Technology for Verification and Analysis - 22nd International Symposium, ATVA 2024, Kyoto, Japan, October 21-25, 2024, Proceedings, Part I*, Lecture

Notes in Computer Science, pages 160–180. Springer, 2024. doi:10.1007/978-3-031-78709-6_8.

[KM13] Siddharth Krishna and Anca Muscholl. A quadratic construction for Zielonka automata with acyclic communication structure. *Theoretical Computer Science*, 503:109–114, 2013. doi:10.1016/j.tcs.2013.07.015.

[LP23] Mathieu Lehaut and Nir Piterman. Adding reconfiguration to Zielonka's asynchronous automata. *arXiv preprint arXiv:2305.01425*, 2023.

[MKS00] Madhavan Mukund, K Narayan Kumar, and Milind Sohoni. Synthesizing distributed finite-state systems from MSCs. In *International Conference on Concurrency Theory*, pages 521–535. Springer, 2000. doi:10.1007/3-540-44618-4_37.

[MS97] Madhavan Mukund and Milind Sohoni. Keeping track of the latest gossip in a distributed system. *Distributed Computing*, 10:137–148, 1997. doi:10.1007/s004460050031.

[MW14] Anca Muscholl and Igor Walukiewicz. Distributed synthesis for acyclic architectures. *arXiv preprint arXiv:1402.3314*, 2014. Conference version FST&TCS 2014.

[TGH+23] Samir Tigane, Fayçal Guerrouf, Nadia Hamani, Laid Kahloul, Mohamed Khalgui, and Masood Ashraf Ali. Dynamic timed automata for reconfigurable system modeling and verification. *Axioms*, 12(3), 2023. doi:10.3390/axioms12030230.

[Zie87] Wieslaw Zielonka. Notes on finite asynchronous automata. *RAIRO Theor. Informatics Appl.*, 21(2):99–135, 1987. doi:10.1051/ita/1987210200991.

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany