RAIRO-Theor. Inf. Appl. 48 (2014) 3–22
DOI: 10.1051/ita/2013046

Available online at:
www.rairo-ita.org

SYSTEMS OF PARALLEL COMMUNICATING RESTARTING AUTOMATA

Marcel Vollweiler[1] and Friedrich Otto[1]

Abstract. Several types of systems of parallel communicating restarting automata are introduced and studied. The main result shows that, for all types of restarting automata X, centralized systems of restarting automata of type X have the same computational power as non-centralized systems of restarting automata of the same type and the same number of components. This result is proved by a direct simulation. In addition, it is shown that for one-way restarting automata without auxiliary symbols, systems of degree at least two are more powerful than the component automata themselves. Finally, a lower and an upper bound are given for the expressive power of systems of parallel communicating restarting automata.

Mathematics Subject Classification. 68Q45.

1. Introduction

Here we define and investigate *systems of parallel communicating restarting automata* in detail. These systems, which were announced in [4], combine two well-known concepts: the *restarting automaton* and the problem solving strategy known as *classroom model*.

The restarting automaton was originally developed by Jančar et al. to model the linguistic technique of "analysis by reduction" [6]. This technique is used for checking the syntactical correctness of sentences of a natural language with free word order. A given sentence is simplified stepwise by a sequence of local rewrite steps until either a correct core sentence is obtained or until an error is located. Meanwhile, many variants and extentions of the original model of the restarting

Keywords and phrases. Parallel communicating system, restarting automaton, language class.

[1] Fachbereich Elektrotechnik/Informatik, Universität Kassel, 34109 Kassel, Germany.
{vollweiler,otto}@theory.informatik.uni-kassel.de

Article published by EDP Sciences

© EDP Sciences 2014

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

automaton have been defined, relating them to various well-known classes of formal languages (see [13] for an overview). Also *cooperating distributed systems* of restarting automata have been studied. In these systems, several components work together in processing an input string in a sequential manner [8]. These systems are closely related to nonforgetting restarting automata [8], and recently it has been shown that some classes of trace languages can be characterized by certain types of cooperating distributed systems of a very simple kind of restarting automaton [10,11].

While the cooperating distributed systems can be seen as an abstraction of the problem solving strategy known as *blackboard model*, in which a finite group of agents take turns in working on a solution of a common instance of some problem, the parallel communicating systems are an abstraction of the *classroom model*, where a finite group of agents work in a parallel manner, each on its own copy of a common instance of some problem. While they work in parallel, but independently of each other, the agents (the components) are allowed to exchange information by passing messages (see, *e.g.*, [2]). Parallel communicating systems have been introduced in the context of phrase-structure grammars in [15]. These so-called parallel communicating grammar systems are discussed in detail in [2]. Since then, also parallel communicating systems of finite-state automata [7] or pushdown automata [3] have been studied.

Here we combine a finite set of restarting automata into a *parallel communicating system* (*PC system*). These restarting automata, which are called the *components* of the PC system, work in parallel and independently of each other on their own working tapes (starting with the same input), but they can exchange information by sending and receiving messages. There are various options for realizing this message passing. Here we use particular internal states, called *communication states*, to realize a communication in the form of a two-way handshake.

Our communication protocol is quite different from those of the above mentioned PC systems. There, the components work in a synchronous way, *i.e.*, there is a global clock that forces all components to execute exactly one computation step in each unit of time. Synchronization can be seen as hidden communication. For instance, a PC system of finite automata with three components is able to accept a non-contextfree language without any (explicit) communication by simply using synchronization [7]. As synchronization is not natural for distributed systems, the local computations of the components of our systems are not synchronized. This means that the numbers of steps of the local computations of two components that are executed between two communications may differ. Moreover, the communication of the mentioned PC systems is one-directional, *i.e.*, only the component that requests some information from another component determines when and with which other component a communication takes place. The component that sends information to the requesting one is not aware of that communication and thus is only passively involved in that communication step. In addition, particularly in PC systems of finite automata, the requesting component forgets everything about its previous computation in each computation step. Here, we want to avoid

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

these particular properties. Therefore, we use a completely different asynchronous point-to-point communication protocol for PC systems of restarting automata. This protocol was also applied for PC systems of finite automata (asynchronous PC systems of finite automata [16]) and (in a modified version) for PC systems of pushdown automata (asynchronous PC systems of pushdown automata [14]).

In general, a communication can occur between any two components of a PC system. If there is a particular component, called the *master* of the system, such that all communications involve this component, then the PC system is called *centralized*. While for PC grammar systems and (synchronous) PC systems of deterministic finite automata the centralized variants are in general less powerful than the non-centralized variants, we will see that for PC systems of restarting automata, the centralized variants have exactly the same computational power as the non-centralized variants. In fact, our main result states that each non-centralized system of degree $n \geq 2$ of restarting automata of any type X can be simulated by a centralized system of the same degree of restarting automata of the same type X. As it turns out it is the fact that by each restart operation, the state of a restarting automaton is reset to its initial state that causes the main problems for this simulation.

This paper is organized as follows. After restating in short the necessary notions and notation on restarting automata, we define the PC systems of restarting automata (*PCRA systems*, for short), and explain in detail the way in which they work, using a simple example. In Section 3, we state our main result and provide a rather detailed proof. Then we compare the computational power of PCRA systems to the power of restarting automata in Section 4, and to that of systems of parallel communicating finite automata and of multi-head automata in Section 5. In this way, the power of PCRA systems is related to the space complexity classes L (deterministic logarithmic space), NL (nondeterministic logarithmic space), and CSL (nondeterministic linear space).

2. Restarting automata and PCRA systems

A restarting automaton M consists of a finite-state control and a flexible tape with a read/write window. Formally, it is described by an 8-tuple $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, where Q is a finite set of states, Σ is a finite input alphabet, Γ is a finite tape alphabet containing Σ, and $q_0 \in Q$ is the initial state. The symbols in $\Gamma \setminus \Sigma$ are called *auxiliary symbols*, the symbols $¢, \$ \notin \Gamma$ are used as left and right delimiters of the tape, and the integer $k \geq 1$ denotes the size of the read/write window. Further, δ is a transition relation that assigns finite sets of possible operations to pairs of the form (p, α), where $p \in Q$ and $\alpha \in \{\varepsilon, ¢\} \cdot \Gamma^* \cdot \{\varepsilon, \$\}$ is a possible content of the window. Here ε denotes the empty word.

There are five different operations that M can use:

– A *move-left step* is of the form $(q, \mathsf{MVL}) \in \delta(p, \alpha)$. It causes M to change into state q and to move the window one position to the left. However, the window cannot move, if it already contains the left delimiter ¢.

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

– A *move-right step* is of the form $(q, \mathsf{MVR}) \in \delta(p, \alpha)$. It causes M to change into state q and to move the window one position to the right. However, the window cannot move, if it just contains the right delimiter \$.

– A *rewrite step* is of the form $(q, \beta) \in \delta(p, \alpha)$, where $|\beta| < |\alpha|$. It causes M to change into state q, to replace α by β, thereby also shortening the tape accordingly, and to move the window immediately to the right of β. Some restrictions apply in that the symbols ¢ and \$ cannot be rewritten.

– A *restart step* is of the form $\mathsf{Restart} \in \delta(p, \alpha)$. It causes M to change into the initial state q_0 and to reposition the window on the left end of the tape.

– An *accept step* is of the form $\mathsf{Accept} \in \delta(p, \alpha)$. It causes M to halt and accept.

If $\delta(p, \alpha) = \emptyset$, then M gets stuck (without acceptance) in a corresponding situation. It is required that in any computation, *rewrite and restart steps occur alternatingly*, with a rewrite step coming first. In general, a restarting automaton M is nondeterministic, but if δ is a (partial) function, then M is a *deterministic* restarting automaton.

A *configuration* κ of a restarting automaton M is either of the form $\kappa = \mathsf{Accept}$, which is called an *accepting configuration*, or of the form $\kappa = \alpha q \beta$, where $q \in Q$ and either $\alpha = \varepsilon$ and $\beta \in \{¢\} \cdot \Gamma^* \cdot \{\$\}$ or $\alpha \in \{¢\} \cdot \Gamma^*$ and $\beta \in \Gamma^* \cdot \{\$\}$. Here q is the current state of M, and $\alpha\beta$ is the current tape content with the window containing the first k symbols of β, if $|\beta| \geq k$, or all symbols of β including the right end marker \$, otherwise. For an input word $w \in \Sigma^*$, $q_0 ¢ w \$$ is the corresponding *initial configuration*, and any configuration of the form $q_0 ¢ u \$$ with $u \in \Gamma^*$ is called a *restarting configuration*.

The transition relation δ induces a binary relation $\vdash_M$ on the set of configurations of M, the so-called *single-step computation relation*. Its reflexive and transitive closure $\vdash_M^*$ is the *computation relation* of M. The language $L(M)$ that is accepted by M is defined as $L(M) = \{ w \in \Sigma^* \mid q_0 ¢ w \$ \vdash_M^* \mathsf{Accept} \}$.

The behaviour of a restarting automaton M can be described informally as follows: beginning in a restarting configuration $q_0 ¢ u \$$ for some $u \in \Gamma^*$ (this includes in particular the initial configurations), M can move its window along the tape, then it rewrites a part of the tape, then it can move the window again, and then M performs a restart step. This takes M back into a restarting configuration, and so this process can be repeated until an accepting configuration is reached or until M gets stuck.

The type of restarting automaton defined above is called an RLWW-automaton. By putting additional restrictions on the definition, various restricted types of restarting automata can be obtained. An RRWW-automaton is an RLWW-automaton that is not allowed to use any MVL-steps. Thus, apart from the restart steps, such an automaton can move its window only from left to right. Accordingly, this type is also called a *one-way restarting automaton*. An RWW-automaton is an RRWW-automaton that has to restart immediately after performing a rewrite step. In other words, an RWW-automaton is not allowed to move the window between

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

a rewrite and a restart step, and hence, the rewrite and the restart steps can be merged into a single operation for these automata. Furthermore, also the rewrite operation can be restricted. An RLWW-automaton (RRWW-, RWW-automaton) is an RLW-automaton (RRW-, RW-automaton), if no auxiliary symbols are available, that is, if $\Sigma = \Gamma$. Finally, an RLW-automaton (RRW-, RW-automaton) is an RL-automaton (RR-, R-automaton), if all its rewrite operations are of the form $(q, \beta') \in \delta(p, \beta)$, where β' is a scattered subword of β. This means that β' can be obtained by deleting one or more symbols from β. To simplify the notation in what follows, we set $\mathcal{T} = \{RLWW, RLW, RL, RRWW, RRW, RR, RWW, RW, R\}$. To denote types of deterministic restarting automata, we use the prefix det. Thus, a deterministic automaton of type $\mathsf{X} \in \mathcal{T}$ is a det-X-automaton. The class of languages that are accepted by any automaton of type (det-)X, $\mathsf{X} \in \mathcal{T}$, is denoted by $\mathcal{L}((\text{det-})\mathsf{X})$.

Now we come to the main definition of this paper. A *system of parallel communicating restarting automata* (a *PCRA system*, for short) of degree n is given by an n-tuple $\mathcal{M} = (M_1, M_2, \ldots, M_n)$, where $M_i = (Q_i, \Sigma, \Gamma_i, \mathfrak{c}, \$, q_i, k_i, \delta_i)$, $1 \leq i \leq n$, are restarting automata, which are called the *components* of the system. The system $\mathcal{M}$ works as follows. Given an input $w \in \Sigma^*$, all components start from their corresponding initial configurations for input w. They perform local operations, concurrently and independently of each other. At some point, a component M_i may request information from another component M_j, or M_j may have information for M_i. The former situation is realized by M_i entering a special internal state req^j_d, called a *request state*, where the superscript j indicates that M_i needs information from M_j, and the subscript d is some local information that M_i wants to remember. The other situation is realized by M_j entering a special internal state $\mathsf{res}^i_{d',c}$, called a *response state*, where the superscript i indicates that M_j has information for M_i, the subscript c encodes this particular information, and the subscript d' is some local information that M_j wants to remember.

If M_i is in state req^j_d and M_j is in state $\mathsf{res}^i_{d',c}$, then a *communication* takes place: M_i enters the internal state $\mathsf{rec}^j_{d,c}$, which is called a *receive state*, indicating that it has received information c from M_j, and M_j enters its internal state $\mathsf{ack}^i_{d',c}$, called an *acknowledge state*, which indicates that it has just sent information c to M_i. Observe how the superscripts of these states of M_i and M_j indicate the communication partners, the subscript c encodes the information transmitted, and the subscripts d and d' denote local information that is preserved by the communication step. If only one of M_i and M_j is in the internal state indicated above, then this component is suspended until the other component reaches its corresponding internal state. If that never happens, then the former component is blocked for the rest of the computation. However, also with one or more blocked components, the system continues with the computation using only the non-blocked components. Accordingly, we assume that, for each component M_i, $1 \leq i \leq n$, the set of internal states Q_i contains a finite number of *communication states*, that is, request states, response states, receive states, and acknowledge states. Notice that there are only finitely many communication states altogether, which implies that the number of different messages c that are communicated by any communication is finite, too.

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

As a *communication step* can be executed, if and only if a component M_i reaches a request state req_d^j through its local computation, and the component M_j reaches a corresponding response state $\mathsf{res}_{d',c}^i$, we admit transition steps of the form $\mathsf{req}_d^j \in \delta_i(q,\alpha)$ and $\mathsf{res}_{d,c}^j \in \delta_i(q,\alpha)$ for all components M_i. By a communication step, the component M_i being in state req_d^j is put into a receive state $\mathsf{rec}_{d,c}^j$, and the component M_j is put into the acknowledge state $\mathsf{ack}_{d',c}^i$, whereby they keep the current window positions. Thereafter, both components continue with their independent local computations. Thus, the receive and acknowledge states can be interpreted as the successor states of the request and response states, respectively.

The type of the components of a PCRA system determines the type of the system. If all its components are restarting automata of type X for some $\mathsf{X} \in \mathcal{T}$, then the system is said to be of type PC-X, and is called a PC-X-system. A system of type X and degree n is called a PC-X(n)-system. Moreover, if the size of the windows of all components is limited by the constant k, then we talk about PC-X(n,k)-systems.

For a formal description of the behaviour of a PCRA system, we need to define the concepts of configurations and computations. A *configuration* of a PCRA system $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ of degree n is an n-tuple $K = (\kappa_1, \kappa_2, \ldots, \kappa_n)$, where κ_i is a configuration of component M_i, $1 \leq i \leq n$. For an input word $w \in \Sigma^*$, the corresponding initial configuration of $\mathcal{M}$ is $K_0 = (q_1 ¢ w \$, q_2 ¢ w \$, \ldots, q_n ¢ w \$)$, where q_i denotes the initial state of M_i, $1 \leq i \leq n$, and each configuration that includes a component Accept is called an *accepting configuration*. A *computational step* of $\mathcal{M}$ is described by the binary relation $\vdash_{\mathcal{M}}$. Let $K = (\kappa_1, \kappa_2, \ldots, \kappa_n)$ and $K' = (\kappa'_1, \kappa'_2, \ldots, \kappa'_n)$ be two configurations. Then $K \vdash_{\mathcal{M}} K'$ if and only if, for all $i \in \{1, 2, \ldots, n\}$, one of the following conditions holds:

1. $\kappa_i \vdash_{M_i} \kappa'_i$ (a *local computation step*);
2. $\exists j \in \{1, 2, \ldots, n\} \setminus \{i\} : \kappa_i = u_i \mathsf{req}_{d_i}^j v_i, \quad \kappa_j = u_j \mathsf{res}_{d_j,c}^i v_j,$
   $\kappa'_i = u_i \mathsf{rec}_{d_i,c}^j v_i, \; \kappa'_j = u_j \mathsf{ack}_{d_j,c}^i v_j$
   (a *communication*);
3. $\exists j \in \{1, 2, \ldots, n\} \setminus \{i\} : \kappa_i = u_i \mathsf{res}_{d_i,c}^j v_i, \; \kappa_j = u_j \mathsf{req}_{d_j}^i v_j,$
   $\kappa'_i = u_i \mathsf{ack}_{d_i,c}^j v_i, \; \kappa'_j = u_j \mathsf{rec}_{d_j,c}^i v_j$
   (a *communication*);
4. $\kappa_i = \kappa'_i$, $\kappa_i \neq \mathsf{Accept}$, and no local operation (MVR, MVL, rewrite, or restart) or communication is possible for M_i.

Observe that the fourth item above leads to a non-synchronous behaviour of local computations, as a component can wait arbitrarily long for the execution of a communication. This implies in particular that a component does not know the exact number of local steps that its communication partner has executed since the last communication took place. Thus, the only synchronization between various components is by explicit communications.

In addition, a component is only allowed to keep its current configuration, if it either waits for a communication or if it is stuck. Whenever a local computation

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

step or a communication is possible, it is executed immediately. Thus, the only reason why the configuration of a system is not changed during a computation step is that all components are stuck or are in a communication deadlock.

The reflexive and transitive closure $\vdash_{\mathcal{M}}^*$ of the relation $\vdash_{\mathcal{M}}$ is the *computation relation* of $\mathcal{M}$, and the language accepted by a PCRA system $\mathcal{M}$ with input alphabet Σ is defined as

$$L(\mathcal{M}) = \{\, w \in \Sigma^* \mid (q_1 ¢ w \$, \ldots, q_n ¢ w \$) \vdash_{\mathcal{M}}^* (\kappa_1, \ldots, \kappa_n) \text{ and } \{\kappa_1, \ldots, \kappa_n\} \cap \{\mathsf{Accept}\} \neq \emptyset \,\}.$$

The class of languages accepted by PC-X-systems is denoted by $\mathcal{L}$(PC-X). A PCRA system is called *centralized*, if every communication involves the first component, which is then called the *master* of the system, that is, every communication takes place between the master and some other component. We use the prefix c to denote centralized PCRA systems, that is, a cPC-X-system is a centralized PC-X-system. In a centralized system, the superscripts of the communication states (that denote the communication partner) can be omitted (except for the master). The class of languages accepted by cPC-X-systems is denoted by $\mathcal{L}$(cPC-X).

A PCRA system is called *nondeterministic*, if at least one of its components is nondeterministic. If all components are deterministic, then the system is called *deterministic*, which is denoted by the prefix det.

We close this section with a detailed example of a PCRA system.

Example 2.1. Let $L_{\text{pal}} = \{w\#w^R \mid w \in \{a,b\}^+\}$ be the *marked mirror language*. This language is accepted by a PC-R-system $\mathcal{M} = (M_1, M_2)$ that consists of two deterministic components with window size one. System $\mathcal{M}$ behaves as follows, where the numbers in brackets correspond to the transitions used (see below): initially, M_1 moves its window two steps to the right, storing the first symbol to the right of the ¢-symbol within its internal control (1). The automaton M_2 moves its window initially one step to the right (14). Now, the window of M_1 is exactly one position further to the right as the window of M_2.

If the second symbol to the right of the ¢-symbol is the #-symbol, then M_1 checks whether the tape content is of the form $¢a\#a\$$ or $¢b\#b\$$. In the affirmative M_1 accepts (2−4). Otherwise, both windows move right stepwise and synchronously until M_1 reads the #-symbol (5−8 and 15−18). The window of M_2 is now exactly on the last symbol to the left of #.

Then, M_1 moves across the # (9), reads the first symbol of the second syllable, and sends it to M_2 (10). So M_1 and M_2 can compare the symbols positioned directly before and behind the #-symbol. If both symbols are different, then M_2 gets stuck, and M_1 cannot accept anymore. If both symbols are equal, then M_1 deletes the first symbol of the second syllable (11), applies a restart operation (12), moves its window again two steps to the right (1), and requests information from M_2. In parallel, M_2 sends a message to M_1 telling it to delete a symbol of the first syllable (19). Thus, M_1 deletes the current symbol from the tape (5, 13) and applies one more restart. Meanwhile M_2 deletes the symbol to the left of # and also

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

applies a restart operation (20, 21). Now the computation is repeated with the modified tape contents.

Observe that M_2 never reads or changes the second syllable, and the window of M_2 never moves across the #-symbol. On the other hand, M_1 not only works on the second syllable during the comparison, but it shortens the first syllable as well, so that the leftmost syllables of both, M_1 and M_2, have the same length. This is important for counting the number of MVR steps, so that the window of M_2 is positioned exactly one position before #. For that, M_1 deletes the second symbol of the first syllable after each comparison. Thus, the first symbol of the first syllable remains unchanged and can therefore be used in the last cycle, where M_1 compares this symbol with the last symbol of the second syllable.

Formally, M_1 and M_2 are defined as follows:

$M_1 = (\{q_0, q_1, q_2, q_a, q_{a2}, q_b, q_{b2}, q_e, \mathsf{req}, \mathsf{rec}_{\mathsf{mvr}}, \mathsf{res}_{\mathsf{mvr}}, \mathsf{ack}_{\mathsf{mvr}}, \mathsf{res}_a, \mathsf{res}_b, \mathsf{ack}_a, \mathsf{ack}_b, q_r, \mathsf{rec}_{\mathsf{del}}\}, \{a, b, \#\}, \{a, b, \#\}, ¢, \$, q_0, 1, \delta_1)$ with δ_1 defined as follows:

1) $\delta_1(q_0, ¢) = (q_0, \mathsf{MVR}), \delta_1(q_0, a) = (q_a, \mathsf{MVR}), \delta_1(q_0, b) = (q_b, \mathsf{MVR})$,
2) $\delta_1(q_a, \#) = (q_{a2}, \mathsf{MVR}), \delta_1(q_b, \#) = (q_{b2}, \mathsf{MVR})$,
3) $\delta_1(q_{a2}, a) = (q_e, \mathsf{MVR}), \delta_1(q_{b2}, b) = (q_e, \mathsf{MVR})$,
4) $\delta_1(q_e, \$) = \mathsf{Accept}$,
5) $\delta_1(q_a, a) = \delta_1(q_a, b) = \delta_1(q_b, a) = \delta_1(q_b, b) = \mathsf{req}$,
6) $\delta_1(\mathsf{rec}_{\mathsf{mvr}}, a) = \delta_1(\mathsf{rec}_{\mathsf{mvr}}, b) = (q_1, \mathsf{MVR})$,
7) $\delta_1(q_1, a) = \delta_1(q_1, b) = \delta_1(q_1, \#) = \mathsf{res}_{\mathsf{mvr}}$,
8) $\delta_1(\mathsf{ack}_{\mathsf{mvr}}, a) = \delta_1(\mathsf{ack}_{\mathsf{mvr}}, b) = (q_1, \mathsf{MVR})$,
9) $\delta_1(\mathsf{ack}_{\mathsf{mvr}}, \#) = (q_2, \mathsf{MVR})$,
10) $\delta_1(q_2, a) = \mathsf{res}_a, \delta_1(q_2, b) = \mathsf{res}_b$,
11) $\delta_1(\mathsf{ack}_a, a) = \delta_1(\mathsf{ack}_b, b) = (q_r, \varepsilon)$,
12) $\delta_1(q_r, a) = \delta_1(q_r, b) = \delta_1(q_r, \#) = \mathsf{Restart}$,
13) $\delta_1(\mathsf{rec}_{\mathsf{del}}, a) = \delta_1(\mathsf{rec}_{\mathsf{del}}, b) = (q_r, \varepsilon)$,

$M_2 = (\{q_0, q_1, q_r, \mathsf{res}_{\mathsf{mvr}}, \mathsf{ack}_{\mathsf{mvr}}, \mathsf{rec}_{\mathsf{mvr}}, \mathsf{req}, \mathsf{rec}_a, \mathsf{rec}_b, \mathsf{res}_{\mathsf{del}}, \mathsf{ack}_{\mathsf{del}}\}, \{a, b, \#\}, \{a, b, \#\}, ¢, \$, q_0, 1, \delta_2)$, where δ_2 is defined as follows:

14) $\delta_2(q_0, ¢) = (q_0, \mathsf{MVR})$,
15) $\delta_2(q_0, a) = \delta_2(q_0, b) = \mathsf{res}_{\mathsf{mvr}}$,
16) $\delta_2(\mathsf{ack}_{\mathsf{mvr}}, a) = \delta_2(\mathsf{ack}_{\mathsf{mvr}}, b) = \mathsf{req}$,
17) $\delta_2(\mathsf{rec}_{\mathsf{mvr}}, a) = \delta_2(\mathsf{rec}_{\mathsf{mvr}}, b) = (q_1, \mathsf{MVR})$,
18) $\delta_2(q_1, a) = \delta_2(q_1, b) = \mathsf{req}$,
19) $\delta_2(\mathsf{rec}_a, a) = \delta_2(\mathsf{rec}_b, b) = \mathsf{res}_{\mathsf{del}}$,
20) $\delta_2(\mathsf{ack}_{\mathsf{del}}, a) = \delta_2(\mathsf{ack}_{\mathsf{del}}, b) = (q_r, \varepsilon)$,
21) $\delta_2(q_r, a) = \delta_2(q_r, b) = \delta_2(q_r, \#) = \mathsf{Restart}$.

A computation of $\mathcal{M}$ for the input word $ab\#ba$ is shown in Figure 1.

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

$$
\begin{array}{lll|ll}
 & (q_0\¢ab\#ba\$, & q_0\¢ab\#ba\$) & \vdash_{\mathcal{M}} & (q_0\¢ab\#a\$, \quad ¢a\mathsf{res}_{\mathsf{del}}b\#ba\$) \\
\vdash_{\mathcal{M}} & (¢q_0ab\#ba\$, & ¢q_0ab\#ba\$) & \vdash_{\mathcal{M}} & (¢q_0ab\#a\$, \quad ¢a\mathsf{res}_{\mathsf{del}}b\#ba\$) \\
\vdash_{\mathcal{M}} & (¢aq_ab\#ba\$, & ¢\mathsf{res}_{\mathsf{mvr}}ab\#ba\$) & \vdash_{\mathcal{M}} & (¢aq_ab\#a\$, \quad ¢a\mathsf{res}_{\mathsf{del}}b\#ba\$) \\
\vdash_{\mathcal{M}} & (¢a\mathsf{req}b\#ba\$, & ¢\mathsf{res}_{\mathsf{mvr}}ab\#ba\$) & \vdash_{\mathcal{M}} & (¢a\mathsf{req}b\#a\$, \quad ¢a\mathsf{res}_{\mathsf{del}}b\#ba\$) \\
\vdash_{\mathcal{M}} & (¢a\mathsf{rec}_{\mathsf{mvr}}b\#ba\$, & ¢\mathsf{ack}_{\mathsf{mvr}}ab\#ba\$) & \vdash_{\mathcal{M}} & (¢a\mathsf{rec}_{\mathsf{del}}b\#a\$, \quad ¢a\mathsf{ack}_{\mathsf{del}}b\#ba\$) \\
\vdash_{\mathcal{M}} & (¢abq_1\#ba\$, & ¢\mathsf{req}ab\#ba\$) & \vdash_{\mathcal{M}} & (¢aq_r\#a\$, \quad ¢aq_r\#ba\$) \\
\vdash_{\mathcal{M}} & (¢ab\mathsf{res}_{\mathsf{mvr}}\#ba\$, & ¢\mathsf{req}ab\#ba\$) & \vdash_{\mathcal{M}} & (q_0\¢a\#a\$, \quad q_0\¢a\#ba\$) \\
\vdash_{\mathcal{M}} & (¢ab\mathsf{ack}_{\mathsf{mvr}}\#ba\$, & ¢\mathsf{rec}_{\mathsf{mvr}}ab\#ba\$) & \vdash_{\mathcal{M}} & (¢q_0a\#a\$, \quad ¢q_0a\#ba\$) \\
\vdash_{\mathcal{M}} & (¢ab\#q_2ba\$, & ¢aq_1b\#ba\$) & \vdash_{\mathcal{M}} & (¢aq_a\#a\$, \quad ¢\mathsf{res}_{\mathsf{mvr}}a\#ba\$) \\
\vdash_{\mathcal{M}} & (¢ab\#\mathsf{res}_b ba\$, & ¢a\mathsf{req}b\#ba\$) & \vdash_{\mathcal{M}} & (¢a\#q_{a2}a\$, \quad ¢\mathsf{res}_{\mathsf{mvr}}a\#ba\$) \\
\vdash_{\mathcal{M}} & (¢ab\#\mathsf{ack}_b ba\$, & ¢a\mathsf{rec}_b b\#ba\$) & \vdash_{\mathcal{M}} & (¢a\#aq_e\$, \quad ¢\mathsf{res}_{\mathsf{mvr}}a\#ba\$) \\
\vdash_{\mathcal{M}} & (¢ab\#q_ra\$, & ¢a\mathsf{res}_{\mathsf{del}}b\#ba\$) & \vdash_{\mathcal{M}} & (\mathsf{Accept}, \quad ¢\mathsf{res}_{\mathsf{mvr}}a\#ba\$).
\end{array}
$$

FIGURE 1. A computation of $\mathcal{M}$ for the input word $ab\#ba$.

3. CENTRALIZED *VERSUS* NON-CENTRALIZED PCRA SYSTEMS

As centralized PCRA systems are a special case of non-centralized PCRA systems, it is clear that non-centralized systems are at least as powerful as centralized systems. Therefore, it remains to determine whether the property of centralization actually decreases the expressive power of PCRA systems. For example, this phenomenon occurs in the setting of (synchronous) deterministic parallel communicating finite automata [1] and for PC grammar systems with regular or linear components [2]. Here we will show that in the case of PCRA systems, centralized systems have the same expressive power as non-centralized systems with the same type of components.

This result will be proved in two main steps. First, we establish it by introducing an additional component that serves as the master of the new system. Then, we present a construction that allows us to get rid of the additional component, in this way obtaining a centralized system of the same degree and with the same type of components as the given non-centralized system.

Proposition 3.1. *For any $\mathsf{X} \in \mathcal{T}$ and any $n \geq 2$, one can effectively construct a centralized PC-X-system $\mathcal{M}'$ of degree $n+1$ from a given PC-X-system $\mathcal{M}$ of degree n such that $L(\mathcal{M}') = L(\mathcal{M})$ holds.*

Proof. Let $\mathsf{X} \in \mathcal{T}$, and let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PC-X-system of degree n, where $M_i = (Q_i, \Sigma, \Gamma_i, ¢, \$, q_0^{(i)}, k, \delta_i)$ $(1 \leq i \leq n)$. From $\mathcal{M}$ we construct a centralized PC-X-system $\mathcal{M}' = (M, M_1', M_2', \ldots, M_n')$ such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$. Here, M is a new master component that controls all communications of $\mathcal{M}'$, that is, while $M_1, \ldots, M_n$ can communicate with each other, $M_1', \ldots, M_n'$ can only communicate with M. The main idea is that the master M cyclically asks the *clients* $M_1', \ldots, M_n'$ about their current situations. For this, it will be important to ensure that computation loops and communication deadlocks of the clients do not cause the master to get stuck.

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

In the following, we describe the behaviour of the components $M'_1, \ldots, M'_n$, and then we give the formal definition. Afterwards we do the same for the new master component. The bracketed annotations refer to the corresponding part of the definition below.

The clients are modified as follows: the operations MVR, Restart, and rewrite are retained unchanged (A1). Observe that there are exactly three possibilities of how a local computation of a component M_i can end: it may accept (A4), it may get stuck (A3), or it may reach a communication state (A5, A7). For these situations the component M'_i sends a corresponding message to the master by entering an appropriate response state: [accept], $[\bot]$, $[req(j)]$ or $[res(j,c)]$. After sending the response, the component is either stuck (if [accept] or $[\bot]$ was sent) or it enters a request state to await information from the master about the currently simulated communication step. The original communication states are included in the subscript (the local information) of the new communication states in order to enable $\mathcal{M}'$ to simulate the communication steps correctly.

By using MVL operations together with MVR operations, a client may get into an infinite loop during a local computation. This would lead to a deadlock of the system, when the master tries to communicate with that component. This is avoided as follows: whenever a component is about to perform a MVL operation, it sends the message [mvl] to the master, stores the successor state within the index of the communication state, and continues the local computation after this communication has been completed (A2).

Formally, a component M'_i $(1 \leq i \leq n)$ is obtained by modifying the transition relation of M_i as follows. Here $q, q' \in Q_i$, α is a possible content of the window, $j \in \{1, 2, \ldots, n\}$, and c and d are strings:

A1. For all $(q', \mathsf{MVR}) \in \delta_i(q, \alpha)$, $(q', \mathsf{MVR}) \in \delta'_i(q, \alpha)$ (analogously for Restart and rewrite operations).

A2. For all $(q', \mathsf{MVL}) \in \delta_i(q, \alpha)$,
- $\mathsf{res}_{[q'],[\mathsf{mvl}]} \in \delta'_i(q, \alpha)$ and
- $\delta'_i(\mathsf{ack}_{[q'],[\mathsf{mvl}]}, \alpha) = \{(q', \mathsf{MVL})\}$.

A3. For all $\delta_i(q, \alpha) = \emptyset$, $\delta'_i(q, \alpha) = \{\mathsf{res}_{[\bot]}\}$.

A4. For all $\mathsf{Accept} \in \delta_i(q, \alpha)$, $\mathsf{res}_{[\mathsf{accept}]} \in \delta'_i(q, \alpha)$.

A5. For $\mathsf{req}_d^j \in \delta_i(q, \alpha)$,
- $\mathsf{res}_{[req(j,d)],[req(j)]} \in \delta'_i(q, \alpha)$ and
- $\mathsf{req}_{[req(j,d)]} \in \delta'_i(\mathsf{ack}_{[req(j,d)],[req(j)]}, \alpha)$.

The local information d of the original communication state is stored within the local information of the new communication state, but it is not sent to the master. After sending the current communication state to the master, M'_i requests the information from the master of whether its communication partner has sent a corresponding communication state. If the communication partner does not do so, then M'_i is stuck just like M_i in this situation.

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

A6. For $A \in \delta_i(\mathsf{rec}^j_{d,c}, \alpha)$, $A \in \delta'_i(\mathsf{rec}_{[req(j,d)],[res(j,c)]}, \alpha)$.
The action A can be performed by M'_i, when the master tells it that the communication partner M'_j has sent the corresponding communication state.

A7. For $\mathsf{res}^j_{d,c} \in \delta_i(q, \alpha)$,
- $\mathsf{res}_{[res(j,d,c)],[res(j,c)]} \in \delta'_i(q, \alpha)$ and
- $\mathsf{req}_{[res(j,d,c)]} \in \delta'_i(\mathsf{ack}_{[res(j,d,c)],[res(j,c)]}, \alpha)$.

This case is similar to A5.

A8. For $A \in \delta_i(\mathsf{ack}^j_{d,c}, \alpha)$, $A \in \delta'_i(\mathsf{rec}_{[res(j,d,c)],[req(j)]}, \alpha)$.
This case is similar to A6.

The sets of states of the modified components are given indirectly by the definition of their transition relations. The initial state of M'_i is the same as that of M_i except in the special case that it is a communication state. If the initial state of M_i is req^j_d (or $\mathsf{res}^j_{d,c}$), then the initial state of M'_i is $\mathsf{res}_{[req(j,d)],[req(j)]}$ (or $\mathsf{res}_{[res(j,d,c)],[res(j,c)]}$).

The master component M has communication states only. In its index (as local information) each of these states contains a situation tuple that has a component for each of the automata M'_1 to M'_n. For example, the tuple $\langle *, \bot, res(1,c)\rangle$ describes the situation that the master does not (yet) know the current situation of M'_1 (that is, the master still has to ask for it), M'_2 is stuck, and M'_3 wants to send a response message to M'_1 with information c.

The initial state of the master is $\mathsf{req}^1_{\langle *,\ldots,*\rangle}$. Thus, initially the master does not have any information about the current situations of the other components, and M starts by asking M'_1 for the result of its local computation. Observe that the superscript of the communication state does indeed refer to the index of the corresponding component and not to its position within the system $\mathcal{M}'$. In the following the master asks all components for which its current situation tuple contains the symbol $*$ for the results of their local computations by contacting them one after the other in a cyclic manner. If the current situation tuple is $\langle t_1, t_2, \ldots, t_n\rangle$, and if M'_m is the last component that the master has communicated with, then the next component $M'_{m'}$ to be contacted is chosen as follows. If $N_1 = \{\, i \mid m < i \leq n \wedge t_i = * \,\}$ and $N_2 = \{\, i \mid 1 \leq i < m \wedge t_i = * \,\}$, then

$$m' = \begin{cases} \min(N_1), & \text{if } N_1 \neq \emptyset, \\ \min(N_2), & \text{if } N_1 = \emptyset \wedge N_2 \neq \emptyset. \end{cases}$$

If there is no $*$ left in the situation tuple, and hence, $N_1 = N_2 = \emptyset$, then the system $\mathcal{M}'$ is stuck, and therewith the current computation does not accept. In addition, observe that the successor component $M'_{m'}$ is obtained deterministically if it exists.

As in the centralized system $\mathcal{M}'$ each communication step must involve the master M, the communication steps between two components M_i and M_j of the original system $\mathcal{M}$ must be simulated by a sequence of communication steps between M'_i and the master M and between M'_j and the master M. This will be

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

done in such a way that the master controls and forwards all communication demands. The information about the original communication demand (component M_i enters state req_d^j or $\mathsf{res}_{c,d}^j$) is first stored within the situation tuple of the master (see B3 "otherwise" and B6 "otherwise" below). When the master receives a corresponding message from M'_j, it immediately informs both communication partners about the respective other message (see B3 "if" to B5 and B6 "if" to B8 below). After these communication steps have been performed, the symbol $*$ is stored in the new situation tuple for both communication partners, and then M continues with asking the next component (see above). Moreover, when receiving the message [mvl], the master just goes on with communicating with the (cyclic) successor component without changing the information about M'_m in its situation tuple (B9). Therefore, M'_m will be asked again after the master has communicated with all other components that are not yet stuck or waiting in a communication state. In between, M'_m continues performing local computation steps. Finally, if the master receives the information [accept] from some component, then it accepts itself, and therewith the system $\mathcal{M}'$ accepts (B1).

Below the formal definition of the transition function of the master M is given, where the transitions are defined for all $1 \leq m \leq n$:

B1. $\delta(\mathsf{rec}^m_{\langle t_1,\ldots,t_n\rangle,[\mathsf{accept}]},\alpha) = \mathsf{Accept}$,

B2. $\delta(\mathsf{rec}^m_{\langle t_1,\ldots,t_n\rangle,[\perp]},\alpha) = \mathsf{req}^{m'}_{\langle t_1,\ldots,t_{m-1},\perp,t_{m+1},\ldots,t_n\rangle}$,

B3. $\delta(\mathsf{rec}^m_{\langle t_1,\ldots,t_n\rangle,[req(j)]},\alpha) =$

$$\begin{cases} \mathsf{res}^m_{\langle t_1,\ldots,t_{m-1},req(j),t_{m+1},\ldots,t_{j-1},res(m,c),t_{j+1},\ldots,t_n\rangle,[res(j,c)]}, & \text{if } t_j = res(m,c) \\ \mathsf{req}^{m'}_{\langle t_1,\ldots,t_{m-1},req(j),t_{m+1},\ldots,t_n\rangle}, & \text{otherwise,} \end{cases}$$

B4. $\delta(\mathsf{ack}^m_{\langle t_1,\ldots,t_{m-1},req(j),t_{m+1},\ldots,t_{j-1},res(m,c),t_{j+1},\ldots,t_n\rangle,[res(j,c)]},\alpha) =$
$\mathsf{res}^j_{\langle t_1,\ldots,t_{m-1},*,t_{m+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle,[req(m)]}$,

B5. $\delta(\mathsf{ack}^j_{\langle t_1,\ldots,t_{m-1},*,t_{m+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle,[req(m)]},\alpha) =$
$\mathsf{req}^{m'}_{\langle t_1,\ldots,t_{m-1},*,t_{m+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle}$,

B6. $\delta(\mathsf{rec}^m_{\langle t_1,\ldots,t_n\rangle,[res(j,c)]},\alpha) =$

$$\begin{cases} \mathsf{res}^m_{\langle t_1,\ldots,t_{m-1},res(j,c),t_{m+1},\ldots,t_{j-1},req(m),t_{j+1},\ldots,t_n\rangle,[req(j)]} & \text{if } t_j = req(m), \\ \mathsf{req}^{m'}_{\langle t_1,\ldots,t_{m-1},res(j,c),t_{m+1},\ldots,t_n\rangle} & \text{otherwise,} \end{cases}$$

B7. $\delta(\mathsf{ack}^m_{\langle t_1,\ldots,t_{m-1},res(j,c),t_{m+1},\ldots,t_{j-1},req(m),t_{j+1},\ldots,t_n\rangle,[req(j)]},\alpha) =$
$\mathsf{res}^j_{\langle t_1,\ldots,t_{m-1},*,t_{m+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle,[res(m,c)]}$,

B8. $\delta(\mathsf{ack}^j_{\langle t_1,\ldots,t_{m-1},*,t_{m+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle,[res(m,c)]},\alpha) =$
$\mathsf{req}^{m'}_{\langle t_1,\ldots,t_{m-1},*,t_{m+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle}$.

B9. $\delta(\mathsf{rec}^m_{\langle t_1,\ldots,t_{m-1},*,t_{m+1},\ldots,t_n\rangle,[\mathsf{mvl}]},\alpha) = \{\mathsf{req}^{m'}_{\langle t_1,\ldots,t_{m-1},*,t_{m+1},\ldots,t_n\rangle}\}$.

Finally, whenever a component M_i reaches the accepting configuration, then M'_i enters the state $\mathsf{res}_{[\mathsf{accept}]}$, and M and therewith $\mathcal{M}'$ accepts the input. In particular, even if one component (or more) is in an infinite loop of computation, M continues

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

with asking all components cyclically. On the other hand, if none of the components reaches the accepting configuration, then none of the components M_1' to M_n' can enter the state $\mathsf{res}_{[\mathsf{accept}]}$, and thus, M cannot reach the accepting configuration, and $\mathcal{M}'$ does not accept the input. Hence, $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$, which completes the proof. □

In the construction presented in the proof of Propositions 3.1, the master component is introduced as an additional component of the centralized PCRA system. Thus, the simulating centralized PCRA system $\mathcal{M}'$ has more components than the non-centralized PCRA system $\mathcal{M}$ being simulated. However, we can actually get rid of this extra component. Observe that the master component in the above construction does not perform any local computation steps at all, that is, it does not access its tape at all. Hence, we can combine the master with component M_1' into a new component $\hat{M}_1$ that is essentially the direct product of M and M_1'. However, the situation is complicated by the fact that M_1' may perform restart steps, during which its internal state is reset to its initial state. Accordingly, also $\hat{M}_1$ will have to perform restart steps, causing it to lose all information about the situation tuple that is currently stored by its subcomponent M. To overcome this problem, we need another component, say M_2', that stores the information about the situation tuple immediately *before* $\hat{M}_1$ performs a restart step, so that $\hat{M}_1$ can restore this information immediately *after* completing a restart step by requesting it from M_2'. Based on this idea, the following result can be derived.

Proposition 3.2. *For any $\mathsf{X} \in \mathcal{T}$ and any $n \geq 2$, one can effectively construct a centralized PC-X-system $\mathcal{M}'$ of degree n from a given PC-X-system $\mathcal{M}$ of degree n such that $L(\mathcal{M}') = L(\mathcal{M})$ holds.*

Proof. For simplicity transitions of the form $q \in \delta(p, \alpha)$ are used in this proof, such that a component can change its internal state without performing any operation on the tape. It can be easily shown by successive elimination of these transitions that each system that uses this kind of transitions can be transformed into an equivalent system without such transitions.

Now, let $\mathcal{M} = (M, M_1, M_2, \ldots, M_n)$ be the centralized system that was constructed in the proof of Proposition 3.1. Since the master component M only performs communication steps independently of its tape content, we merge M and M_1 to a new component $\hat{M}_1$ in the first step of this proof. However, in order to ensure that $\hat{M}_1$ does not lose the information on the actual state of M when performing a restart, we use annotated restart states for $\hat{M}_1$, that is, we take $\hat{M}_1$ to be a *non-forgetting* X-automaton (see, *e.g.*, [8]). This annotation will be eliminated in the second part of the proof.

Each computation of $\hat{M}_1$ starts with the new initial state $(\mathsf{req}^1_{\langle *,\ldots,* \rangle}, q_0^{(1)})$, which combines the original initial states of M and M_1. Now, $\hat{M}_1$ behaves as follows. Whenever M wants to communicate with M_1 (as for the initial state) and the latter is still not in a communication state, $\hat{M}_1$ keeps the current state of M and simulates the computation of M_1 until M_1 reaches a corresponding communication

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

state:

$$\begin{aligned}
\hat{\delta}_1((\mathsf{req}^1_d, q), \alpha) &= \{((\mathsf{req}^1_d, q'), \mathsf{MVR}) \mid (q', \mathsf{MVR}) \in \delta_1(q, \alpha)\} \\
&\quad \cup \{((\mathsf{req}^1_d, q'), \mathsf{MVL}) \mid (q', \mathsf{MVL}) \in \delta_1(q, \alpha)\} \\
&\quad \cup \{((\mathsf{req}^1_d, q'), \beta) \mid (q', \beta) \in \delta_1(q, \alpha)\} \\
&\quad \cup \{(\mathsf{Restart}, (\mathsf{req}^1_d, q_0^{(1)})) \mid \mathsf{Restart} \in \delta_1(q, \alpha)\}, \\
\hat{\delta}_1((\mathsf{res}^1_{d,c}, q), \alpha) &= \{((\mathsf{res}^1_{d,c}, q'), \mathsf{MVR}) \mid (q', \mathsf{MVR}) \in \delta_1(q, \alpha)\} \\
&\quad \cup \{((\mathsf{res}^1_{d,c}, q'), \mathsf{MVL}) \mid (q', \mathsf{MVL}) \in \delta_1(q, \alpha)\} \\
&\quad \cup \{((\mathsf{res}^1_{d,c}, q'), \beta) \mid (q', \beta) \in \delta_1(q, \alpha)\} \\
&\quad \cup \{(\mathsf{Restart}, (\mathsf{res}^1_{d,c}, q_0^{(1)})) \mid \mathsf{Restart} \in \delta_1(q, \alpha)\}.
\end{aligned}$$

As mentioned before, the restart operation is annotated by a state. If M_1 enters a corresponding communication state, then the communication step between M and M_1 is resolved by $\hat{M}_1$:

$$\begin{aligned}
\hat{\delta}_1((\mathsf{req}^1_d, \mathsf{res}_{d',c}), \alpha) &= \{(\mathsf{rec}^1_{d,c}, \mathsf{ack}_{d',c})\}, \\
\hat{\delta}_1((\mathsf{res}^1_{d,c}, \mathsf{req}_{d'}), \alpha) &= \{(\mathsf{ack}^1_{d,c}, \mathsf{rec}_{d',c})\}.
\end{aligned}$$

If M_1 does not reach a corresponding communication state, then $\hat{M}_1$ is blocked in the system $\mathcal{M}'$ just as the master M is in this situation. Thereafter, the next communication step of M is initiated immediately. For communications between M and M_1 we therefore define:

$$\begin{aligned}
\hat{\delta}_1((\mathsf{rec}^1_{d,c}, \mathsf{ack}_{d',c}), \alpha) &= \{(p, \mathsf{ack}_{d',c}) \mid p \in \delta(\mathsf{rec}^1_{d,c}, \alpha)\}, \\
\hat{\delta}_1((\mathsf{ack}^1_{d,c}, \mathsf{rec}_{d',c}), \alpha) &= \{(p, \mathsf{rec}_{d',c}) \mid p \in \delta(\mathsf{ack}^1_{d,c}, \alpha)\}.
\end{aligned}$$

For communications between M and any other component, $\hat{M}_1$ enters the according communication state and, moreover, keeps the current internal state of M_1 within the local information of its communication state:

$$\begin{aligned}
\hat{\delta}_1((\mathsf{req}^i_d, q), \alpha) &= \{\mathsf{req}^i_{[d,q]}\}, \\
\hat{\delta}_1(\mathsf{rec}^i_{[d,q],c}, \alpha) &= \{(p, q) \mid p \in \delta_1(\mathsf{rec}^i_{d,c}, \alpha)\}, \\
\hat{\delta}_1((\mathsf{res}^i_{d,c}, q), \alpha) &= \{\mathsf{res}^i_{[d,q],c}\}, \\
\hat{\delta}_1(\mathsf{ack}^i_{[d,q],c}, \alpha) &= \{(p, q) \mid p \in \delta_1(\mathsf{ack}^i_{d,c}, \alpha)\},
\end{aligned}$$

for $i = 2, \ldots, n$.

Above, we used annotations for the restart operations of $\hat{M}_1$ in order to not lose its current internal state during the restarts. Now, in the second part of the proof, we construct a system $\mathcal{M}' = (\hat{M}'_1, M'_2, M_3, \ldots, M_n)$, where we modify $\hat{M}_1$ and M_2 in such a way that this annotation is not necessary anymore. The basic idea is that $\hat{M}'_1$ controls the computation of M'_2 in the following sense: M'_2 always requests instructions from $\hat{M}'_1$. There are only two different situations, in which M'_2 has to execute computation steps: either $\hat{M}'_1$ needs M'_2 for storing its current state during a restart operation, or $\hat{M}'_1$ wants to simulate a communication between $\hat{M}_1$ and M_2. In the second case it is important that M'_2 simulates the local

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

computation of M_2 in order to be able to enter a corresponding communication state. Observe that M_2 is not allowed to communicate with any other component since $\mathcal{M}$ is centralized. To synchronize $\hat{M}_1'$ and M_2' their new initial states are req^2_R and $\mathsf{res}_{[q_0^{(2)}],[(\mathsf{req}^1_{\langle *,\ldots,*\rangle},q_0^{(1)})]}$. This shows that $\hat{M}_1'$ requests its last entered state after each restart operation ("R" for restart), and M_2' initially sends the original initial state of $\hat{M}_1$ to $\hat{M}_1'$. Together with the definition of

$$p \in \hat{\delta}_1'(\mathsf{rec}^2_{R,[p]},\alpha) \text{ and } \mathsf{req}_{[q]} \in \delta_2'(\mathsf{ack}_{[q],[(\mathsf{req}^1_{\langle *,\ldots,*\rangle},q_0^{(1)})]},\beta)$$

for all $p \in \hat{Q}_1$, $q \in Q_2$, and all possible window contents α and β, the beginning of each computation leads to the configuration $((\mathsf{req}^1_{\langle *,\ldots,*\rangle},q_0^{(1)})\mathbin{¢}w\$, \mathsf{req}_{[q_0^{(2)}]}\mathbin{¢}w\$, q_0^{(3)}\mathbin{¢}w\$,\ldots,q_0^{(n)}\mathbin{¢}w\$)$ for an input word w.

Each annotated transition of the form $(\mathsf{Restart},r) \in \hat{\delta}_1(p,\alpha)$ is replaced by:

$$\begin{aligned} \mathsf{res}^2_{R,[r]} &\in \hat{\delta}_1'(p,\alpha), & \mathsf{res}_{[q],[r]} &\in \delta_2'(\mathsf{rec}_{[q],[r]},\beta),\\ \mathsf{Restart} &\in \hat{\delta}_1'(\mathsf{ack}^2_{R,[r]},\alpha), & \mathsf{req}_{[q]} &\in \delta_2'(\mathsf{ack}_{[q],[r]},\beta), \end{aligned}$$

for all $q \in Q_2$ and all window contents β. Whenever $\hat{M}_1$ executes a restart operation, then $\hat{M}_1'$ sends the annotation to M_2', executes the restart, and requests the state back from M_2'. Meanwhile, M_2' only receives the state from $\hat{M}_1'$ and sends it back immediately.

To start the local computation of M_2' and simulate a communication between $\hat{M}_1$ and M_2, the transition $q \in \delta_2'(\mathsf{rec}_{[q],L},\beta)$ is added for all $q \in Q_2$ and all possible window contents β, where the message "L" stands for "local computation".

Whenever $\hat{M}_1$ enters a communication state that initiates a communication with M_2, then $\hat{M}_1'$ sends the message "L" to M_2', M_2' simulates the local computation of M_2 until it enters a corresponding communication state, and finally the communication is resolved. A special situation occurs when M_2 performs a restart for itself. In this case, M_2' sends the initial state to $\hat{M}_1'$, $\hat{M}_1'$ ignores this and replies to M_2' with the message "L", and M_2' continues its local computation while $\hat{M}_1'$ waits for the communication answer again. Formally, all transitions of the form $\mathsf{req}^2_d \in \hat{\delta}_1(p,\alpha)$ of $\hat{M}_1$ and $\mathsf{res}_{d',c} \in \delta_2(q,\beta)$ of M_2 are replaced by:

$$\begin{aligned} \mathsf{res}^2_{req(d),L} &\in \hat{\delta}_1'(p,\alpha),\\ \mathsf{req}^2_{req(d)} &\in \hat{\delta}_1'(\mathsf{ack}^2_{req(d),L},\alpha),\\ \mathsf{rec}^2_{d,c} &\in \hat{\delta}_1'(\mathsf{rec}^2_{req(d),res(c)},\alpha),\\ \mathsf{res}^2_{req(d),L} &\in \hat{\delta}_1'(\mathsf{rec}^2_{req(d),[\mathsf{res}_{[q_0^{(2)}],[(\mathsf{req}^1_{\langle *,\ldots,*\rangle},q_0^{(1)})]}]},\alpha),\\ \mathsf{res}_{[ack_{d',c}],res(c)} &\in \delta_2'(q,\beta),\\ \mathsf{req}_{[ack_{d',c}]} &\in \delta_2'(\mathsf{ack}_{[ack_{d',c}],res(c)},\beta). \end{aligned}$$

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

Moreover, replace $\mathsf{res}^2_{d,c} \in \hat{\delta}_1(p,\alpha)$ and $\mathsf{req}_{d'} \in \delta_2(q,\beta)$ by:

$$\begin{aligned}
\mathsf{res}^2_{res(d,c),L} &\in \hat{\delta}'_1(p,\alpha),\\
\mathsf{req}^2_{res(d,c)} &\in \hat{\delta}'_1(\mathsf{ack}^2_{res(d,c),L},\alpha),\\
\mathsf{res}^2_{d,c} &\in \hat{\delta}'_1(\mathsf{rec}^2_{res(d,c),req},\alpha),\\
\mathsf{res}^2_{res(d,c),L} &\in \hat{\delta}'_1(\mathsf{rec}^2_{res(d,c),[\mathsf{res}_{[q_0^{(2)}],[(\mathsf{req}^1_{\langle *,\ldots,*\rangle},q_0^{(1)})]}]},\alpha),\\
\mathsf{res}_{rec(d'),req} &\in \delta'_2(q,\beta),\\
\mathsf{req}_{rec(d')} &\in \delta'_2(\mathsf{ack}_{rec(d'),req},\beta),\\
\mathsf{req}_{[rec_{d',c}]} &\in \delta'_2(\mathsf{rec}_{rec(d'),c},\beta).
\end{aligned}$$

In summary, if any of the components of the non-centralized system reaches the accepting configuration, then component M of system $\mathcal{M}$ accepts the input. Then, also $\hat{M}_1$, $\hat{M}'_1$, and therefore $\mathcal{M}'$ accept due to the above construction. On the other hand, if none of the components of $\mathcal{M}$ accepts the input, then also none of the components of $\mathcal{M}'$ can reach the accepting configuration. Hence, $L(\mathcal{M}') = L(\mathcal{M})$. □

The PCRA system that is obtained from the proofs of Propositions 3.1 and 3.2 is not only centralized, but it also has the property that it accepts if and only if its first component accepts. Hence, we can summarize the results presented in this section as follows.

Theorem 3.3. *For all $X \in \mathcal{T}$ and all $n \geq 2$, $\mathcal{L}(\mathsf{cPC\text{-}X}(n)) = \mathcal{L}(\mathsf{PC\text{-}X}(n))$ and, moreover, one can effectively construct a centralized PC-X-system $\mathcal{M}'$ of degree n from a given PC-X-system $\mathcal{M}$ of degree n such that $L(\mathcal{M}') = L(\mathcal{M})$, and $\mathcal{M}'$ accepts with its first component.*

4. Comparison to restarting automata

From the definition of acceptance by PCRA systems, the following closure properties are easily obtained.

Proposition 4.1. *For all $X \in \mathcal{T}$, the classes $\mathcal{L}(\mathsf{PC\text{-}X})$ and $\mathcal{L}(\mathsf{det\text{-}PC\text{-}X})$ are closed under union and intersection.*

As $\mathsf{DCFL} \subseteq \mathcal{L}(\mathsf{det\text{-}R})$ (see [6]) this yields the following results.

Proposition 4.2.

(a) *Every finite union of deterministic context-free languages is accepted by a det-PC-R-system that does not use any communication steps.*
(b) *Every finite intersection of deterministic context-free languages is accepted by a det-PC-R-system that only uses a constant number of communication steps. In fact, the number of communication steps used is at most the degree of the system minus one.*

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

Proof. (b) Let $L = \bigcap_{i=1}^{m} D_i$, where $D_1, \ldots, D_m$ are deterministic context-free languages, and for each $i = 1, \ldots, m$, let M_i be a det-R-automaton that accepts D_i. Then we form the system $\mathcal{M} = (M_1', \ldots, M_m')$ that proceeds as follows. On input w, each of the components M_i' checks whether w belongs to the language D_i. For $i \geq 2$, if $w \in D_i$, then M_i' sends a corresponding message to M_1', which accepts if $w \in D_1$ and if it has obtained corresponding messages from all components M_2' to M_m'. □

Thus, we see that the language $L = \{ a^n b^n \mid n \geq 0 \} \cup \{ a^n b^m \mid m > 2n \geq 0 \}$ is accepted by a det-PC-R-system with two components (even without communication). But L cannot be accepted by any one-way restarting automaton without auxiliary symbols (see [13]). Hence, we obtain the following result.

Proposition 4.3. *For all* $X \in \{R, RR, RW, RRW\}$,

$$\mathcal{L}(\textit{det-X}) \subset \mathcal{L}(\textit{det-PC-X}(2)) \text{ and } \mathcal{L}(X) \subset \mathcal{L}(\textit{PC-X}(2)).$$

For restarting automata with auxiliary symbols, that is, for automata of type RWW, RRWW, RLWW, det-RWW, det-RRWW, or det-RLWW, we have the following results. Recall that RLW- (RRW-, RW-) automata are just RLWW- (RRWW-, RWW-) automata that cannot use auxiliary symbols.

Proposition 4.4.

(a) $\mathcal{L}(\textit{RLWW}) \subseteq \mathcal{L}(\textit{PC-RLW}(2))$, (d) $\mathcal{L}(\textit{det-RLWW}) \subseteq \mathcal{L}(\textit{det-PC-RLW}(2))$,
(b) $\mathcal{L}(\textit{RRWW}) \subseteq \mathcal{L}(\textit{PC-RRW}(2))$, (e) $\mathcal{L}(\textit{det-RRWW}) \subseteq \mathcal{L}(\textit{det-PC-RRW}(2))$,
(c) $\mathcal{L}(\textit{RWW}) \subseteq \mathcal{L}(\textit{PC-RW}(2))$, (f) $\mathcal{L}(\textit{det-RWW}) \subseteq \mathcal{L}(\textit{det-PC-RW}(2))$.

Proof. Assume that L is accepted by a (deterministic) R(R)WW-automaton. Hence, as shown in [12], there exist a (deterministic) R(R)W-automaton M and a regular language R such that $L = L(M) \cap R$. The proof of that result can also be applied to RLWW-automata [13]. Moreover, the class of regular languages is a subset of $\mathcal{L}$(det-R). Hence, it follows from Proposition 4.1 that L is accepted by a (deterministic) PC-R(R)W-system, PC-RLW-system respectively, of degree 2. □

5. Comparison to PC systems of finite automata

To obtain a lower bound for the computational power of PCRA systems, we consider the weakest systems, that is, PC systems of R-automata with window size one (PC-R$(n, 1)$-systems). Restarting automata of type R with window size one are known to characterize the regular languages [9]. Thus, the question is natural as to whether PC-R$(n, 1)$-systems have the same expressive power as PC systems of finite automata (PCFA systems, for short). We will see that, although the components of both types of PC systems accept the same class of languages, that is, the regular languages, PC-R$(n, 1)$-systems are more powerful than PCFA systems of degree n.

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

It is known that PCFA systems correspond to one-way multi-head finite automata [7], and that (deterministic) two-way multi-head finite automata characterize the complexity classes L and NL, that is, the class of languages that are accepted by deterministic, respectively by nondeterministic, Turing machines with a logarithmic space bound [5]. Here we give an informal definition of multi-head finite automata and then use this model to obtain our simulation results.

A *nondeterministic two-way n-head finite automaton* A (a 2-NFA(n), for short) consists of a finite control and an input tape with n read-only heads. Initially, A is in the initial state, the input word is placed on the tape enclosed by a left and a right end marker, and all heads are set on the leftmost position of the tape, that is, on the left end marker. Depending on the current state that is stored within the finite control and the currently read symbols of all heads, A changes the state and moves the heads one step to the right, one step to the left, or keeps their positions. An input word w is accepted by A if and only if there exists a computation of A that starts with the initial configuration on input w and that halts in a final state after a finite number of computational steps. The language accepted by A is denoted by $L(A)$, which consists of all accepted words.

If there exists at most one possible computation step in each situation, then we say that A is a *deterministic n-head finite automaton*, 2-DFA(n) for short. Moreover, if no head movements to the left are possible, then A is a *one-way n-head finite automaton*, denoted as 1-DFA(n) in the deterministic case or 1-NFA(n) in the nondeterministic case. The class of all languages that are accepted by n-head finite automata of type $\mathsf{X} \in \{\text{2-NFA}, \text{2-DFA}, \text{1-NFA}, \text{1-DFA}\}$ is denoted by $\mathcal{L}(\mathsf{X}(n))$, and the class of languages that are accepted by multi-head finite automata with any finite number of heads is denoted by $\mathcal{L}(\mathsf{X}) = \bigcup_{n \geq 1} \mathcal{L}(\mathsf{X}(n))$.

Proposition 5.1. *For all $n \geq 2$,*

(a) $\mathcal{L}(\mathit{DPCFA}(n)) = \mathcal{L}(\mathit{1\text{-}DFA}(n)) \subset \mathcal{L}(\mathit{det\text{-}PC\text{-}R}(n,1))$.
(b) $\mathcal{L}(\mathit{PCFA}(n)) = \mathcal{L}(\mathit{1\text{-}NFA}(n)) \subset \mathcal{L}(\mathit{PC\text{-}R}(n,1))$.
(c) $\mathcal{L}(\mathit{2\text{-}DFA}(n)) \subseteq \mathcal{L}(\mathit{det\text{-}PC\text{-}RL}(n,1))$.
(d) $\mathcal{L}(\mathit{2\text{-}NFA}(n)) \subseteq \mathcal{L}(\mathit{PC\text{-}RL}(n,1))$.

Proof. The equalities were shown in [7]. The inclusions can be shown by direct simulation. Let A be a nondeterministic two-way n-head finite automaton. A PCRA system $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ of type PC-RL$(n,1)$ can simulate A as follows. The first component simulates the first head of A and determines the transitions to be simulated, while the components $M_2, \ldots, M_n$ are used to simulate the other $n-1$ heads of A. For doing so, M_1 asks M_2 to M_n for the symbols they currently read, then it determines the transition of A to be simulated, and sends M_2 to M_n the information about their head movements. The PCRA system $\mathcal{M}$ is deterministic if A is. Moreover, $\mathcal{M}$ consists of one-way components only, if A is a one-way multi-head finite automaton. In the one-way case, the inclusions are proper, as the marked mirror language L_{pal} is not accepted by any one-way multi-head finite automaton, but it is accepted by some det-PC-R$(2,1)$-system (see Example 2.1). For details concerning this simulation see the proceedings of NCMA 2012 [17]. □

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

From Proposition 5.1 we obtain the following consequences.

Corollary 5.2. (a) $\mathsf{L} = \mathcal{L}(\mathit{2\text{-}DFA}) \subseteq \mathcal{L}(\mathit{det\text{-}PC\text{-}RL})$.
(b) $\mathsf{NL} = \mathcal{L}(\mathit{2\text{-}NFA}) \subseteq \mathcal{L}(\mathit{PC\text{-}RL})$.

From the linear space bound for restarting automata and PCRA systems, we obtain the following upper bound for the expressive power of PCRA systems.

Proposition 5.3. $\mathcal{L}(\mathit{PC\text{-}RLWW}) \subseteq \mathit{CSL}$.

6. Conclusion

We have introduced PC systems of restarting automata. Our main result shows that, for all types X of restarting automata, centralized PCRA systems of type X and degree $n \geq 2$ can simulate non-centralized PCRA systems of type X and degree n. Thus, for these kinds of PC systems, the property of being centralized is not a restriction. In the case of one-way restarting automata without auxiliary symbols, we have seen that systems with two components are already more expressive than restarting automata of the same type. Further, restarting automata with auxiliary symbols can be simulated by PCRA systems without auxiliary symbols with only two components. Moreover, we have established a lower bound for the computational power of PCRA systems with one-way components through the class of languages that are accepted by PC systems of finite automata. For PCRA systems with two-way components, the complexity classes L and NL are lower bounds in the deterministic and in the nondeterministic case, respectively, and the class CSL is clearly an upper bound for all types of PCRA systems. However, it remains open whether CSL is a proper superset of the language classes accepted by PCRA systems. Also it remains open whether the most general PCRA systems, that is, the PC-RRWW- and PC-RLWW-systems, are more powerful than RRWW- or RLWW-automata. It is known that these two types of restarting automata accept the same class of languages. Does a corresponding result also hold for the PCRA systems of these types? As even the weakest types of PCRA systems, those with deterministic R-automata as components, accept quite complicated languages (*e.g.*, the copy language, the exponential language, languages with crossed dependencies), we could not yet find any (context-sensitive) language that cannot be accepted by any PCRA system. In this context, it also remains the problem of whether the language classes accepted by PCRA systems of type X form a strict hierarchy based on the number of components.

References

[1] H. Bordihn, M. Kutrib and A. Malcher, On the computational capacity of parallel communicating finite automata, in *Developments in Language Theory*. In vol. 5257 of *Lect. Notes Comput. Sci.*, edited by M. Ito and M. Toyama. Springer, Berlin (2008) 146–157.

[2] E. Csuhaj-Varjú, J. Dassow, J. Kelemen and G. Păun, *Grammar Systems: A Grammatical Approach to Distribution and Cooperation*. Gordon and Breach Science Publishers, Newark, NJ, USA (1994).

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press

[3] E. Csuhaj-Varjú, C. Martín-Vide, V. Mitrana and G. Vaszil, Parallel communicating pushdown automata systems. *Int. J. Found. Comput. Sci.* **11** (2000) 633–650.
[4] M. Goehring, PC-systems of restarting automata, in *Theorietag Automaten und Formale Sprachen 2009*, Technical Report 2009/03, edited by J. Mielke, L. Staiger and R. Winter. Universität Halle-Wittenberg, Institut für Informatik (2009) 26–27.
[5] J. Hartmanis, On non-determinancy in simple computing devices. *Acta Informatica* **1** (1972) 336–344.
[6] P. Jančar, F. Mráz, M. Plátek and J. Vogel, Restarting automata, *FCT'95*, in vol. 965 of *Lect. Notes Comput. Sci.*, edited by H. Reichel. Springer, Berlin (1995) 283–292.
[7] C. Martín-Vide, A. Mateescu and V. Mitrana, Parallel finite automata systems communicating by states. *Int. J. Found. Comput. Sci.* **13** (2002) 733–749.
[8] H. Messerschmidt and F. Otto, Cooperating distributed systems of restarting automata. *Int. J. Found. Comput. Sci.* **18** (2007) 1333–1342.
[9] F. Mráz, Lookahead hierarchies of restarting automata. *J. Automata, Languages and Combinatorics* **6** (2001) 493–506.
[10] B. Nagy and F. Otto, CD-systems of stateless deterministic R(1)-automata governed by an external pushdown store. *RAIRO: ITA* **45** (2011) 413–448.
[11] B. Nagy and F. Otto, On CD-systems of stateless deterministic R-automata with window size one. *J. Comput. Syst. Sci.* **78** (2012) 780–806.
[12] G. Niemann and F. Otto, Further results on restarting automata, in *Words, Languages and Combinatorics*, edited by M. Ito and T. Imaoka. World Scientific, Singapore (2003) 352–369.
[13] F. Otto, Restarting automata, in *Recent Advances in Formal Languages and Applications*, vol. 25 of *Studies in Computational Intelligence*, edited by Z. Ésik, C. Martín-Vide and V. Mitrana. Springer, Berlin (2006) 269–303.
[14] F. Otto, Asynchronous PC Systems of Pushdown Automata, in *Language and Automata Theory and Applications*. In vol. 7810 of *Lect. Notes Comput. Sci.*, edited by A.-H. Dediu, C. Martín-Vide and B. Truthe. Springer Berlin/Heidelberg (2013) 456–467.
[15] G. Păun and L. Santean, Parallel communicating grammar systems: the regular case. *Analele Universitatii din Bucuresti, Seria matematica-informatica* **2** (1989) 55–63.
[16] M. Vollweiler, Asynchronous systems of parallel communicating finite automata, in vol. 294 of *5th Workshop on Non-Classical Models of Automata and Appl.*, edited by S. Bensch, F. Drewes, R. Freund and F. Otto. *books@acg.at*, Österreichische Computer Gesellschaft, Wien (2013) 243–257.
[17] M. Vollweiler and F. Otto, Systems of parallel communicating restarting automata, in vol. 290 of *4th Workshop on Non-Classical Models of Automata and Applications*, edited by R. Freund, M. Holzer, B. Truthe and U. Ultes-Nitsche. *books@acg.at*, Österreichische Computer Gesellschaft, Wien (2012) 197–212.

Communicated by M. Holzer.
Received January 23, 2013. Accepted December 11, 2013.

https://doi.org/10.1051/ita/2013046 Published online by Cambridge University Press