Systems of Parallel Communicating Restarting Automata

Dissertation
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
im Fachbereich Elektrotechnik / Informatik der Universität Kassel

Marcel Vollweiler

Kassel im Januar 2013

Erster Gutachter:	Prof. Dr. Friedrich Otto
Zweiter Gutachter:	Prof. Dr. Jürgen Dassow
Tag der Disputation:	3. Juli 2013

Zusammenfassung

In der vorliegenden Dissertation werden Systeme von parallel arbeitenden und miteinander kommunizierenden Restart-Automaten (engl.: *parallel communicating restarting automata systems*; abgekürzt *PCRA-Systeme*) vorgestellt und untersucht. Dabei werden zwei bekannte Konzepte aus den Bereichen Formale Sprachen und Automatentheorie miteinander verknüpft: das Modell des *Restart-Automaten* und die sogenannten PC-Systeme (*systems of parallel communicating components*).

Der Restart-Automat wurde entwickelt, um die linguistische Technik *Analyse durch Reduktion* zu modellieren. Diese wird unter anderem dazu verwendet, die syntaktische Korrektheit von Sätzen natürlicher Sprachen zu überprüfen. Bisher wurden viele verschiedene Varianten des Restart-Automaten definiert und untersucht, wobei sich unterschiedliche Eigenschaften und diverse Parallelen zu anderen Aspekten aus der Theorie der Formalen Sprachen gezeigt haben. Die Zusammenarbeit von Restart-Automaten wurde erstmalig anhand der CD-Systeme von Restart-Automaten (*cooperating distributed restarting automata systems*) betrachtet, in denen die Komponenten sequentiell an einer gemeinsamen Zeichenkette arbeiten.

Das Konzept der PC-Systeme beruht auf dem *classroom model* als eine von Menschen angewandte Problemlösungsstrategie, in der mehrere Experten gemeinsam und parallel an der Lösung eines gegebenen Problems arbeiten und dabei Information via Kommunikation austauschen können. Dieser Ansatz wurde bereits auf verschiedene Automaten- und Grammatikmodelle übertragen, z. B. auf endliche Automaten, Kellerautomaten, Watson-Crick-Automaten und Chomsky-Grammatiken unterschiedlichen Typs. In den meisten Fällen waren PC-Systeme trotz einer sehr einfachen Kommunikationsstruktur deutlich ausdrucksstärker als eine einzelne Komponente für sich genommen. Im übertragenen Sinn bedeutet dies, dass ein Team von Experten durch eine einfach strukturierte Kooperation Aufgaben bewältigen kann, die ein einzelner Experte allein nicht zu lösen vermag. Da sich nun Formale Sprachen als Problemspezifikationen interpretieren lassen, liegt es nahe, Sprachen akzeptierende Automaten oder Sprachen generierende Grammatiken zur Modellierung der Experten zu betrachten.

Die vorliegende Arbeit ist wie folgt aufgebaut: Zunächst wird eine ausführliche Motivation für die PCRA-Systeme gegeben und der aktuelle Forschungsstand dargelegt. Nach dem Festlegen grundlegender Definitionen und Notationen werden die wichtigsten Vertreter von bereits bekannten PC-Systemen anhand von Defi-

nitionen, Beispielen und bekannten Resultaten vorgestellt. Dies soll einen ersten Eindruck von der Funktionsweise, den unterschiedlichen Kommunikationsstrategien und den sich daraus ergebenden Eigenschaften solcher Systeme vermitteln. Darüber hinaus dienen diese Ausführungen als Modellierungs- und Vergleichsgrundlage für die PCRA-Systeme. Nachfolgend wird das Modell des Restart-Automaten detailliert beschrieben, da es die Grundlage für die hier betrachteten Systeme bildet.

Im darauffolgenden Hauptteil dieser Arbeit werden PCRA-Systeme eingeführt und untersucht. Dazu wird zunächst das Kommunikationsprotokoll erläutert, welches festlegt, wie die einzelnen Restart-Automaten miteinander kommunizieren, und damit die Zusammenarbeit der Komponenten essentiell bestimmt. Diesbezüglich werden auch wesentliche Unterschiede zu anderen PC-Systemen herausgestellt. Eine wichtige Eigenschaft der Kommunikationsstruktur ist die Tatsache, ob sie zentralisiert oder nichtzentralisiert ist. Während in einer nichtzentralisierten Kommunikationsstruktur jede Komponente mit jeder anderen Komponente Information austauschen darf, gibt es in einer zentralisierten Kommunikationsstruktur eine ausgezeichnete Master-Komponente, sodass alle anderen (Client-) Komponenten ausschließlich mit der Master-Komponente nicht jedoch mit einer anderen Client-Komponente kommunizieren dürfen. Es hat sich herausgestellt, dass PCRA-Systeme mit zentraler und nichtzentraler Kommunikationsstruktur die gleiche Ausdrucksstärke besitzen, die Zentralisierung der Kommunikation also keinen Nachteil bewirkt (das ist im Allgemeinen bei PC-Systemen nicht so).

Im Folgenden werden PCRA-Systeme mit der *non-forgetting*-Eigenschaft untersucht und obwohl diese Eigenschaft eine echte Erweiterung für diverse Typen von Restart-Automaten darstellt und deren Ausdrucksstärke erhöht, wird gezeigt, dass diese Eigenschaft innerhalb der Systeme keine Erhöhung der Ausdrucksstärke mit sich bringt.

Um Aussagen über die Robustheit der PCRA-Systeme treffen zu können, werden die Sprachklassen der von PCRA-Systemen charakterisierten Sprachen bezüglich Abschlusseigenschaften analysiert. Die betrachteten Sprachklassen sind unter diversen Sprachoperationen abgeschlossen und einige Sprachklassen sind sogar *abstrakte Sprachfamilien* (sogenannte *AFL's*). Im darauffolgenden Abschnitt wird die Ausdrucksstärke von PCRA-Systemen untersucht und mit der von PC-Systemen von endlichen Automaten und mit der von Mehrkopfautomaten verglichen. PC-Systeme von endlichen Automaten besitzen bekanntermaßen die gleiche Ausdrucksstärke wie Einwegmehrkopfautomaten und bilden eine untere Schranke für die Ausdrucksstärke von PCRA-Systemen mit Einwegkomponenten. Tatsächlich sind PCRA-Systeme auch dann stärker als PC-Systeme von endlichen Automaten, wenn die Komponenten für sich genommen die gleiche Ausdrucksstärke besitzen,

also die regulären Sprachen charakterisieren. Für PCRA-Systeme mit Zweiwegekomponenten werden als untere Schranke die Sprachklassen der Zweiwegemehrkopfautomaten im deterministischen und im nichtdeterministischen Fall gezeigt, welche wiederum den bekannten Komplexitätsklassen L (deterministisch logarithmischer Platz) und NL (nichtdeterministisch logarithmischer Platz) entsprechen. Als obere Schranke wird die Klasse der kontextsensitiven Sprachen gezeigt.

Außerdem werden für PCRA-Systeme spezifische Probleme auf ihre Entscheidbarkeit hin untersucht. Es wird gezeigt, dass Leerheit, Universalität, Inklusion und Gleichheit bereits für die schwächste Art von PCRA-Systemen nicht einmal semientscheidbar sind. Für das Wortproblem wird gezeigt, dass es im deterministischen Fall in quadratischer Zeit und im nichtdeterministischen Fall in exponentieller Zeit entscheidbar ist.

Im letzten Abschnitt werden die erzielten Resultate noch einmal zusammengefasst und diskutiert. Offen gebliebene Fragen werden aufgezeigt und Ansätze für weitergehende Forschung über dieses Thema werden angeboten.

Publications

Some ideas and results of this thesis were already presented at the following workshops and conferences:

[Goe09] Marcel Goehring. PC-systems of restarting automata. In Jöran Mielke, Ludwig Staiger, and Renate Winter, editors, *Theorietag Automaten und Formale Sprachen 2009*, pages 26–27, Universität Halle-Wittenberg, Institut für Informatik, 2009. Technical Report 2009/03.

[Vol10] Marcel Vollweiler. Centralized Versus Non-Centralized Parallel Communicating Systems of Restarting Automata. In Friedrich Otto, Norbert Hundeshagen, and Marcel Vollweiler, editors, *20. Theorietag Automaten und Formale Sprachen 2010*, number 2010/3 in Kasseler Informatikschriften, pages 130–135. Fachbereich Elektrotechnik / Informatik, Universität Kassel, 2010. http://nbn-resolving.de/urn:nbn:de:hebis:34-2010110534894.

[HOV11] Norbert Hundeshagen, Friedrich Otto, and Marcel Vollweiler. Transductions Computed by PC-Systems of Monotone Deterministic Restarting Automata. In Michael Domaratzki and Kai Salomaa, editors, *Implementation and Application of Automata*, volume 6482 of *Lecture Notes in Computer Science*, pages 163–172. Springer Berlin / Heidelberg, 2011.

[VO12] Marcel Vollweiler and Friedrich Otto. Systems of Parallel Communicating Restarting Automata. In Rudolf Freund, Markus Holzer, Bianca Truthe, and Ulrich Ultes-Nitsche, editors, *4th Workshop on Non-Classical Models for Automata and Applications*. Österreichische Computer Gesellschaft, 2012.

[Vol12] Marcel Vollweiler. Closure Properties of Parallel Communicating Restarting Automata Systems. In František Mráz, editor, *22. Theorietag Automaten und Formale Sprachen 2012*, pages 133–138. MATFYZPRESS, publishing house of the Faculty of Mathematics and Physics, Charles University in Prague, 2012.

In [Goe09] the notion of systems of parallel communicating restarting automata was announced, where the communication protocol differs a little bit from the one that is used in this thesis. In [Vol10] the equivalence of centralized and non-centralized systems was stated with a proof outline. Then, in [HOV11] systems of two monotone and deterministic restarting automata were considered that compute transductions. There, the notation of the communication states is simplified, due to the existence of only two components. In [VO12] the following results on systems of parallel communicating restarting automata were published: the equivalence between centralized and non-centralized systems, the comparison to individual restarting automata, systems of parallel communicating finite automata, and multi-head automata with respect to their computational power. Finally, in [Vol12] a summery of the Section 5.5 was presented, which contains the results on closure properties of language classes that are characterized by systems of parallel communicating restarting automata.

Acknowledgements

At first, I would like to offer special thanks to my supervisor Prof. Dr. Friedrich Otto for his continual support, all the fruitful discussions, suggestions, and the critical comments during the whole development of this thesis. Moreover, I thank Prof. Dr. Jürgen Dassow for refereeing this thesis and for his stimulating questions whenever I had a talk on the topic of the thesis.

Then, I would like to thank my colleague Norbert Hundeshagen for the frequent inspiring discussions, the creative proposals, and the great teamwork over the years during the development of this thesis.

I would also like to acknowledge Prof. Dr. Helmut Jürgensen, PD Dr. habil. Henning Bordihn, and Dr. Suna Bensch, who aroused my interest in alternative models of computability and particularly in systems of parallel communicating components during my studies and helped me indirectly to find the topic of this thesis.

Moreover, I wish to thank Prof. Dr. Alexander Torossian for his helpful advices, particularly when writing down this thesis.

Last but not least, I thank Gabriele Djawadi and Angelika Hoffmann-Hesse very much for their support for a lot of administrative tasks.

Contents

1 Introduction

Over the last decades distributed and parallel computing has gained a more and more important role. Significant reasons for this are on the one hand an increase of computational power and therefore a more efficient and faster computation (see e.g. [Fou94], [Hro97]), and on the other hand an efficient shared allocation of resources [CDK05]. The former reason is especially relevant in the development of real-time applications and for problems where the power of sequential devices is not sufficient anymore. Moreover, parallelism is somehow natural, so there are various examples where parallelism occurs in the natural environment. A frequently given example is the human brain [Fou94], where information is interchanged in a massively parallel way between neurons via the synapses. Another example is the growth of a cell culture, where many cells divide simultaneously. An additional reason why parallelism is useful is the fact that many problems and their underlying data structures are somehow suitable for a divide and conquer approach. Consider, e.g. the typesetting of a book, where the different chapters can be worked on more or less independently. Consider two more simple examples that are suitable for parallelization [SCB05]:

1. The problem to sum the elements of a vector $V = (v_1, v_2, \ldots, v_n)$ to a scalar A:
$$A = \sum_{i=1}^{n} v_i.$$
With p processors $P_1, P_2, \ldots, P_p$ and the assumption that there exists a k such that $n = k \cdot p$ we can divide the problem into p parts such that, for each $1 \leq j \leq p$, P_j computes the partial sum $A_j = v_{jk-k+1} + \ldots + v_{jk}$. At the end the partial results are summed to $A = A_1 + A_2 + \ldots + A_p$.

2. Computing the prime number sieve that is also called the 'sieve of Eratosthenes'. Let $S(k)$ be the set of all prime numbers less than k. Then it can be tested whether any n between k and k^2 is a prime number by trying to divide it by the elements of $S(k)$ (if there are i and j such that $n = i \cdot j < k^2$, then at least one of i and j must be less than k). Thus, all these n can be checked in parallel and new primes are added to $S(k^2)$. Then we can continue with testing the numbers between k^2 and $(k^2)^2$ and so forth.

Distributed and parallel computing were not only developed in computer science applications but also in theoretical computer science. Some popular examples for theoretic approaches of distributed and parallel computing are: multi-head automata [RS59, Ros66][1], multi-processor automata [Bud87], artificial neural networks [Hay94], networks of parallel language processors [CS97], Lindenmayer systems [PL96], eco-grammar systems [CKKP94, CKKP97], cooperating distributed grammar and automata systems (CD systems), and parallel communicating grammar and automata systems (PC systems)[2]. Throughout the development it can be observed that by combining simple components to systems usually new interesting properties arise and the computational power increases although the connection between the components is mostly very simply structured. Moreover, various approaches of cooperation and communication were considered within that context. In multi-head and multi-processor automata one device just uses some resources in parallel (i.e. different reading heads or processors, respectively). The typical structure of a network is a graph, where the nodes can be seen as independent processors and the edges are communication connections. In such networks, the particular power of the processors and the way of communication differs from one model to another. PC grammar and automata systems can be seen as special types of (communication) networks.

Some of the examples above are motivated by biological issues: the artificial neural networks as a formal model for the biological neural network, the Lindenmayer systems for describing the growth of plants, and the eco-grammar systems for modeling the interaction between agents in a common biological or social environment. The CD and PC systems, mentioned above, are of particular interest, because they represent two approaches of (human) problem solving: the *blackboard model* and the *classroom model* [CDKP94]. The blackboard model originally arises from the field of artificial intelligence and consists of a central commonly used data structure (blackboard) that contains the ‘problem solving state data’, several separate and independent knowledge sources (experts), and an implicit control that determines the currently working knowledge source [Nii86]. To solve a given problem that is initially written on the blackboard, the experts work alternatingly on the blackboard, one at each point of time, and change its content until the problem is solved. The essential idea of this strategy is to combine the different knowledge and skills of the experts to find a solution for a problem that cannot be solved by a single expert or that can just be solved more efficiently in this manner. This way of problem solving is indeed distributed, but the interaction of the experts is rather sequential. In contrast, in the classroom model, introduced in [CDKP94], only one distinguished knowledge source works on the blackboard, this

[1] The original definitions do not supply a parallel work, but over the years an equivalent definition was established where all read-only heads are moved simultaneously, see e.g. [HKM11].

[2] CD and PC grammar systems are considered in detail in [CDKP94]. Examples for CD and PC automata systems are presented below.

is the team leader, and the others work on their own data structures ('notebooks'). All experts work independently and mainly simultaneously (in parallel), and they can communicate with each other. Thus, the experts solve parts of the problem and communicate the partial results. The problem is solved once the solution is written on the blackboard, i.e. the data structure of the leader contains it, and the leader decides that this is indeed a solution to the given problem.

Distributed and parallel computing is one of the two main aspects of this work. The other one is the model of the *restarting automaton*. The restarting automaton is a device from formal language theory that was investigated throughout the last two decades. Originally it was developed by Jančar et al. in 1995 to model a linguistic technique that is called *analysis by reduction* [JMPV95] (see also [JMP$^+$97, JMPV97, MPV96, NO01, NO03, Ott06, PLO03]). This technique is used for checking the syntactical correctness of a sentence of a natural language with a free word order, which means that the positions of the parts of the sentence are not strictly determined. Over the years many properties (e.g. monotonicity [JMPV98, JMPV99, JMPV07, JOMP05a, JOMP05b, JOMP08, MPJV97, Plá01], lookahead hierarchies [Mrá01, Sch10, Sch11], the number of auxiliary symbols [JO06b, JO06a], the descriptional complexity [KR08, Rei07], closure properties [JLNO04]) and variants of the restarting automata were considered, and interesting connections to well-known language classes were achieved [JMP$^+$95, JLNO04, NO99a, NO99b, Ott03], for example: stateless restarting automata [KMO09, Ott08b], shrinking restarting automata [JO05, JO07], nonforgetting restarting automata [Mes08, MS04, Mes07, MO06, MO11], restarting tree automata [SO07a, SO07b, Sta08], and clearing restarting automata [ČM10, ČM11].

Additionally, interesting results were shown for cooperating restarting automata, i.e. for CD systems of restarting automata [MO07a, MO07b, MO09, Ott08a, Ott10], where the components work sequentially one after another on a common string. It was proved that these systems have the same computational power as nonforgetting automata [Mes07, MO07a]. Moreover, connections between the language classes accepted by CD systems of restarting automata, rational trace languages, and context-free trace languages have been established [NO10, NO11a, NO11b]. From another point of view the CD systems of restarting automata can be seen as finite-state acceptors with translucent letters [NO11c].

The aim of this thesis is to combine these two concepts: systems of parallel communicating components and restarting automata. *Systems of parallel communicating restarting automata* are defined (abbreviated as *PCRA systems*), where several restarting automata work in parallel and independently of each other with the possibility to communicate. For this, a specification of an appropriate communication protocol is necessary. Afterwards, various properties of these systems are investigated like their computational power, decidability questions, and closure properties. Moreover, they are related to existing results from formal language theory, complexity theory, and computability theory. This thesis consists of six main parts. It is organized as follows.

At first, we give some preliminaries and basic notations in Section 2 that are used within this work. In Section 3, the most popular representatives of parallel communicating systems are presented giving their definitions, examples, and results that can be found in the literature. This should give a first impression of the structure, the used communication protocols, the principles of operation, and their properties. We will use these explanations sometimes for the purpose of comparisons. Section 4 gives a short overview of the model of the restarting automaton, which is used as the basic device for the systems investigated in Section 5.

Section 5 is the main part of this work. There, PCRA systems are introduced and investigated. At first, these systems are defined and explained. The communication protocol is presented and important differences to the systems given in Section 3 are emphasized. Moreover, a first detailed example is given. Then, after defining some more useful technical material in Subsection 5.1, we will see some other example systems for various well-known formal languages in Subsection 5.2. More examples are given throughout the further subsections whenever it seems helpful. Afterwards, we consider properties that are typically interesting for systems of parallel communicating components. In Subsection 5.3, we investigate an essential property of the communication, namely whether a centralized communication structure is a restriction for the computational power of our systems. In Subsection 5.4, we study the effect of the 'nonforgetting property' on our systems. For single restarting automata (that are in general 'forgetting'), this property is an extension that in fact increases the computational power. In Subsection 5.5, we show some closure properties for language classes that are characterized by our systems. We will see that some of them are so-called 'abstract families of languages' (AFLs). Then, in Section 5.6 we investigate the computational power of PCRA systems. In this context our systems are compared with single restarting automata and with other types of parallel communicating systems, and we obtain some correlations to popular complexity classes. Moreover, systems of shrinking restarting automata are considered there. In Section 5.7 it is shown that, although the word problem is decidable in quadratic time for deterministic systems and in exponential time for nondeterministic systems, many other questions are not decidable even for systems with weak types of restarting automata as components.

In the last section we summarize our results, give a short discussion on interesting problems that remain open, and present some approaches for further research.

2 Preliminaries

Here basic definitions and notations of formal language theory that are used in the subsequent sections are introduced. Since this is only a short survey, the reader is referred to the literature for further details, e.g. [HU79, RS97].

We denote *sets* with capital Latin letters. For a set M, the number of elements is denoted by $|M|$. For two sets M_1 and M_2, we can apply the usual set operations: union ($M_1 \cup M_2$ and $M_1 \dot{\cup} M_2$ for the disjoint union), intersection ($M_1 \cap M_2$), difference ($M_1 \setminus M_2$), cartesian product ($M_1 \times M_2$), the power set ($\mathcal{P}(M_1)$), and the set of all finite subsets ($\mathcal{P}_e(M_1)$). With M^n we mean the cartesian product

$$\underbrace{M \times M \times \ldots \times M}_{n-\text{times}}.$$

For n sets $M_1, M_2, \ldots, M_n$, we can define an n-ary relation R, i.e. a subset of $M_1 \times M_2 \times \ldots \times M_n$. If M_1 is a subset of M_2, we notate this by $M_1 \subseteq M_2$. If the subset is even proper, then this is denoted by $M_1 \subset M_2$.

In general, the members of a set are not ordered and no member occurs more than once. In contrast, an n-tuple is a finite and ordered list of objects that is denoted by $A = (a_1, a_2, \ldots, a_n)$, where the members have not necessarily to be distinct. Such tuples are usually used for the description of mathematical structures and the formalization of formal language devices.

An *alphabet* is a finite collection of distinct *symbols* and is denoted by a capital Greek letter. Symbols are mostly written with the first lower case Latin letters a, b, c, and so forth. A *word* or *string* is a finite sequence of symbols and denoted by the last Latin letters u, v, w, x, y, and z. The length $|w|$ of a word w is just the number of symbols it consists of, e.g. for the word $w = abcba$, we have $|w| = |abcba| = 5$. The word of length zero is called the *empty word* and denoted by ε. With $|w|_a$ we mean the number of occurences of the symbol a in w. Two words u and v can be combined to a word w with the associative operation of concatenation: $w = u \cdot v$. Obviously, $|w| = |u \cdot v| = |u| + |v|$ and $\varepsilon \cdot w = w \cdot \varepsilon = w$. The operator $\cdot$ is often omitted and we write $w = uv$. Let Σ be an alphabet, then Σ^* is the monoid over Σ (according to the concatenation as operation and the identity element ε), i.e. the set of all words that can be obtained by combining an arbitrary but finite number of symbols of Σ. The set Σ^* without ε is denoted by Σ^+. Moreover, with Σ^n and $\Sigma^{\leq n}$ we denote the set of all words of length n and of length at most n, respectively. A subset L of Σ^* is called a *(formal) language* over the alphabet Σ. Besides the usual set operations mentioned above we need some more operations for formal languages L, L_1, and L_2 over an alphabet Σ:

- complement: $\overline{L} = \Sigma^* \setminus L$,
- reversal: $L^R = \{w^R \mid w \in L\}$, where for a word $w = a_1a_2 \dots a_n$, the reversal is $w^R = a_n \dots a_2a_1$ that is also called the *mirror image*,
- product: $L_1 \cdot L_2 = \{uv \mid u \in L_1 \text{ and } v \in L_2\}$,
- Kleene closure:
$$L^* = \bigcup_{i=0}^{\infty} L^i,$$
where $L^0 = \{\varepsilon\}$ and $L^i = L \cdot L^{i-1}$ for all $i > 0$,
- positive closure:
$$L^+ = \bigcup_{i=1}^{\infty} L^i.$$

Another operation that can be applied to words as well as to languages are homomorphisms, which are defined by

$$h : \Sigma \to \Gamma^*,$$

where Σ and Γ are alphabets. Usually, h is extended to $h : \Sigma^* \to \Gamma^*$ by defining $h(\varepsilon) = \varepsilon$ and $h(ua) = h(u)h(a)$ for any word $u \in \Sigma^*$ and any symbol $a \in \Sigma$. A homomorphism is called *non-erasing* if $|h(a)| > 0$ for all $a \in \Sigma$.

A set of languages is called a *language class*. Famous language classes in formal language theory are the set of *regular languages* (REG), the set of *context-free languages* (CFL), the set of *context-sensitive languages* (CSL), and the set of *recursively enumerable languages* (RE). The connection between these classes according to the inclusion relation is given through the Chomsky hierarchy:

$$\mathsf{REG} \subset \mathsf{CFL} \subset \mathsf{CSL} \subset \mathsf{RE}.$$

These language classes are characterized by formal grammars of different types, where the type indicates particular restrictions: regular grammars, context-free grammars, context-sensitive grammars, and unrestricted grammars. Over the years there were developed and investigated many more types of formal grammars and automata models characterizing language classes with rather different properties.

Other well-known language classes we will use here are: the *deterministic context-free languages* (DCFL), which are characterized by deterministic pushdown automata; the *growing context-sensitive languages* (GCSL) [DW86], which are characterized by strictly monotone grammars, i.e. grammars with rules only of the form $\alpha \to \beta$ s.t. $|\alpha| < |\beta|$; the *Church-Rosser languages* (CRL) [MNO88], characterized by Church-Rosser Thue systems; and the classes of languages that can be accepted by a Turing machine with only a logarithmic amount of space in the deterministic case (L) or in the nondeterministic case (NL). The first three classes can also be placed within the Chomsky hierarchy:

$$\mathsf{REG} \subset \mathsf{DCFL} \begin{array}{c} \rotatebox{45}{$\subset$}\ \mathsf{CFL}\ \rotatebox{-45}{$\subset$} \\ \rotatebox{-45}{$\subset$}\ \mathsf{CRL}\ \rotatebox{45}{$\subset$} \end{array} \mathsf{GCSL} \subset \mathsf{CSL} \subset \mathsf{RE}.$$

An important property of language classes that we will investigate here are closure properties. We say that a language class $\mathcal{L}$ ist closed under an n-ary operation $\circ \subseteq \mathcal{L}^n$ if, for arbitrary languages $L_1, L_2, \ldots, L_n \in \mathcal{L}$, it holds that $\circ(L_1, L_2, \ldots, L_n) \in \mathcal{L}$. A language class is called an *abstract family of languages* (abbreviated by *AFL*) if it is closed under the operations union, intersection with regular languages, non-erasing homomorphisms, inverse homomorphisms, product, and positive closure.

Let M be a particular formal language device (e.g. an automaton, a grammar, a system etc.). Then $\mathcal{L}(M)$ denotes the language class consisting of all languages that can be characterized (generated, accepted) by this device.

3 Systems of parallel communicating components

3.1 Multi-head finite automata and multi-processor automata

In this section we consider the model of multi-head finite automata. Although it is not a system of several components that work in a distributed manner, it plays an important role within this research field. On the one hand, it is closely connected to the parallel communicating systems of finite automata, and the coordination of the various heads can be seen as a simple kind of implicit communication. On the other hand, they characterize the popular complexity classes L and NL that are the classes of languages accepted by deterministic, nondeterministic respectively, Turing machines using only a logarithmic amount of workspace (according to the length of the input).

The *multi-head finite automaton* was introduced by Rabin and Scott [RS64] and Rosenberg [Ros67]. It consists of a finite state control and an input tape with a fixed number of read-only heads (see Figure 3.1). Basically, the finite control requests the input symbols read by all heads and then determines a successor state and the moving directions for all heads. The heads are abstract in the sense that they can freely pass each other. Moreover, we consider so-called non-sensing multi-head automata, which means that the automaton cannot check whether some heads have the same position on the tape or not.

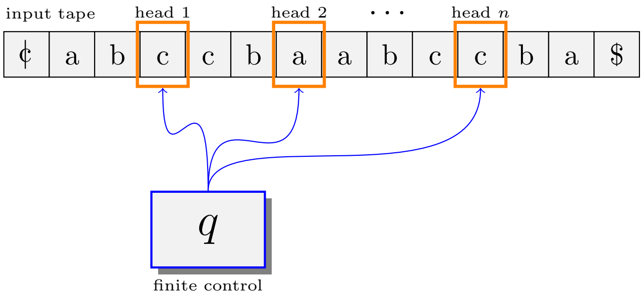

Figure 3.1: Schematic representation of a multi-head finite automaton.

Formally[1], a nondeterministic two-way n-head automaton M ($\mathsf{2\text{-}NFA}(n)$ for

[1] Here, for technical reasons we do not use the original definition, but a modified one that can be found in e.g. [HKM09].

short) is a tuple

$$M = (Q, \Sigma, n, \delta, ¢, \$, q_0, F)$$

with the finite set of states Q, the input alphabet Σ, the initial state $q_0 \in Q$, and the set of final states $F \subseteq Q$. The symbols ¢ and \$ are not included in Σ and used as the left and the right sentinels of the input. Further, δ is a mapping of $Q \times (\Sigma \cup \{¢, \$\})^n$ into the finite subsets of $Q \times \{-1, 0, 1\}^n$. We write $(q, (d_1, d_2, \ldots, d_n)) \in \delta(p, (a_1, a_2, \ldots, a_n))$ if the automaton M is allowed to change into the state q and to move the i-th head d_i steps to the right for all $1 \leq i \leq n$, when it is currently in state p and reads the input symbol a_i with the i-th head for all $1 \leq i \leq n$. In particular, if $d_i = -1$ for some $1 \leq i \leq n$, then the i-th head is moved one step to the left. In the cases $d_i = 0$ and $d_i = 1$, the i-th head does not move and keeps the current position, or it moves one step to the right, respectively. Whenever a head of the automaton reads the left sentinel ¢, or the right sentinel \$ respectively, it is not allowed to move to the left, to the right respectively. That is, $(q, (d_1, d_2, \ldots, d_n)) \in \delta(p, (a_1, a_2, \ldots, a_n))$ with $a_i = ¢$ ($a_i = \$$) for some $1 \leq i \leq n$ implies $d_i \in \{0, 1\}$ ($d_i \in \{-1, 0\}$).

A configuration of a 2-NFA(n) M with input w is an n-tuple $(x_1qy_1, x_2qy_2, \ldots, x_nqy_n)$, where $x_1y_1 = x_2y_2 = \ldots = x_ny_n = ¢w\$$, q is the current state of the finite control, and for all $1 \leq i \leq n$, the i-th head is positioned on the first symbol of y_i. The initial configuration for an input w is the n-tuple $(q_0¢w\$, q_0¢w\$, \ldots, q_0¢w\$)$. For two configurations $(x_1pa_1y_1, x_2pa_2y_2, \ldots, x_npa_ny_n)$ and $(x'_1qy'_1, x'_2qy'_2, \ldots, x'_nqy'_n)$ of M, a computation step

$$(x_1pa_1y_1, x_2pa_2y_2, \ldots, x_npa_ny_n) \vdash_M (x'_1qy'_1, x'_2qy'_2, \ldots, x'_nqy'_n)$$

is possible if and only if $(q, (d_1, d_2, \ldots, d_n)) \in \delta(p, (a_1, a_2, \ldots, a_n))$ and for all $1 \leq i \leq n$,

1. $d_i = -1$, $x_i = x''_ib_i$, $x'_i = x''_i$, and $y'_i = b_ia_iy_i$; or

2. $d_i = 0$, $x'_i = x_i$, and $y'_i = a_iy_i$; or

3. $d_i = 1$, $x'_i = x_ia_i$, and $y'_i = y_i$.

A computation of an automaton M is denoted by $\vdash_M^*$, which is the reflexive and transitive closure of $\vdash_M$. If there is no applicable transition for some configuration, then the automaton halts. An input word w is accepted by an automaton M if and only if M starts the computation from the initial configuration on input w and halts after a finite number of computation steps in a final state. The language that is accepted by M with initial state q_0 is

$$\begin{aligned} L(M) = \{w \in \Sigma^* \mid {} & (q_0 ¢w\$, q_0 ¢w\$, \ldots, q_0 ¢w\$) \vdash_M^* (x_1 q a_1 y_1, x_2 q a_2 y_2, \ldots, x_n q a_n y_n) \\ & \text{such that } q \in F, x_i y_i = ¢w\$ \text{ for all } 1 \leq i \leq n, \\ & \text{and } \delta(q, (a_1, a_2, \ldots, a_n)) = \emptyset\}. \end{aligned}$$

Whenever the mapping δ of an n-head automaton M is a (partial) function into the set $Q \times \{-1, 0, 1\}^n$, then we say M is a deterministic n-head automaton, $\mathsf{2\text{-}DFA}(n)$ for short. Moreover, if δ maps into the set $Q \times \{0, 1\}^n$, i.e. no left moves can be done, then we call M a one-way n-head automaton, that is, a $\mathsf{1\text{-}DFA}(n)$ in the deterministic case or a $\mathsf{1\text{-}NFA}(n)$ in the nondeterministic case.

The class of all languages that can be accepted by any n-head automaton of type $\mathsf{X} \in \{\mathsf{2\text{-}NFA}, \mathsf{2\text{-}DFA}, \mathsf{1\text{-}NFA}, \mathsf{1\text{-}DFA}\}$ is denoted by $\mathcal{L}(\mathsf{X}(n))$. The class of languages accepted by a multi-head finite automaton of type X with arbitrarily many heads is denoted by

$$\mathcal{L}(\mathsf{X}) = \bigcup_{n \geq 1} \mathcal{L}(\mathsf{X}(n)).$$

In the following example a $\mathsf{1\text{-}DFA}(2)$ M is given that accepts the language $L_{a^n b^n c^n} = \{a^n b^n c^n \mid n \geq 1\}$. The automaton M is just a deterministic finite automaton with one additional head. Nevertheless it can accept languages like $L_{a^n b^n c^n}$ that are not even context-free. This simple example demonstrates how even small extensions by additional resources that can be used in parallel can increase the computational power of a device.

Example 1. A two-head automaton that accepts the language $L_{a^n b^n c^n}$ behaves as follows: initially, reading the ¢-symbol with both heads, the heads are moved one step to the right. Then, while reading a's the second head moves to the right and the first head stays on the first a. When the second head reaches the first b of the input, then both heads move to the right simultaneously, where the first head only reads a's and the second head only reads b's. At some point in time the first head reads the first b and the second head reads the first c - at least if the number of a's and b's are equal. Then, both heads are synchronously moved to the right, while the first head reads the b's and the second head reads the c's. If the number of b's and c's are equal, then the first head reads the first c at the same point in time that the second head reads the \$-symbol. In this case M switches into a final state and halts. If the input word is not of the correct form, then M will halt without reaching the final state and thus reject. Formally, M is given by

$$M = (\{p, q, r, s\}, \{a, b, c\}, 2, \delta, ¢, \$, p, \{s\}),$$

where

$$\begin{array}{ll}\delta(p,(\cent,\cent))=(p,(1,1)), & \delta(q,(b,c))=(r,(1,1)),\\ \delta(p,(a,a))=(p,(0,1)), & \delta(r,(b,c))=(r,(1,1)),\\ \delta(p,(a,b))=(q,(1,1)), & \delta(r,(c,\$))=(s,(0,0)).\\ \delta(q,(a,b))=(q,(1,1)), & \end{array}$$

The computation on an input word $a^kb^kc^k$ for some $k>0$ is

$$\begin{array}{lll} & (p\cent a^kb^kc^k\$, & p\cent a^kb^kc^k\$\quad)\\ \vdash_M & (\cent pa^kb^kc^k\$, & \cent pa^kb^kc^k\$\quad)\\ \vdash_M^k & (\cent pa^kb^kc^k\$, & \cent a^kpb^kc^k\$\quad)\\ \vdash_M & (\cent aqa^{k-1}b^kc^k\$, & \cent a^kbqb^{k-1}c^k\$)\\ \vdash_M^{k-1} & (\cent a^kqb^kc^k\$, & \cent a^kb^kqc^k\$\quad)\\ \vdash_M & (\cent a^kbrb^{k-1}c^k\$, & \cent a^kb^kcrc^{k-1}\$)\\ \vdash_M^{k-1} & (\cent a^kb^krc^k\$, & \cent a^kb^kc^kr\$\quad)\\ \vdash_M & (\cent a^kb^ksc^k\$, & \cent a^kb^kc^ks\$\quad). \end{array}$$

Observe that M is deterministic, since in each situation at most one move is possible. Moreover, it moves the heads only to the right but not to the left, thus it is a one-way automaton. All in all, M is a $\mathsf{1\text{-}DFA}(2)$ with $L(M)=L_{a^nb^nc^n}$. □

There are several results concerning the computational power of multi-head finite automata. For multi-head automata with only one head we just obtain a finite automaton and thus for all variants of one-way, two-way, deterministic, and nondeterministic automata the regular languages are characterized [RS59, She64]:

$$\mathcal{L}(\mathsf{1\text{-}DFA}(1))=\mathcal{L}(\mathsf{1\text{-}NFA}(1))=\mathcal{L}(\mathsf{2\text{-}DFA}(1))=\mathcal{L}(\mathsf{2\text{-}NFA}(1))=\mathsf{REG}.$$

In [YR78] a hierarchy of the language classes according to the number of heads was shown for one-way multi-head automata:

$$\mathcal{L}(\mathsf{1\text{-}DFA}(n))\subset\mathcal{L}(\mathsf{1\text{-}DFA}(n{+}1)) \text{ and } \mathcal{L}(\mathsf{1\text{-}NFA}(n))\subset\mathcal{L}(\mathsf{1\text{-}NFA}(n{+}1)) \text{ for all } n\geq 1.$$

For this, the language

$$L_b=\{w_1*w_2*\ldots*w_{2b}\mid (w_i\in\{0,1\}^*) \text{ and } (w_i=w_{2b+1-i}) \text{ for all } 1\leq i\leq 2b\}$$

was shown to be accepted by a $\mathsf{1\text{-}DFA}(n+1)$ but not to be accepted by any $\mathsf{1\text{-}NFA}(n)$ for $b=\binom{n+1}{2}$. The same language and the fact that deterministic one-way automata are closed under complementation are used to separate the deterministic and the nondeterministic language classes:

$$\mathcal{L}(\mathsf{1\text{-}NFA}(2))\setminus\mathcal{L}(\mathsf{1\text{-}DFA})\neq\emptyset.$$

Comparing one-way and two-way multi-head automata this leads to the result that the former ones are less powerful than the latter ones:

$$\mathcal{L}(\mathsf{1\text{-}DFA}(n)) \subset \mathcal{L}(\mathsf{2\text{-}DFA}(n)) \text{ and } \mathcal{L}(\mathsf{1\text{-}NFA}(n)) \subset \mathcal{L}(\mathsf{2\text{-}NFA}(n)) \text{ for all } n \geq 2.$$

This holds because the language of palindromes

$$L_{pal} = \{w \mid w \in \{a,b\}^* \text{ and } w = w^R\}$$

cannot be accepted by any one-way multi-head automaton independently of how many heads are used. But L_{pal} can easily be accepted by a deterministic two-way multi-head automaton with only two heads: initially the first head is positioned on the first letter of the input, while the second head is moved to the right end, i.e. the last letter of the input word. Then, both heads are moved simultaneously comparing each scanned symbol, the first head reads the input from the left to the right and the second head reads the input from the right to the left. Reaching the according other end of the tape, the word is accepted by the automaton. Otherwise, if two compared symbols are not equal, the automaton halts and rejects, since the input is not a palindrome.

Hierarchy results for two-way multi-head automata can be found in [Mon80, Iba73, Sud77]. Summarizing it is known that

$$\mathcal{L}(\mathsf{2\text{-}DFA}(n)) \subset \mathcal{L}(\mathsf{2\text{-}DFA}(n+1)) \text{ and } \mathcal{L}(\mathsf{2\text{-}NFA}(n)) \subset \mathcal{L}(\mathsf{2\text{-}NFA}(n+1)).$$

An important result that we will use below is the characterization of the well-known complexity classes L and NL by two-way multi-head automata [Har72]:

$$\mathsf{L} = \mathcal{L}(\mathsf{2\text{-}DFA}) \text{ and } \mathsf{NL} = \mathcal{L}(\mathsf{2\text{-}NFA}).$$

Another approach concerning parallel computations in finite automata is the *multi-processor automaton* that was introduced by A.O. Buda in [Bud87]. This device is equipped with a finite number of processors that are working simultaneously on a common working tape. The behaviour of each processor depends on its own current state and the input symbol it currently reads and is determined by a common transition mapping. Moreover, a global switching function determines, depending on the current states of all processors, which of them are active and which of them are idle.

In contrast to multi-head automata that have only one control with several heads the computation of a multi-processor automaton with several (more or less independent[2]) processing units has a higher degree of distribution. Nevertheless it was proved that both concepts have the same computational power. In the original paper it was proved that each n-processor automaton can be simulated by an n-head automaton. Later Ďuriš et al. proved the other direction [ĎJKL98] with the following consequences:

$$\mathcal{L}(\mathsf{1\text{-}DP}) = \mathcal{L}(\mathsf{1\text{-}DFA}), \mathcal{L}(\mathsf{1\text{-}NP}) = \mathcal{L}(\mathsf{1\text{-}NFA}), \mathcal{L}(\mathsf{2\text{-}DP}) = \mathcal{L}(\mathsf{2\text{-}DFA}),$$
$$\text{and } \mathcal{L}(\mathsf{2\text{-}NP}) = \mathcal{L}(\mathsf{2\text{-}NFA}),$$

where $\mathcal{L}(\mathsf{1\text{-}DP})$ ($\mathcal{L}(\mathsf{1\text{-}NP})$, $\mathcal{L}(\mathsf{2\text{-}DP})$, $\mathcal{L}(\mathsf{2\text{-}NP})$) is the class of languages that are accepted by a one-way deterministic (one-way nondeterministic, two-way deterministic, two-way nondeterministic) multi-processor automaton.

According to the number of heads or processors, respectively, the authors of [ĎJKL98] showed that each nondeterministic n-head automaton can be simulated by an n-processor automaton:

$$\mathcal{L}(\mathsf{1\text{-}NP}(n)) = \mathcal{L}(\mathsf{1\text{-}NFA}(n)) \text{ and } \mathcal{L}(\mathsf{2\text{-}NP}(n)) = \mathcal{L}(\mathsf{2\text{-}NFA}(n)).$$

In the deterministic one-way case this equality does not hold:

$$\mathcal{L}(\mathsf{1\text{-}DP}(2)) \subset \mathcal{L}(\mathsf{1\text{-}DFA}(2)).$$

But the simulation works when two additional processors are used:

$$\mathcal{L}(\mathsf{1\text{-}DFA}(n)) \subseteq \mathcal{L}(\mathsf{1\text{-}DP}(n+2)) \text{ and } \mathcal{L}(\mathsf{2\text{-}DFA}(n)) \subseteq \mathcal{L}(\mathsf{2\text{-}DP}(n+2)).$$

Multi-head and multi-processor automata are early examples for parallel computations in formal language devices. In the next sections systems of parallel communicating devices are presented that have a rather different structure, but also some interesting similarities.

3.2 Parallel communicating finite automata systems

Parallel communicating finite automata systems (PCFA systems for short) were introduced by Martín-Vide, Mateescu, and Mitrana in 2002 [MMM02]. They can be seen as an in-between of the multi-processor automaton on the one hand and the multi-head finite automaton on the other hand. Both kinds of models are somehow extremes in coordinating their work: in the former one the processors

[2] On the one hand a transition step for a particular processor is independent of the other processors. On the other hand it depends on the states of all processors, whether a particular processor is active or not.

work quite independently of each other and the cooperation is very loose, while in the latter one there is just a single control using several heads. Further, PCFA systems can be seen as a counterpart to PC grammar systems with the difference that an automata system accepts input words instead of generating them like a grammar system does.

A PCFA system consists of several finite automata that communicate by their states. Originally, they all use the same input tape, but have their own finite control and one-way read-only head (observe the similarity to multi-processor automata). However, w.l.o.g. it seems more natural to imagine that each automaton has its own input tape like it is shown in Figure 3.2. The finite automata within the system are called components and the number of components determines the degree of the system.

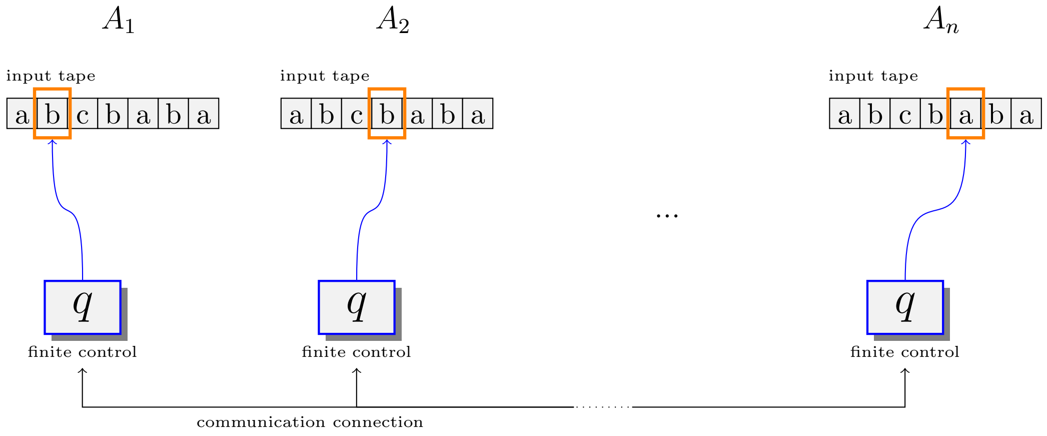

Figure 3.2: Schematic representation of a PCFA system with components A_1, A_2, …, A_n.

Formally, a *parallel communicating finite automata system* $\mathcal{A}$ of degree n is described by an $(n+2)$-tuple

$$\mathcal{A} = (\Sigma, A_1, A_2, \ldots, A_n, K),$$

where Σ is the non-empty finite input alphabet, and for all $1 \leq i \leq n$,

$$A_i = (Q_i, \Sigma, \delta_i, q_i, F_i)$$

is a finite automaton with a set of states Q_i, the initial state $q_i \in Q_i$, the set of final states $F_i \subseteq Q_i$, and the transition mapping $\delta_i : Q_i \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q_i)$. The set

$$K = \{K_1, K_2, \ldots, K_n\} \subseteq \bigcup_{i=1}^{n} Q_i$$

contains the query states that are used for communication[3]. For each component A_i, the set K contains a dedicated query state K_i. In general, a PCFA system is nondeterministic. However, if for all $1 \leq i \leq n$, δ_i is a partial function, and moreover, $\delta_i(s, \varepsilon) \neq \emptyset$ implies $\delta_i(s, a) = \emptyset$ for all $s \in Q_i$ and $a \in \Sigma$, then the system is deterministic.

A PCFA system is called *centralized* if there exists only one $1 \leq i \leq n$ with $K \subseteq Q_i$. The component A_i is then called the *master* of the system, and each communication can only be initiated by A_i. For simplicity it is mostly assumed that the first component is the master of the system.

A configuration of a PCFA system of degree n is a $2n$-tuple

$$(s_1, x_1, s_2, x_2, \ldots, s_n, x_n),$$

where $s_i \in Q_i$ is the current state of A_i, and x_i is the part of the input word that is not yet read by the component A_i, $1 \leq i \leq n$. There are two kinds of computation steps: 1) a local computation step and 2) a communication step. A local step can only be applied if none of the current states is a query state. Then each component of the system performs exactly one computation step. If at least one component is actually in a query state, then communication takes place. That means that each requested component sends its current state to the requesting component. It is required that query states themselves cannot be communicated. Thus, if the requested state is also a query state, then this communication has to be resolved first. Whenever there occurs a circular query, then the computation of the system is blocked, the system halts, and it rejects the input.

Let $(s_1, x_1, s_2, x_2, \ldots, s_n, x_n)$ and $(p_1, y_1, p_2, y_2, \ldots, p_n, y_n)$ be two configurations. Then

$$(s_1, x_1, s_2, x_2, \ldots, s_n, x_n) \vdash (p_1, y_1, p_2, y_2, \ldots, p_n, y_n)$$

is a computation step if and only if one of the following two conditions is fulfilled:

- (local computation step) $K \cap \{s_1, s_2, \ldots, s_n\} = \emptyset$, $x_i = a_i y_i$, $a_i \in \Sigma \cup \{\varepsilon\}$, and $p_i \in \delta_i(s_i, a_i)$ for all $1 \leq i \leq n$; or

- (communication step) $K \cap \{s_1, s_2, \ldots, s_n\} \neq \emptyset$, for all $1 \leq i \leq n$ for which $s_i = K_{j_i}$ and $s_{j_i} \notin K$, $p_i = s_{j_i}$, for all the other $1 \leq r \leq n$, $p_i = s_i$, and for all $1 \leq t \leq n$, $y_i = x_i$.

[3] In difference to the original definition we allow $K \subseteq \{K_1, \ldots, K_n\}$ instead of $K = \{K_1, \ldots, K_n\}$, ommiting communication states that will not be used. Otherwise, an inaccuracy would appear in the case of centralized systems (see below), where component A_1 must unnecessarily contain the state K_1.

In the above definition of a computation step, a component whose state was requested by another component continues its computation from the communicated state. This strategy is called *non-returning mode.* Another strategy is to reset the requested component to its initial state. This is called the *returning mode* and is formally defined in the following way. For two configurations $(s_1, x_1, s_2, x_2, \ldots, s_n, x_n)$ and $(p_1, y_1, p_2, y_2, \ldots, p_n, y_n)$

$$(s_1, x_1, s_2, x_2, \ldots, s_n, x_n) \vdash_r (p_1, y_1, p_2, y_2, \ldots, p_n, y_n)$$

is a computation step in the returning mode iff one of the following two conditions holds:

- (local computation step) $K \cap \{s_1, s_2, \ldots, s_n\} = \emptyset$, $x_i = a_i y_i$, $a_i \in \Sigma \cup \{\varepsilon\}$, and $p_i \in \delta_i(s_i, a_i)$ for all $1 \leq i \leq n$; or
- (communication step) $K \cap \{s_1, s_2, \ldots, s_n\} \neq \emptyset$, for all $1 \leq i \leq n$ for which $s_i = K_{j_i}$ and $s_{j_i} \notin K$, $p_i = s_{j_i}$ and $p_{j_i} = q_{j_i}$, for all the other $1 \leq r \leq n$, $p_i = s_i$, and for all $1 \leq t \leq n$, $y_i = x_i$.

The reflexive and transitive closures of the relations $\vdash$ and $\vdash_r$ are denoted by $\vdash^*$ and $\vdash_r^*$, respectively. A computation of a PCFA system is a sequence of computation steps and it holds that a pair of two configurations κ and κ' is an element of $\vdash^*$, $\vdash_r^*$ respectively, if and only if there exists a computation from κ to κ'. The initial configuration for a system of degree n and an input word w is $(q_1, w, q_2, w, \ldots, q_n, w)$. The languages that are accepted by a PCFA system $\mathcal{A}$ in the non-returning and the returning mode are defined by

$$\begin{aligned} L(\mathcal{A}) = \{w \in \Sigma^* \mid (q_1, w, q_2, w, \ldots, q_n, w) \vdash^* (s_1, \varepsilon, s_2, \varepsilon, \ldots, s_n, \varepsilon) \\ \text{with } s_i \in F_i \text{ for all } 1 \leq i \leq n\} \end{aligned}$$

and

$$\begin{aligned} L_r(\mathcal{A}) = \{w \in \Sigma^* \mid (q_1, w, q_2, w, \ldots, q_n, w) \vdash_r^* (s_1, \varepsilon, s_2, \varepsilon, \ldots, s_n, \varepsilon) \\ \text{with } s_i \in F_i \text{ for all } 1 \leq i \leq n\}. \end{aligned}$$

Due to the different possibilities of how to combine the properties of being centralized/non-centralized and returning/non-returning, we distinguish four different types of PCFA systems:

- $\mathsf{PCFA}(n)$ - non-centralized and non-returning PCFA systems of degree n,
- $\mathsf{CPCFA}(n)$ - centralized and non-returning PCFA systems of degree n,
- $\mathsf{RPCFA}(n)$ - non-centralized and returning PCFA systems of degree n, and

- $\mathsf{RCPCFA}(n)$ - centralized and returning PCFA systems of degree n.

The deterministic variants of the PCFA systems get the prefix 'D', i.e. DPCFA, DCPCFA, DRPCFA, and $\mathsf{DRCPCFA}$. The class of languages that are accepted by a PCFA system of type $\mathsf{X} \in \{\mathsf{PCFA}, \mathsf{CPCFA}, \mathsf{RPCFA}, \mathsf{RCPCFA}, \mathsf{DPCFA}, \mathsf{DCPCFA}, \mathsf{DRPCFA}, \mathsf{DRCPCFA}\}$ and degree n is denoted by $\mathcal{L}(\mathsf{X}(n))$. The class of languages that can be accepted by a PCFA system of type X with arbitrarily many components is denoted by

$$\mathcal{L}(\mathsf{X}) = \bigcup_{n \geq 1} \mathcal{L}(\mathsf{X}(n)).$$

In the following example from [MMM02] a system $\mathcal{A}$ is given that accepts the language $L_{a^n b^n c^n}$. This language is not context-free, which gives a first impression of the computational power of PCFA systems.

Example 2. A system that accepts $L_{a^n b^n c^n}$ consists of two components that basically work as follows: the first component waits, while the second component moves its head over the a's to the beginning of the b's. Then both components compare the number of a's and the number of b's just by moving their heads synchronously. If the number of a's equals the number of b's, then A_1 reaches the b's at the same time that A_2 reaches the c's. The number of b's and c's are compared in the same way. In the whole computation both components work synchronously, and A_1 requests the state of A_2 after each local computation step. Since A_2 stores the information about its own currently read symbol in its state, A_1 knows the current symbol of A_2 in each step.
The system is given by

$$\mathcal{A} = (\{a, b, c\}, A_1, A_2, \{K_1, K_2\})$$

with the two components

$$\begin{aligned} A_1 &= (\{q_1, q_2, q_f, s_1, s_2, s_f, K_2\}, \{a, b, c\}, \delta_1, q_1, \{q_f\}) \text{ and} \\ A_2 &= (\{q_1, q_2, q_f, s_1, s_2, s_f\}, \{a, b, c\}, \delta_2, q_2, \{s_f\}), \end{aligned}$$

where δ_1 and δ_2 are defined as follows:

$$\begin{aligned}
\delta_1(q_1, \varepsilon) &= K_2, & \delta_2(q_2, a) &= q_2, \\
\delta_1(s_1, a) &= K_2, & \delta_2(q_2, b) &= s_1, \\
\delta_1(q_2, \varepsilon) &= K_2, & \delta_2(s_1, b) &= s_1, \\
\delta_1(s_2, b) &= K_2, & \delta_2(s_1, c) &= s_2, \\
\delta_1(s_f, c) &= q_f, & \delta_2(s_2, c) &= s_2, \\
\delta_1(q_f, c) &= q_f, & \delta_2(s_2, \varepsilon) &= s_f, \\
& & \delta_2(s_f, \varepsilon) &= s_f.
\end{aligned}$$

A computation of $\mathcal{A}$ for an input word $a^nb^nc^n$ proceeds as follows:

$$\begin{array}{ll|ll}
 & (q_1,\ a^nb^nc^n,\ q_2,\ a^nb^nc^n) & \vdash & (K_2,\ b^nc^n,\ s_2,\ c^{n-1}) \\
\vdash & (K_2,\ a^nb^nc^n,\ q_2,\ a^{n-1}b^nc^n) & \vdash & (s_2,\ b^nc^n,\ s_2,\ c^{n-1}) \\
\vdash & (q_2,\ a^nb^nc^n,\ q_2,\ a^{n-1}b^nc^n) & \vdash^{2(n-1)} & (s_2,\ bc^n,\ s_2,\ \varepsilon) \\
\vdash^{2(n-1)} & (q_2,\ a^nb^nc^n,\ q_2,\ b^nc^n) & \vdash & (K_2,\ c^n,\ s_f,\ \varepsilon) \\
\vdash & (K_2,\ a^nb^nc^n,\ s_1,\ b^{n-1}c^n) & \vdash & (s_f,\ c^n,\ s_f,\ \varepsilon) \\
\vdash & (s_1,\ a^nb^nc^n,\ s_1,\ b^{n-1}c^n) & \vdash & (q_f,\ c^{n-1},\ s_f,\ \varepsilon) \\
\vdash^{2(n-1)} & (s_1,\ ab^nc^n,\ s_1,\ c^n) & \vdash^{n-1} & (q_f,\ \varepsilon,\ s_f,\ \varepsilon)
\end{array}$$

□

The following facts result immediately from the definition:

1. $\mathcal{L}(\mathsf{CPCFA}(n)) \subseteq \mathcal{L}(\mathsf{PCFA}(n))$, $\mathcal{L}(\mathsf{RCPCFA}(n)) \subseteq \mathcal{L}(\mathsf{RPCFA}(n))$, $\mathcal{L}(\mathsf{DCPCFA}(n)) \subseteq \mathcal{L}(\mathsf{DPCFA}(n))$, and $\mathcal{L}(\mathsf{DRCPCFA}(n)) \subseteq \mathcal{L}(\mathsf{DRPCFA}(n))$,
2. $\mathcal{L}(\mathsf{X}(n)) \subseteq \mathcal{L}(\mathsf{X}(n+1))$ for all $\mathsf{X} \in \{\mathsf{PCFA}, \mathsf{CPCFA}, \mathsf{RPCFA}, \mathsf{RCPCFA}, \mathsf{DPCFA}, \mathsf{DCPCFA}, \mathsf{DRPCFA}, \mathsf{DRCPCFA}\}$.[4]

The authors of [MMM02] showed that non-centralized and non-returning PCFA systems of degree n have the same computational power as one-way n-head automata in the deterministic and in the nondeterministic case:

$$\mathcal{L}(\mathsf{DPCFA}(n)) = \mathcal{L}(\mathsf{1\text{-}DFA}(n)) \text{ and } \mathcal{L}(\mathsf{PCFA}(n)) = \mathcal{L}(\mathsf{1\text{-}NFA}(n)).$$

A direct consequence of this result and the fact that multi-head pushdown automata satisfy the semi-linearity property is that all types of PCFA systems can only accept semi-linear languages. Later, Choudhary, Krithivasan, and Mitrana showed in [CKM07] that the second of the above equalities also holds for nondeterministic, non-centralized, and returning PCFA systems:

$$\mathcal{L}(\mathsf{RPCFA}(n)) = \mathcal{L}(\mathsf{PCFA}(n)) = \mathcal{L}(\mathsf{1\text{-}NFA}(n)).$$

Thus, returning and non-returning systems are equally powerful in the nondeterministic and non-centralized version. Like in [MMM02] the authors proved this by simulating a one-way multi-head finite automaton. The question, whether this equation also holds for deterministic non-centralized PCFA systems was answered by Bordihn, Kutrib, and Malcher in [BKM08]:

$$\mathcal{L}(\mathsf{DRPCFA}(n)) = \mathcal{L}(\mathsf{DPCFA}(n)) = \mathcal{L}(\mathsf{1\text{-}DFA}(n)).$$

[4] Whether these inclusions form a strict hierarchy we will see below.

For the proof the authors defined and used the 'cycling-token-method', where an information token is cyclically communicated within the system. Moreover, they proved that in the deterministic non-returning case, centralized systems are weaker than non-centralized systems:

$$\mathcal{L}(\mathsf{DCPCFA}) \subset \mathcal{L}(\mathsf{DPCFA})$$

and that deterministic systems are less powerful than nondeterministic systems:

$$\mathcal{L}(\mathsf{DCPCFA}) \subset \mathcal{L}(\mathsf{CPCFA}) \text{ and } \mathcal{L}(\mathsf{DPCFA}) \subset \mathcal{L}(\mathsf{PCFA}).$$

For returning systems the proper inclusion

$$\mathcal{L}(\mathsf{DRPCFA}) \subset \mathcal{L}(\mathsf{RPCFA})$$

is immediately obtained due to the above connections and

$$\mathcal{L}(\mathsf{DRCPCFA}) \subset \mathcal{L}(\mathsf{RCPCFA})$$

can be shown similarly to the proof in [BKM08]: the complement of the language of palindromes L_{pal} over the alphabet $\{a, b\}$, i.e. $\overline{L_{pal}}$, is accepted by an $\mathsf{RCPCFA}(3)$-system (see Example 3 below). Suppose $\overline{L_{pal}}$ is also accepted by any $\mathsf{DRCPCFA}$-system with arbitrarily many components, then $\overline{L_{pal}} \in \mathcal{L}(\mathsf{1\text{-}DFA})$. Since $\mathcal{L}(\mathsf{1\text{-}DFA})$ is closed under complementation [Ros66], $L_{pal} \in \mathcal{L}(\mathsf{1\text{-}DFA})$. This is a contradiction, because it is well-known that even $L_{pal} \notin \mathcal{L}(\mathsf{1\text{-}NFA})$ [WW86]. Thus, $L_{pal} \notin \mathsf{DRCPCFA}$.

Example 3 (RCPCFA-system for $\overline{L_{pal}} = \{a, b\}^* \setminus \{w \mid w \in \{a, b\}^* \text{ and } w = w^R\}$). In Example 2 we presented a centralized system working in non-returning mode. Now, we define a centralized system

$$\mathcal{A} = (\{a, b\}, A_1, A_2, A_3, \{K_1, K_2, K_3\})$$

working in returning mode and accepting the complement of the language of all palindromes over the alphabet {a,b}, i.e. $\overline{L_{pal}}$. A word $w = w_1 w_2 \ldots w_l$ is contained in $\overline{L_{pal}}$ if and only if $l \geq 2$ and there exists an integer i with $\lceil \frac{l}{2} \rceil < i \leq l$ such that $w_i \neq w_{l-i+1}$.

The components of the system $\mathcal{A}$ behave as follows: at the beginning of a computation the first component only performs ε-steps and remains in its initial state q_0 without moving its head, while A_2 and A_3 move their heads in each step one position to the right. Thereby in each step A_2 stores the last symbol read in the

current state, i.e. it changes into q_a if it reads an a or into q_b reading the symbol b. Subsequently, after i steps A_2 is in state q_{w_i}. After exactly i ε-steps A_1 requests the current state of A_2 that contains the symbol w_i by reaching nondeterministically the communication state K_2. From here the second component is not of interest anymore: it just reads the rest of the input, and performs ε-steps in an accepting state. After the communication between A_1 and A_2 the former automaton moves its head one position to the right in each computation step. After exactly $l-i+1$ steps it reaches the communication state K_3 due to a nondeterministic decision. Thus, A_1 reaches state K_3 in $i+l-i+1=l+1$ local steps. Since component A_3 moves its head in each step one position to the right, it has read the entire input after l steps. With an additional ε-step it reaches a dedicated stated q_f that is communicated to A_1. In the last phase of the computation A_1 reads the remaining $i-1$ input symbols and $\mathcal{A}$ accepts.
Formally, the components of $\mathcal{A}$ are given by:

$$\begin{aligned}
A_1 &= (\{q_0, K_2, q_a, q_b, q_f\}, \{a, b\}, \delta_1, q_0, \{q_f\}),\\
A_2 &= (\{p_0, q_a, q_b, p_e, \}, \{a, b\}, \delta_2, p_0, \{p_e\}), \text{ and}\\
A_3 &= (\{s_0, s_1, q_f, s_e\}, \{a, b\}, \delta_3, s_0, \{q_f, s_e, s_0\})
\end{aligned}$$

with

$$\begin{array}{ll}
\delta_1(q_0,\varepsilon) = \{q_0, K_2\}, & \delta_2(p_0,a) = \delta_2(q_a,a) = \delta_2(q_b,a) = \{q_a, p_e\},\\
\delta_1(q_a,b) = \{q_a, K_3\}, & \delta_2(p_0,b) = \delta_2(q_a,b) = \delta_2(q_b,b) = \{q_b, p_e\},\\
\delta_1(q_a,a) = \{q_a\}, & \delta_2(p_0,\varepsilon) = \delta_2(p_e,\varepsilon) = \{p_e\},\\
\delta_1(q_b,a) = \{q_b, K_3\}, & \\
\delta_1(q_b,b) = \{q_b\}, & \delta_3(s_0,a) = \delta_3(s_0,b) = \{s_0, s_1\},\\
\delta_1(q_f,a) = \delta_1(q_f,b) = \{q_f\}, & \delta_3(s_1,\varepsilon) = \{q_f\},\\
 & \delta_3(q_f,\varepsilon) = \delta_3(s_e,\varepsilon) = \delta_3(s_0,\varepsilon) = \{s_e\}.
\end{array}$$

A computation of $\mathcal{A}$ for a word $w \in \overline{L_{pal}}$ is given by:

$$\begin{array}{ll|ll}
\multicolumn{2}{c|}{\text{for } i<l:} & \multicolumn{2}{c}{\text{for } i=l:}\\
 & (q_0, w, p_0, w, s_0, w) & & (q_0, w, p_0, w, s_0, w)\\
\vdash^{i-1} & (q_0, w, q_{w_{i-1}}, w_i \dots w_l, s_0, w_i \dots w_l) & \vdash^{l-1} & (q_0, w, q_{w_{l-1}}, w_l, s_0, w_l)\\
\vdash & (K_2, w, q_{w_i}, w_{i+1} \dots w_l, s_0, w_{i+1} \dots w_l) & \vdash & (K_2, w, q_{w_l}, \varepsilon, s_1, \varepsilon)\\
\vdash & (q_{w_i}, w, p_0, w_{i+1} \dots w_l, s_0, w_{i+1} \dots w_l) & \vdash & (q_{w_l}, w, p_0, \varepsilon, s_1, \varepsilon)\\
\vdash^{l-i} & (q_{w_i}, w_{l-i+1} \dots w_l, p_e, \varepsilon, s_1, \varepsilon) & \vdash & (K_3, w_2 \dots w_l, p_e, \varepsilon, q_f, \varepsilon)\\
\vdash & (K_3, w_{l-i+2} \dots w_l, p_e, \varepsilon, q_f, \varepsilon) & \vdash & (q_f, w_2 \dots w_l, p_e, \varepsilon, s_0, \varepsilon)\\
\vdash & (q_f, w_{l-i+2} \dots w_l, p_e, \varepsilon, s_0, \varepsilon) & \vdash^{l-1} & (q_f, \varepsilon, p_e, \varepsilon, s_e, \varepsilon)\\
\vdash^{i-1} & (q_f, \varepsilon, p_e, \varepsilon, s_e, \varepsilon) & &
\end{array}$$

Observe that although the components can nondeterministically choose the next state, there is only one correct choice for A_2 and A_3 in each accepting computation. If they choose wrongly by executing an ε-move in the middle of the input word (instead of at the end), then the remaining part of the input cannot be read anymore and thus the system does not accept. Moreover, in any accepting computation there are two situations where A_1 performs nondeterministic choices: guess the position i and guess the position $l - i + 1$. If the second guess is wrong, i.e. it does not match the first guess, then A_1 does not obtain q_f from A_3, because A_3 is only in q_f after exactly $l + 1$ local computation steps. If the first guess is wrong and the second guess fits to the first one, i.e. $w_i = w_{l-i+1}$, then A_1 is not allowed to switch into K_3 in $l + 1$ steps due to third and the fifth transitions. In both cases A_1 does not accept.
Obviously, if $w \notin \overline{L_{pal}}$, then A_1 cannot guess successfully and thus cannot reach q_f. Therefore, in this case $\mathcal{A}$ does not accept w. □

In Figure 3.3 we summarize the relations between the language classes that are characterized by multi-head finite automata and PCFA systems. An arrow denotes a proper inclusion and a dotted arrow denotes an inclusion that is not yet known to be proper.

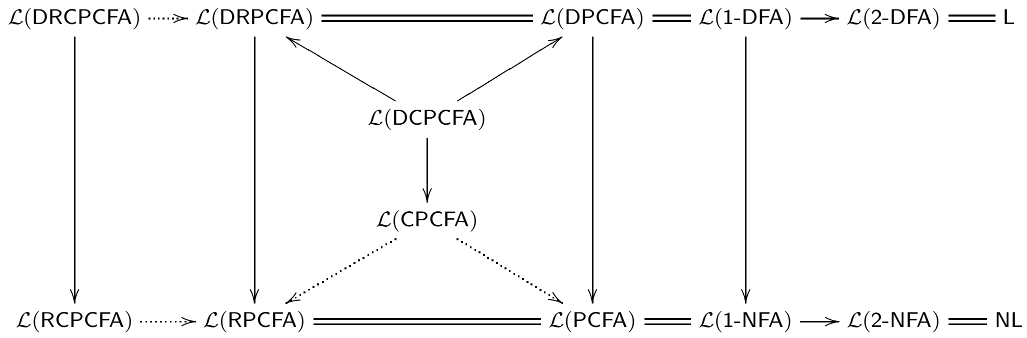

Figure 3.3: Relations between language classes characterized by PCFA systems and multi-head finite automata.

Besides the consideration of the computational power of the various types of PCFA systems, typical decidability problems were investigated in [MM01, BKM10]. Though almost all questions are decidable for single finite automata (see e.g. [HU79]), recent results show that most typical problems are not even semi-decidable even for the weakest types of PCFA systems [BKM11]. Table 3.1 gives an overview of some problems and these results can immediately be carried over to the language classes of the remaining types of PCFA systems.

Dealing with systems of several components, another interesting question is whether systems with more components are more powerful than systems with less

	$\mathcal{L}(\mathsf{CPCFA}(n))$	$\mathcal{L}(\mathsf{RCPCFA}(n))$	$\mathcal{L}(\mathsf{DCPCFA}(n))$	$\mathcal{L}(\mathsf{DRCPCFA}(n))$
emptiness	$n \geq 2$	$n \geq 3$	$n \geq 2$	$n \geq 3$
universality	$n \geq 2$	$n \geq 2$	$n \geq 2$	$n \geq 3$
inclusion	$n \geq 2$	$n \geq 2$	$n \geq 2$	$n \geq 3$
equivalence	$n \geq 2$	$n \geq 2$	$n \geq 2$	$n \geq 3$
finiteness	$n \geq 2$	$n \geq 3$	$n \geq 2$	$n \geq 3$
infiniteness	$n \geq 2$	$n \geq 3$	$n \geq 2$	$n \geq 3$
regularity	$n \geq 2$	$n \geq 3$	$n \geq 2$	$n \geq 3$
context-freeness	$n \geq 2$	$n \geq 3$	$n \geq 2$	$n \geq 3$

Table 3.1: Undecidability results for PCFA systems. All problems are not even semi-decidable for the stated number of components.

components. For non-centralized PCFA systems this follows immediately from the equivalence with multi-head automata and the hierarchy results that are already known for multi-head automata:

$$\mathcal{L}(\mathsf{X}(n)) \subset \mathcal{L}(\mathsf{X}(n+1)) \text{ for all } n \geq 1 \text{ and } \mathsf{X} \in \{\mathsf{PCFA}, \mathsf{RPCFA}, \mathsf{DPCFA}, \mathsf{DRPCFA}\}.$$

In [BKM11] it is proved that there exists an infinite strict hierarchy also for centralized non-returning PCFA systems:

$$\mathcal{L}(\mathsf{X}(n)) \subset \mathcal{L}(\mathsf{X}(n+1)) \text{ for all } n \geq 1 \text{ and } \mathsf{X} \in \{\mathsf{CPCFA}, \mathsf{DCPCFA}\}.$$

For centralized systems working in the returning mode this is still an open question.

In the next section we will have a look at a more powerful variant of PC systems, namely parallel communicating pushdown automata systems. These systems use a quite different communication protocol: instead of communicating states as in PCFA systems, the components send their stack content by request. It turns out that also these systems are related to well-known language classes of the Chomsky-hierarchy.

3.3 Parallel communicating pushdown automata systems

Parallel communicating pushdown automata systems (we will abbreviate them as PCPA systems) were introduced by Csuhaj-Varjú et al. in the year 2000 [CMMV00] (see also [MM00] for a more compact introduction). Such a system consists of several pushdown automata (the components) that are able to communicate with each other. In contrast to PCFA systems, PCPA systems are defined to communicate via requesting and sending the content of their stacks. Although a communication by states is possible for PCPA systems as well, the authors of [CMMV00] stated that such systems can obviously simulate two-stack automata. Figure 3.4 shows the schematical construction of a PCPA system.

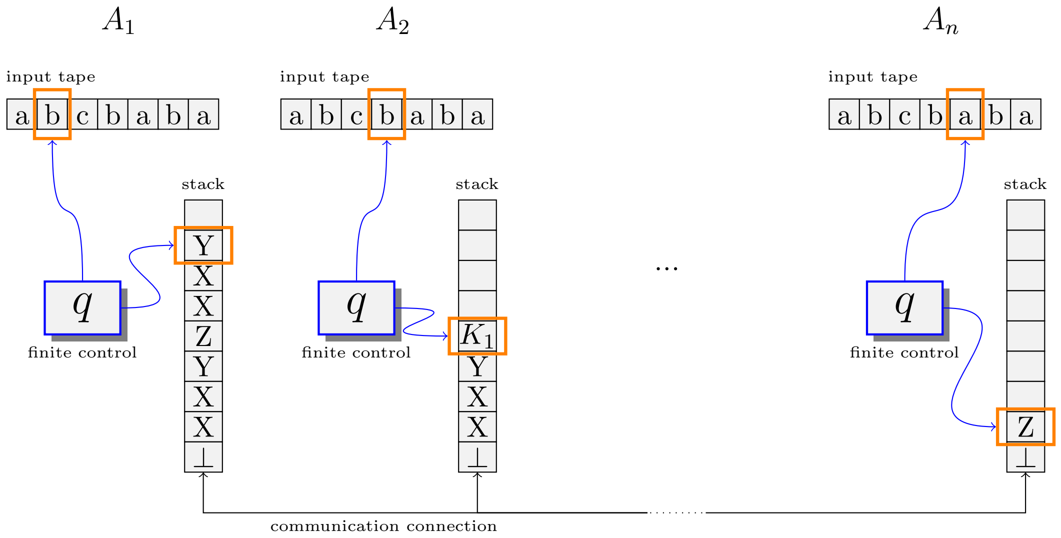

Figure 3.4: Schematic representation of a PCPA system.

Formally, a *parallel communicating pushdown automata system* $\mathcal{A}$ of degree n is an $(n+3)$-tuple

$$\mathcal{A} = (V, \Delta, A_1, A_2, \dots, A_n, K),$$

where V is the input alphabet, Δ is the stack alphabet, $K \subseteq \{K_1, K_2, \dots, K_n\} \subseteq \Delta$ is the set of query symbols, and for all $1 \leq i \leq n$,

$$A_i = (Q_i, V, \Delta, \delta_i, q_i, Z_i, F_i)$$

is a pushdown automaton, where

- Q_i is the set of states,
- V is the input alphabet,
- Δ is the stack alphabet,
- δ_i is the transition mapping from $Q_i \times (V \cup \{\varepsilon\}) \times \Delta$ into $\mathcal{P}(Q_i \times \Delta^*)$,
- $q_i \in Q_i$ is the initial state,
- $Z_i \in \Delta$ is the bottom symbol of the stack, and
- $F_i \subseteq Q_i$ is the set of final states.

A configuration of a PCPA system of degree n is a $3n$-tuple

$$(s_1, x_1, \alpha_1, s_2, x_2, \alpha_2, \ldots, s_n, x_n, \alpha_n),$$

where for all $1 \leq i \leq n$, $s_i \in Q_i$ is the current state of A_i, $x_i \in V^*$ is the remaining part of the input word that is not yet read by A_i, and $\alpha_i \in \Delta^*$ is the current stack content of A_i with the first letter being the topmost symbol of the stack. For an input word $w \in V^*$, the initial configuration is

$$(q_1, w, Z_1, q_2, w, Z_2, \ldots, q_n, w, Z_n)$$

such that each component starts with the same input on its input tape and with only the bottom symbol on its stack.

For two configurations $(s_1, x_1, B_1\alpha_1, \ldots, s_n, x_n, B_n\alpha_n)$ and $(p_1, y_1, \beta_1, \ldots, p_n, y_n, \beta_n)$ with $B_i \in \Delta$ and $\alpha_i, \beta_i \in \Delta^*$ for all $1 \leq i \leq n$,

$$(s_1, x_1, B_1\alpha_1, \ldots, s_n, x_n, B_n\alpha_n) \vdash (p_1, y_1, \beta_1, \ldots, p_n, y_n, \beta_n)$$

is a computation step iff one of the following two conditions holds:

- (local computation step)
  $K \cap \{B_1, B_2, \ldots, B_n\} = \emptyset$ and for all $1 \leq i \leq n$:
  $x_i = a_i y_i$, $a_i \in V \cup \{\varepsilon\}$, $(p_i, \beta_i') \in \delta_i(s_i, a_i, B_i)$, and $\beta_i = \beta_i'\alpha_i$; or
- (communication step)
  - for all $1 \leq i \leq n$ with $B_i = K_{j_i}$ and $B_{j_i} \notin K$: $\beta_i = B_{j_i}\alpha_{j_i}\alpha_i$,
  - for all other $1 \leq r \leq n$: $\beta_r = B_r\alpha_r$, and
  - for all $1 \leq t \leq n$: $y_t = x_t$ and $p_t = s_t$.

Whenever a communication can be performed this has a higher priority than the local computation. Thus, if the topmost stack symbol of any component is a query symbol, then a communication step has to be executed and no component is allowed to perform a local computation step. Moreover, a component can send its stack content only if the topmost symbol of its own stack is not a query symbol. In that case this communication has to be resolved first. If there is a circular query, then the system is blocked. Another strategy for the communication is the returning mode (the previous strategy is called non-returning mode). Here, after communicating the stack content, the stack of the requested component is emptied and the bottom symbol is put onto it. Thus, the requested stack is initialized like in the very beginning of the computation. Formally,

$$(s_1, x_1, B_1\alpha_1, \ldots, s_n, x_n, B_n\alpha_n) \vdash_r (p_1, y_1, \beta_1, \ldots, p_n, y_n, \beta_n)$$

is a computation step iff one of the following two conditions holds:

- (local computation step)
  $K \cap \{B_1, B_2, \ldots, B_n\} = \emptyset$ and for all $1 \leq i \leq n$:
  $x_i = a_i y_i$, $a_i \in V \cup \{\varepsilon\}$, $(p_i, \beta_i') \in \delta_i(s_i, a_i, B_i)$, and $\beta_i = \beta_i'\alpha_i$; or
- (communication step in the returning mode)
  - for all $1 \leq i \leq n$ with $B_i = K_{j_i}$ and $B_{j_i} \notin K$: $\beta_i = B_{j_i}\alpha_{j_i}\alpha_i$ and $\beta_{j_i} = Z_{j_i}$,
  - for all other $1 \leq r \leq n$: $\beta_r = B_r\alpha_r$, and
  - for all $1 \leq t \leq n$: $y_t = x_t$ and $p_t = s_t$.

A computation of a PCPA system is denoted by $\vdash^*$ in the non-returning mode and $\vdash_r^*$ in the returning mode that are the reflexive and transitive closures of $\vdash$ and $\vdash_r$, respectively. The language that is accepted by a PCPA system $\mathcal{A}$ is defined as

$$L(\mathcal{A}) = \{w \in V^* \mid (q_1, w, Z_1, q_2, w, Z_2, \ldots, q_n, w, Z_n) \vdash^* (s_1, \varepsilon, \alpha_1, \ldots, s_n, \varepsilon, \alpha_n) \text{ with } s_i \in F_i \text{ for all } 1 \leq i \leq n\}$$

in the non-returning mode and

$$L_r(\mathcal{A}) = \{w \in V^* \mid (q_1, w, Z_1, q_2, w, Z_2, \ldots, q_n, w, Z_n) \vdash_r^* (s_1, \varepsilon, \alpha_1, \ldots, s_n, \varepsilon, \alpha_n) \text{ with } s_i \in F_i \text{ for all } 1 \leq i \leq n\}$$

in the returning mode.

If only one component is allowed to request the stack content of any other component (i.e. only this component is allowed to write query symbols on the stack), then the system is called *centralized* and this component is called the *master* of the system. For simplicity it is often assumed that the first component is the master within a centralized system.

To distinguish the different types of PCPA systems we use the following notation[5]:

- $\mathsf{PCPA}(n)$: PCPA system of degree n,

[5] Here we differ from the notation in [CMMV00], where the different types are written in lowercase and the according language classes are written in capital letters.

- $\mathsf{CPCPA}(n)$: centralized PCPA system of degree n,
- $\mathsf{RPCPA}(n)$: returning PCPA system of degree n,
- $\mathsf{RCPCPA}(n)$: returning centralized PCPA system of degree n.

The class of all languages that are accepted by any PCPA system of type $\mathsf{X} \in \{\mathsf{PCPA}(n), \mathsf{CPCPA}(n), \mathsf{RPCPA}(n), \mathsf{RCPCPA}(n)\}$ is denoted by $\mathcal{L}(\mathsf{X})$.

The authors of [CMMV00] established the following results. Non-centralized systems are at least as powerful as centralized systems with the same number of components in the returning mode as well as in the non-returning mode:

$$\mathcal{L}(\mathsf{CPCPA}(n)) \subseteq \mathcal{L}(\mathsf{PCPA}(n)) \text{ and } \mathcal{L}(\mathsf{RCPCPA}(n)) \subseteq \mathcal{L}(\mathsf{RPCPA}(n)) \text{ for all } n \geq 1.$$

Systems of degree 1 characterize exactly the context-free languages and systems with more components are at least as powerful as systems with less components:

$$\begin{gathered}\mathcal{L}(\mathsf{X}(1)) = \mathsf{CFL} \text{ and } \mathcal{L}(\mathsf{X}(n)) \subseteq \mathcal{L}(\mathsf{X}(n+1)) \\ \text{for all } \mathsf{X} \in \{\mathsf{PCPA}, \mathsf{CPCPA}, \mathsf{RPCPA}, \mathsf{RCPCPA}\} \text{ and } n \geq 1.\end{gathered}$$

Whether the hierarchy is strict for any of the above types is still an open problem.

Investigating the computational power of PCPA systems, it is proved in [CMMV00] that each non-returning PCPA system can be simulated by a returning system with twice as many components in the non-centralized case:

$$\mathcal{L}(\mathsf{PCPA}(n)) \subseteq \mathcal{L}(\mathsf{RPCPA}(2n)) \text{ for all } n \geq 2.$$

Furthermore, they proved that already two components are enough for non-returning and non-centralized systems to accept all recursively enumerable languages. In the case of returning non-centralized systems three components are used. For the proof a two-stack automaton is simulated that characterizes the language class RE [Har78]:

$$\mathcal{L}(\mathsf{PCPA}(2)) = \mathcal{L}(\mathsf{RPCPA}(3)) = \mathsf{RE}.$$

For centralized systems a lower bound of the computational power is given by multi-head pushdown automata [CMMV00] (these are automata with one stack and multiple one-way read-only heads on the input tape, see e.g. [HI68] for further details): each language accepted by an n-head pushdown automaton is accepted by a centralized PCPA system with n components in the returning and in the non-returning case. Moreover, any n-pushdown automaton can be simulated by a centralized PCPA system with $n+1$ components in the non-returning mode. An n-pushdown automaton uses one input tape with a one-way head and n stacks, see e.g. [BCCC96].

Later, Balan, Krithivasan, and Madhu proved in [BKM03] that even centralized PCPA systems with three components accept all recursively enumerable languages in the non-returning case:

$$\mathcal{L}(\mathsf{CPCPA}(3)) = \mathsf{RE}.$$

Similar to the proofs in [CMMV00] they simulate a two-stack automaton with a centralized PCPA system of degree 3. Moreover, the authors introduce two new communication protocols for PCPA systems: 1) filtered communication and 2) communicating a constant amount of stack symbols. The communication filter is realized by a projection for each component that determines which symbols of the according stack content are transmitted and which are not. In a communication step a requesting component replaces the query symbol only with the transmitted symbols, i.e. only with those symbols that were not filtered out. In the returning-mode only the transmitted symbols are deleted from the requested stack, such that the filtered symbols remain in the stack of the requested component. It was shown that PCPA systems with two components and communication filter are computationally complete independent of the system's type:

$$\mathcal{L}(\mathsf{CPCPAF}(2)) = \mathcal{L}(\mathsf{PCPAF}(2)) = \mathcal{L}(\mathsf{RCPCPAF}(2)) = \mathcal{L}(\mathsf{RPCPAF}(2)) = \mathsf{RE},$$

where the 'F' in the name of the type signalizes the usage of a communication filter. With the other communication protocol, namely that only the topmost k symbols of the stack are transmitted for a constant k has lead to a similar result. The authors of [BKM03] proved that

$$\mathcal{L}(\mathsf{PCPA}(2,k)) = \mathcal{L}(\mathsf{RPCPA}(2,k)) = \mathcal{L}(\mathsf{CPCPA}(3,k)) = \mathcal{L}(\mathsf{RCPCPA}(2,k)) = \mathsf{RE}$$

for all $k \geq 1$, where $\mathsf{X}(n,k)$ denotes the type of the according PCPA system of degree n that communicates only the topmost k stack symbols. Moreover, they stated that even $\mathcal{L}(\mathsf{CPCPA}(2,k)) = \mathsf{RE}$ holds.

In 2009 M. S. Balan [Bal09] gave an answer to the question for the computational power of the remaining type of PCPA systems, namely the centralized returning PCPA systems. He claimed that any RCPCPA system of degree n can be simulated by an n-head pushdown automaton (n-PDA). As mentioned above, the opposite direction was already proved in [CMMV00]. Unfortunately, the proof of Balan is not correct. This was recently shown in [Ott12]. Moreover, the claim does not hold either as it was recently shown in [Pet12]. For this, Petersen constructed a RCPCPA-system of degree two that accepts the exponential language $\{a^{2^n} \mid n \geq 1\}$ and used the fact that each unary language that is accepted by a multi-head pushdown automaton must be regular [HI68]. In the same work, Petersen showed that a $\mathsf{RCPCPA}(2)$-system can simulate a one register machine with the operations multiplication and conditional division by 2 or 3. Thus, centralized and returning PCPA systems of degree two are universal. This also improves the result for non-centralized returning systems. In summery, we have:

$$\mathcal{L}(\mathsf{RCPCPA}(2)) = \mathcal{L}(\mathsf{RPCPA}(2)) = \mathsf{RE}.$$

3.4 Parallel communicating Watson-Crick automata systems

In contrast to the variants of automata considered above, the Watson-Crick finite automaton is biochemically inspired. It was introduced in [FPRS97] as a theoretical approach for DNA computing. Basically, a *Watson-Crick finite automaton* (WK automaton, for short[6]) consists of a finite control and a Watson-Crick input tape that has two tracks of the same length lying one upon the other. On each track there is a one-way read-only head and both heads can be moved independently from the left-hand side of the tape to the right. The strings on the two tracks represent a double-stranded DNA molecule[7], and each two symbols that stand at the same position (one stands above the other) are combined by a complementarity relation. In DNA molecules both strands are connected by a fixed one-to-one complementarity relation called *Watson-Crick complementarity relation* with the pairing (A, T) and (C, G)[8]. Figure 3.5 illustrates the structure of a WK automaton.

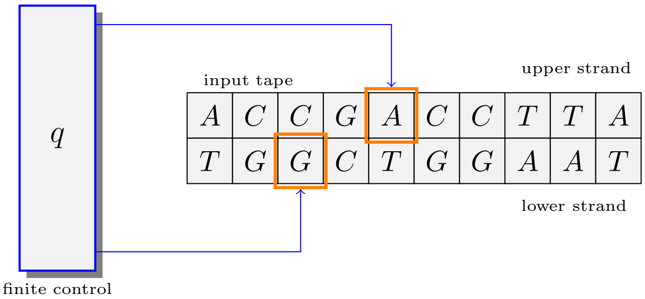

Figure 3.5: Schematic representation of a Watson-Crick finite automaton (following [PRS98], p. 154).

Although the Watson-Crick complementarity relation is determined by $\{(A, T),$

[6] The abbreviation WK is obtained by taking the first and the last letter of 'WATSON-CRICK' [FPRS97].

[7] For this model DNA molecules are idealised and seen as linear double-stranded sequences of nucleotides without loops and branches.

[8] A, C, G, and T stand for the four types of nucleotides DNA molecules consist of: adenine, cytosine, guanine, and thymine.

$(C, G)\}$, the definition of WK automata is generalized in the sense that arbitrary symmetric binary relations over a finite alphabet are allowed. Formally, a Watson-Crick finite automaton is a 6-tuple

$$A = (V, \rho, Q, q_0, F, \delta),$$

where V is the finite input alphabet, $\rho \subseteq V \times V$ is a symmetric relation representing the complementarity relation on V, Q is a finite set of states containing the initial state q_0, F is the set of final states with $F \subseteq Q$, and $\delta : Q \times (V^* \times V^*) \to \mathcal{P}(Q)$ is the transition mapping with $\delta(q, \binom{x}{y}) \neq \emptyset$ only for finitely many triples $(q, x, y) \in Q \times V^* \times V^*$. A transition $q \in \delta(p, \binom{x}{y})$ means that being in state p, reading x on the upper strand, and reading y on the lower strand, the automaton can change into state q and move the input heads to the right of x and y, respectively. In general, a WK automaton is nondeterministic. Whenever δ is a (partial) function to Q, the automaton is called deterministic.

Referring to the notation of double stranded DNA molecules the elements of $V^* \times V^*$ are usually written as $\binom{x}{y}$ instead of (x, y) and a transition $q \in \delta(p, \binom{x}{y})$ is written as a rewriting rule $p \binom{x}{y} \to \binom{x}{y} q$. By $\begin{bmatrix} x \\ y \end{bmatrix}$ well-formed double-stranded sequences are denoted, i.e. x and y have the same length and match according to the complementarity relation ρ (if $x = x_1 \dots x_n$ and $y = y_1 \dots y_n$, then for each $1 \leq i \leq n$, it holds that $(x_i, y_i) \in \rho$). Then, the Watson-Crick domain that is associated to ρ and V is defined as

$$\begin{aligned} WK_\rho(V) = \Big\{ \begin{bmatrix} a_1 a_2 \dots a_n \\ b_1 b_2 \dots b_n \end{bmatrix} \mid n \geq 0, a_1, \dots, a_n, b_1, \dots, b_n \in V, \\ (a_1, b_1), \dots, (a_n, b_n) \in \rho \Big\} \end{aligned}$$

and contains all well-formed double-stranded sequences over V.

A computation step of a WK automaton is of the form

$$\binom{x_1}{y_2} p \binom{x_2}{y_2} \binom{x_3}{y_3} \Rightarrow \binom{x_1}{y_1} \binom{x_2}{y_2} q \binom{x_3}{y_3},$$

and it is allowed if and only if $q \in \delta(p, \binom{x_2}{y_2})$, where $x_i, y_i \in V^*$ for $1 \leq i \leq 3$ and $\begin{bmatrix} x_1 x_2 x_3 \\ y_1 y_2 y_3 \end{bmatrix} \in WK_\rho(V)$. The reflexive and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. The language that is accepted by a Watson-Crick automaton A is

$$L(A) = \left\{ x \in V^* \mid q_0 \begin{bmatrix} x \\ y \end{bmatrix} \Rightarrow^* \begin{bmatrix} x \\ y \end{bmatrix} q, q \in F, \begin{bmatrix} x \\ y \end{bmatrix} \in WK_\rho(V) \right\}.$$

The class of languages that are accepted by any Watson-Crick finite automaton is denoted by $\mathcal{L}(\mathsf{WK})$.

The following example is taken from [CC06a]. It shows how a deterministic WK automaton A accepts the language $L_{a^nb^nc^n}$. Observe the similarity between the definition and the behaviour of A, the 2-head automaton in Example 1, and the PCFA system constructed in Example 2.

Example 4. A WK automaton A accepting the language $L_{a^nb^nc^n}$ is constructed as follows: $A = (\{a, b, c\}, \rho, \{q_0, q_1, q_2, q_f\}, q_0, \{q_f\}, \delta)$. Here $\rho = \{(a, a), (b, b), (c, c)\}$, and δ is given through:

$$\delta(q_0, \binom{a}{\varepsilon}) = q_0, \qquad \delta(q_1, \binom{b}{a}) = q_1, \qquad \delta(q_2, \binom{c}{b}) = q_2,$$

$$\delta(q_0, \binom{b}{a}) = q_1, \qquad \delta(q_1, \binom{c}{b}) = q_2, \qquad \delta(q_2, \binom{\varepsilon}{c}) = q_f,$$

$$\delta(q_f, \binom{\varepsilon}{c}) = q_f.$$

First, the head on the upper strand reads over all a's until it reaches the first b. Then, the number of b's of the upper strand is compared with the number of a's of the lower strand by moving both heads synchronously. The number of b's and c's are compared in the same manner. At the end, the second head is moved to the right hand end of the input and the word is accepted if and only if it is of the form $a^nb^nc^n$. □

The complementarity relation is said to be injective if each symbol $a \in V$ has a unique complementary symbol $b \in V$ with $(a, b) \in \rho$. It is known that this is not a restriction and, moreover, for each WK automaton A with an arbitrary complementarity relation, there exists a WK automaton A' with the identity complementarity relation, i.e. $\rho = \{(a, a) \mid a \in V\}$, such that $L(A) = L(A')$ [KW05]. A WK automaton with the identity complementarity relation can be interpreted just as an automaton with a tape with one track and two one-way read-only heads. Indeed, Watson-Crick finite automata have the same computational power as one-way 2-head finite automata [FPRS97].

In the literature several modifications and extensions of Watson-Crick automata have been studied, for instance: stateless WK automata, where $Q = F = \{q_0\}$ [FPRS97], Watson-Crick ω-automata [Pet03], Watson-Crick finite transducers [FPRS97], WK automata with an additional Watson-Crick memory [PRS98]. Based on the fact that the two strands of a DNA molecule have opposite 5'-3' orientations, reverse Watson-Crick finite automata are investigated (also called $5' \to 3'$ Watson-Crick automata [FPRS97, LN10]). There, the two heads start on opposite ends of the input and one head moves from the left to the right and the other one moves from the right to the left. Reverse Watson-Crick automata are also considered in the sensing case, i.e. the computation of an automaton ends when both heads meet each other [Nag08].

In [FPRS97] several variants of WK automata are introduced and compared with each other according to their computational power. Moreover, different characterizations of recursively enumerable languages by WK languages can be found there. In addition, results about deterministic WK automata and about their descriptional complexity are presented in [CCKS09].

Besides the Watson-Crick complementarity relation parallelism plays a major rule in DNA computing. Accordingly, *parallel communicating Watson-Crick automata systems* (PCWK systems, for short) were introduced in [CC05] and investigated in [CC06b, CC06c, CC06a]. Similar to PCFA systems, a PCWK system is a finite collection of Watson-Crick finite automata, where each automaton works on its own input tape and communicates through sending its current state on request. Figure 3.6 shows the basic structure of a PCWK system. Formally, a PCWK system of degree n (abbreviated by $\mathsf{PCWK}(n)$) is an $n+3$-tuple

$$\mathcal{A} = (V, \rho, A_1, A_2, \ldots, A_n, K),$$

where V is the finite input alphabet, ρ is the complementarity relation, $A_i = (V, \rho, Q_i, q_i, F_i, \delta_i)$ are the components of $\mathcal{A}$, $1 \leq i \leq n$, and $K \subseteq \{K_1, K_2, \ldots, K_n\}$ is the set of communication states with $K \subseteq \bigcup_{i=1}^{n} Q_i$.

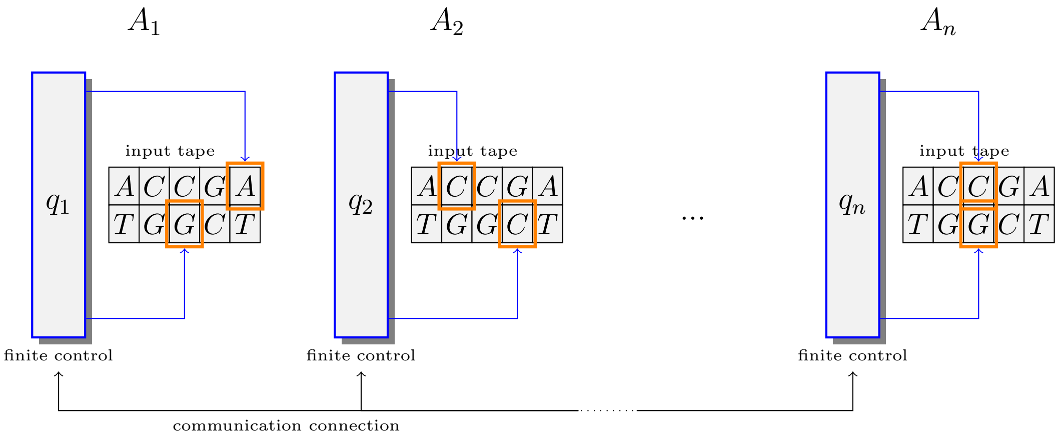

Figure 3.6: Schematic representation of a PCWK system with components A_1, A_2, ..., A_n.

A configuration of a PCWK system $\mathcal{A}$ of degree n is a $2n$-tuple

$$\left(p_1, \binom{u_1}{v_1}, p_2, \binom{u_2}{v_2}, \ldots, p_n, \binom{u_n}{v_n}\right),$$

where p_i is the current state of component A_i and $\binom{u_i}{v_i}$ is the remaining part of the input that has not yet been read by component A_i, $1 \leq i \leq n$. Then, a computation step of $\mathcal{A}$ is of the form

$$\left(p_1, \binom{u_1}{v_1}, p_2, \binom{u_2}{v_2}, \ldots, p_n, \binom{u_n}{v_n}\right) \vdash \left(r_1, \binom{x_1}{y_1}, r_2, \binom{x_2}{y_2}, \ldots, r_n, \binom{x_n}{y_n}\right)$$

and can be executed by $\mathcal{A}$ if and only if one of the following two conditions holds:

1. $K \cap \{p_1, p_2, \ldots, p_n\} = \emptyset$, $\binom{u_i}{v_i} = \binom{z_i}{z_i'}\binom{x_i}{y_i}$, and $r_i \in \delta_i(p_i, \binom{z_i}{z_i'})$ for all $i \in \{1, \ldots, n\}$ (local computation step);

2. for all $i \in \{1, \ldots, n\}$ such that $p_i = K_{j_i}$ and $p_{j_i} \notin K$, it holds that $r_i = p_{j_i}$. For all other $l \in \{1, \ldots, n\}$, $r_l = p_l$. Moreover, for all $i \in \{1, \ldots, n\}$, $\binom{x_i}{y_i} = \binom{u_i}{v_i}$ (communication step).

The reflexive and transitive closure of $\vdash$ is $\vdash^*$, and the language accepted by $\mathcal{A}$ is defined by

$$L(\mathcal{A}) = \Big\{ x \in V^* \mid \left(q_1, \begin{bmatrix} x \\ y \end{bmatrix}, q_2, \begin{bmatrix} x \\ y \end{bmatrix}, \ldots, q_n, \begin{bmatrix} x \\ y \end{bmatrix}\right) \vdash^*$$

$$\left(p_1, \begin{bmatrix} \varepsilon \\ \varepsilon \end{bmatrix}, p_2, \begin{bmatrix} \varepsilon \\ \varepsilon \end{bmatrix}, \ldots, p_n, \begin{bmatrix} \varepsilon \\ \varepsilon \end{bmatrix}\right), p_i \in F_i \text{ for all } i \in \{1, \ldots, n\} \Big\}.$$

The class of all languages that are accepted by any parallel communicating Watson-Crick automata system is denoted by $\mathcal{L}(\mathsf{PCWK})$. Moreover, by $\mathcal{L}(\mathsf{PCWK}_{in})$ we denote the class of languages that are accepted by any PCWK system with an injective complementarity relation.

The next example is taken from [CC06a] and shows how a non-injective complementarity relation can be used to accept the language $\{a^{2^n} \mid n \geq 1\}$.

Example 5. The idea for the construction of a PCWK system that accepts the language $\{a^{2^n} \mid n \geq 1\}$ uses the equation

$$2^n = 1 + 2^0 + 2^1 + 2^2 + \ldots + 2^{n-1}$$

and its representation in the complement *bcbbccccbbbbbbbbcc*... according to the complementarity relation $\{(a, b), (a, c)\}$. Thus, for each word of the form a^{2^n}, there exists the complement $bcb^2c^4 \ldots b^{n-1}$ for even n and $bcb^2c^4 \ldots c^{n-1}$ for odd n, respectively, such that $\begin{bmatrix} aa\ldots a \\ bcb^2 \ldots b^{n-1} \end{bmatrix}$ ($\begin{bmatrix} aa\ldots a \\ bcb^2 \ldots c^{n-1} \end{bmatrix}$) is the input of the system, i.e. the input of all components. If the number of a's is not a power of two, then the described complement does not exist. Thus, the system basically verifies whether the complement begins with bc and the blocks of b's and c's alternate,

such that beginning with the second block, each following block contains exactly twice as many symbols as the previous block. For doing this, two deterministic components are taken: A_1 and A_2. The first one reads the according previous block and A_2 reads the according following block. In each comparison step, A_1 reads one symbol and A_2 reads two symbols. Thus, to verify whether the following block has twice the length of the previous block, both components check if they reach the beginning of the new block at the same computation step.
Formally, $\mathcal{A}$ is defined by

$$\mathcal{A} = (\{a, b, c\}, \rho, A_1, A_2, \{K_1\})$$

with

$$\begin{array}{rcl} \rho & = & \{(a, b), (a, c)\}, \\ A_1 & = & (\{a, b, c\}, \rho, \{q_1, r_b, r_c, r_{bc}, r_{cb}\}, q_1, \{r_b, r_c, r_{bc}, r_{cb}\}, \delta_1), \\ A_1 & = & (\{a, b, c\}, \rho, \{q_2, s_1, r_b, r_c, r_{bc}, r_{cb}, f_2, K_1\}, q_2, \{f_2\}, \delta_2), \end{array}$$

where

$$\begin{array}{rrrr} \delta_1(q_1, \binom{a}{b}) = r_b, & \delta_1(r_b, \binom{a}{c}) = r_{bc}, & \delta_1(r_{bc}, \binom{a}{c}) = r_c, & \delta_1(r_{bc}, \binom{a}{b}) = r_{cb}, \\ \delta_1(r_b, \binom{a}{b}) = r_b, & \delta_1(r_c, \binom{a}{c}) = r_c, & \delta_1(r_c, \binom{a}{b}) = r_{cb}, & \delta_1(r_{cb}, \binom{a}{b}) = r_b, \end{array}$$

$$\begin{array}{rrrr} \delta_2(q_2, \binom{aa}{bc}) = s_1, & \delta_2(s_1, \binom{\varepsilon}{\varepsilon}) = K_1, & \delta_2(r_{bc}, \binom{\varepsilon}{\varepsilon}) = f_2, & \delta_2(r_{cb}, \binom{\varepsilon}{\varepsilon}) = f_2, \\ \delta_2(f_2, \binom{\varepsilon}{\varepsilon}) = f_2, & \delta_2(r_{bc}, \binom{aa}{bb}) = K_1, & \delta_2(r_c, \binom{aa}{bb}) = K_1, & \delta_2(r_b, \binom{aa}{cc}) = K_1, \\ \delta_2(r_{cb}, \binom{aa}{cc}) = K_1. & & & \end{array}$$

□

It is known that each PCWK system of degree n with injective complementarity relation can be simulated by a one-way $2n$-head finite automaton and hence by a PCFA system of degree $2n$ [CC06c]:

$$\mathcal{L}(\mathsf{PCWK}_{in}(n)) \subseteq \mathcal{L}(\mathsf{1\text{-}NFA}(2n)) = \mathcal{L}(\mathsf{PCFA}(2n)).$$

Although for single Watson-Crick automata it has no effect whether the complementarity relation is injective or not, PCWK systems with non-injective complementarity relation are strictly more powerful than PCWK systems with injective complementarity relation [CC06b]. The PCWK system in Example 5 has a non-injective complementarity relation and accepts the non-semi-linear exponential language. On the other hand it is well-known that one-way multi-head automata only accept semi-linear languages.

That PCWK systems are strictly more powerful than single Watson-Crick automata even with injective complementarity relation was proved in [CC05]. There, a system is given for the language $\{a^n b^m c^m d^n \mid n, m \geq 1\}$ that cannot be accepted by any single Watson-Crick automaton. An upper bound for $\mathcal{L}(\mathsf{PCWK})$ is CSL, since each PCWK system can be simulated by a linear bounded automaton [CC05]:

$$\mathcal{L}(\mathsf{PCWK}) \subseteq \mathsf{CSL}.$$

If we consider only languages over a one-letter alphabet, then PCWK systems with injective complementarity relation can only accept regular languages [CC05]. This follows by the facts that each such language L can be represented as $L = h(L')$, where h is a morphism, and L' is a unary language that is accepted by a WK automaton, each one-letter language accepted by a WK automaton is regular [FPRS97], and the regular languages are closed under morphisms.

Furthermore, it is known that $\mathcal{L}(\mathsf{PCWK})$ is closed under the following operations [CC06c]. Let $L_1, L_2 \in \mathcal{L}(\mathsf{PCWK})$ over an input alphabet V, $\# \notin V$ be a new symbol, and $(\#, \#) \in \rho$. Then $(L_1 \cdot \{\#\}) \cap (L_2 \cdot \{\#\})$, $L_1 \cdot \{\#\} \cdot L_2 \cdot \{\#\}$, and $(L_1 \cdot \{\#\})^*$ are contained in $\mathcal{L}(\mathsf{PCWK})$.

3.5 Parallel communicating grammar systems

Parallel communicating grammar systems are shortly presented here, since they are chronologically the first kind of PC systems and play an important role within this research area. In contrast to PC automata systems, PC grammar systems are language generating devices instead of language accepting ones. They were introduced in [PS89] and intensively studied in [CDKP94].

A *parallel communicating grammar system* $\mathcal{G}$ of degree n is an $(n+3)$-tuple

$$\mathcal{G} = (N, K, T, G_1, \ldots, G_n),$$

where N is the finite set of nonterminal symbols, K is the set of so-called query symbols, T is the finite set of terminal symbols, and $G_1, \ldots, G_n$ are usual phrase structure grammars with

$$G_i = (N \cup K, T, P_i, S_i)$$

for all $1 \leq i \leq n$, also called the components of $\mathcal{G}$. The set of query symbols K contains elements of the form $Q_1, Q_2, \ldots, Q_n$, one query symbol for each component. To describe derivations of $\mathcal{G}$, *configurations* are defined as n-tuples $(x_1, \ldots, x_n)$, where $x_i \in (N \cup K \cup T)^*$ is a sentential form of component G_i for all $1 \leq i \leq n$.

As we have already seen in the case of PC automata systems, a global clock is used to synchronize the computation of the components. The grammars of $\mathcal{G}$ work together in the following way. Each starts its derivation from the start axiom S_i as usual. Then, in each global step, all G_i execute one local derivation step in parallel provided that no query symbol appears in a sentential form of any component. If there is a query symbol in some sentential form, then a communication step has to take place. In a communication step of the system all query symbols within the sentential forms are replaced by the corresponding sentential forms, i.e. a query symbol Q_j within the sentential form of G_i is replaced by the whole sentential form of G_j. This replacing can be done only if the sentential form of G_j does not contain query symbols itself. Such query symbols are not replaced in the current step but possibly in one of the next communication steps. As long as there are query symbols left in any sentential form, no local derivation is allowed. If there occurs a cyclic query, then the whole system is blocked. The description above demonstrates how components of a PC grammar system communicate with each other. They exchange information by sending their generated strings on request.

Similar to PC automata systems two different modes of communication steps can be defined: returning and non-returning. In the former case the grammars that have sent their sentential forms continue the derivation from their start axioms. In the latter case they continue from their unchanged sentential forms. Formally, these two types of derivation steps of a PC grammar system are defined as follows. For two configurations $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$, a derivation step $(x_1, \ldots, x_n) \Rightarrow (y_1, \ldots, y_n)$ is allowed if and only if one of the following two conditions hold:

- For all $1 \leq i \leq n$, it holds either
  - $x_i \in T^*$ and $y_i = x_i$ (terminal word), or
  - $|x_i|_K = 0$, $|x_i|_V > 0$, and $x_i \Rightarrow y_i$ in G_i (local derivation step).
- Communication step. For the non-returning case:
  For all x_i of the form $x_i = z_1 Q_{i_1} z_2 Q_{i_2} \ldots z_t Q_{i_t} z_{t+1}$, $1 \leq i \leq n$, with $t > 0$ and $z_j \in (N \cup T)^*$ for all $1 \leq j \leq t+1$, such that $|x_{i_j}|_K = 0$, $1 \leq j \leq t$: $y_i = z_1 x_{i_1} z_2 x_{i_2} \ldots z_t x_{i_t} z_{t+1}$. For all non-specified y_s, $y_s = x_s$.
  For the returning case:
  For all x_i of the form $x_i = z_1 Q_{i_1} z_2 Q_{i_2} \ldots z_t Q_{i_t} z_{t+1}$, $1 \leq i \leq n$, with $t > 0$ and $z_j \in (N \cup T)^*$ for all $1 \leq j \leq t+1$, such that $|x_{i_j}|_K = 0$, $1 \leq j \leq t$: $y_i = z_1 x_{i_1} z_2 x_{i_2} \ldots z_t x_{i_t} z_{t+1}$ and $y_{i_j} = S_{i_j}$, $1 \leq j \leq t$. For all non-specified y_s, $y_s = x_s$.

The reflexive and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$ and describes arbitrary long derivations. With $\Rightarrow^k$ we denote derivations of length k. Now, the language that is generated by a PC grammar system Γ is defined by

$$L(\Gamma) = \{w \in T^* \mid (S_1, S_2, \ldots, S_n) \Rightarrow^* (w, \alpha_2, \ldots, \alpha_n), \alpha_i \in (N \cup T \cup K)^*, 2 \leq i \leq n\}.$$

Example 6. [CDKP94] Consider the PC grammar system

$$\Gamma = (\{S_1, S_2, S_3\}, \{Q_1, Q_2, Q_3\}, \{a, b, c, d\}, G_1, G_2, G_3)$$

with the sets of rules

$$\begin{aligned} P_1 &= \{S_1 \to aS_1, S_1 \to aQ_2, S_3 \to d\}, \\ P_2 &= \{S_2 \to bS_2, S_2 \to bQ_3\}, \\ P_3 &= \{S_3 \to cS_3\} \end{aligned}$$

for the regular components G_1, G_2, G_3. At the beginning of each derivation of the grammar system, G_1, G_2, and G_3 generate exactly one a, b, and c, respectively, with their according first rule:

$$(S_1, S_2, S_3) \Rightarrow^k (a^k S_1, b^k S_2, c^k S_3).$$

At one point in time, G_1 and G_2 apply their second rule in order to query the sentential form of G_2 and G_3, respectively. In doing so, G_1 generates another a, G_2 generates another b, and G_3 generates another c:

$$(a^k S_1, b^k S_2, c^k S_3) \Rightarrow (a^{k+1} Q_2, b^{k+1} Q_3, c^{k+1} S_3).$$

It is important that G_1 and G_2 apply their second rule at the same time, otherwise the system is blocked. Thereafter, two communication steps are executed:

$$(a^{k+1} Q_2, b^{k+1} Q_3, c^{k+1} S_3) \Rightarrow (a^{k+1} Q_2, b^{k+1} c^{k+1} S_3, S_3) \Rightarrow (a^{k+1} b^{k+1} c^{k+1} S_3, S_2, S_3).$$

Here, the system works in returning mode, thus after the communications the sentential forms of G_2 and G_3 are reset to S_2 and S_3, respectively. Working in non-returning mode the system is blocked, since there is no rule for S_3 in P_2. In the last step G_1 generates the terminal word $a^{k+1} b^{k+1} c^{k+1} d$ by using its third rule. Hence, the generated language of Γ is $L(\Gamma) = \{a^n b^n c^n d \mid n \geq 1\}$. □

A PC grammar system is called *centralized* if only the first component is allowed to introduce query symbols. All the other components are not allowed to request the sentential form of any other component. It is known that in case of returning systems with regular or linear grammars, centralized systems are weaker than non-centralized systems.

Additionally, unsynchronized grammar systems were considered, where each component is allowed to wait although it could apply a local derivation step. Each unsynchronized system can be simulated by a synchronized system by just introducing chain rules for each nonterminal symbol. Moreover, unsynchronized systems are properly weaker than synchronized systems in the case of centralized and returning grammar systems with regular or linear components. These systems are even not more powerful than the individual grammars, which means they can only generate regular or linear languages, respectively. This shows that synchronization is in fact a powerful mechanism.

Further, it is known that there exist infinite hierarchies according to the number of components for centralized and returning systems with regular or linear components.

Since this work is not dedicated to grammar systems, the reader is referred to the annotated bibliography of Erzsébet Csuhaj-Varjú and György Vaszil [CV] for more information about parallel communicating grammar system.

4 Restarting Automata

The model of the restarting automaton was introduced by Petr Jančar, František Mráz, Martin Plátek, and Jörg Vogel in 1995 [JMPV95] as a variant of the forgetting automaton [JMP93b, JMP93a]. It is motivated by the linguistic technique of *analysis by reduction* that is used for checking the syntactical correctness of a sentence of a natural language. For that purpose a given sentence is simplified stepwise by deleting or rewriting selected parts until a simple sentence is obtained whose correctness can be checked immediately. This simplification is also called *reduction*. An essential property of this method is that a deletion or rewriting may not change the correctness of the sentence. Thus, if the sentence is correct before a reduction step, then it has to be correct afterwards, too. If the sentence is not correct, then the modified sentence must not be correct either. This property is called *correctness preserving property* and *error preserving property*, respectively. The following example from [JMPV95] demonstrates how a syntactical error can be detected using analysis by reduction:

Example 7 (Analysis by reduction). Consider the sentence

The little boys I mentioned runs very quickly.

In the first step, the word 'little' can be deleted without changing the syntactical correctness of the sentence:

The boys I mentioned runs very quickly.

During the next steps the parts 'I mentioned'[1], 'The', 'very', and 'quickly' are erased successively:

The boys runs very quickly,
boys runs very quickly,
boys runs quickly,
boys runs.

After the last reduction step we obtain a simple sentence, where the syntactical error can be immediately detected. □

Observe that for each reduction step the information about the previous steps are unimportant and thus can be forgotten. Moreover, we can interpret each

[1] Here we swapped the second and the third step of the original example for a better reading.

reduction step as a ‘restart’ of the processing with a new input sentence. Now we define the restarting automaton whose behaviour represents the ‘reduction by analysis’.

A restarting automaton consists of a finite state control and a working tape with a read/write window of a fixed size (see Figure 4.1). At the beginning of each computation the tape contains an input word (that is, what we called ‘sentence’ above) enclosed by the left sentinel ¢ and the right sentinel $, and the window is positioned on the leftmost border of the tape. The sentinels can only be read but they are not allowed to be erased or rewritten within the whole computation. Furthermore, the tape is flexible, that means, if a symbol is deleted from the tape, then the corresponding cell is cut out such that no empty cells can occur. During the computation the automaton can move the window across the tape, then replace the currently read string and at last perform a restart operation, that is, the finite state control is set into the initial state and the window is placed on the leftmost border of the tape.

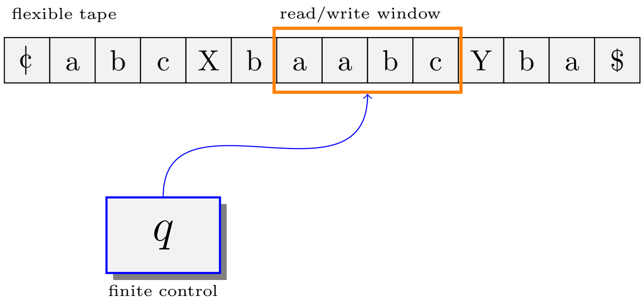

Figure 4.1: Schematic representation of a restarting automaton.

Over the years various types of restarting automata were considered and different definitions have been used. First, we define the most general case of the restarting automaton, the two-way restarting automaton (RLWW-automaton for short), and then we describe the various types with their according restrictions. Thereby we lean on the definition of [Ott06].

Definition 1. An RLWW-automaton M is an 8-tuple

$$M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta),$$

where

- Q is a non-empty finite set of states,
- Σ is a non-empty finite input alphabet,

- $\Gamma \supseteq \Sigma$ is a finite tape alphabet (the symbols from $\Gamma \setminus \Sigma$ are called *auxiliary symbols*),
- $¢, \$ \notin \Gamma$ are the left and the right sentinels,
- $q_0 \in Q$ is the initial state,
- $k \geq 1$ is the size of the read/write window, and
- $\delta : Q \times \mathcal{PC}^{(k)} \to \mathcal{P}_e((Q \times (\{\mathsf{MVR}, \mathsf{MVL}\} \cup \mathcal{PC}^{\leq(\mathsf{k}-1)})) \cup \{\mathsf{Restart}, \mathsf{Accept}\})$ is the transition relation, where the set $\mathcal{PC}^{(k)}$ contains all possible window contents and is formally defined as

  $$\mathcal{PC}^{(i)} = (\{¢\} \cdot \Gamma^{i-1}) \cup \Gamma^i \cup (\Gamma^{\leq(i-1)} \cdot \{\$\}) \cup (\{¢\} \cdot \Gamma^{\leq(i-2)} \cdot \{\$\}) \quad (i \geq 0),$$

  $$\Gamma^{\leq n} = \bigcup_{i=0}^{n} \Gamma^i, \text{ and } \quad \mathcal{PC}^{\leq(k-1)} = \bigcup_{i=0}^{k-1} \mathcal{PC}^{(i)}.$$

  To avoid confusion when speaking about several automata with different tape alphabets and window sizes, we extend this notation and associate it with the name of the automaton: $\mathcal{PC}_M$. □

In general, a restarting automaton is nondeterministic. If δ is a (partial) function from $Q \times \mathcal{PC}^{(k)}$ into $(Q \times (\{\mathsf{MVR}, \mathsf{MVL}\} \cup \mathcal{PC}^{\leq(k-1)})) \cup \{\mathsf{Restart}, \mathsf{Accept}\}$, then the automaton is deterministic.

A *configuration* κ describes an instantaneous situation of an automaton and is either of the type $\kappa = \mathsf{Accept}$, what we call an *accepting configuration*, or it is written as a string $\kappa = \alpha q \beta$ with $q \in Q$ 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 the finite control, $\alpha\beta$ is the current tape content, and the window is currently positioned on the first symbol of β. That means that the window contains the first k symbols of β if $|\beta| \geq k$, or otherwise it contains all symbols of β including the right end marker \$, that is, all symbols of the right end of the tape. For an input word $w \in \Sigma^*$ and the initial state q_0, the configuration $q_0 ¢ w \$$ is called an *initial configuration.* Any configuration of the form $q_0 ¢ u \$$ with $u \in \Gamma^*$ is called a *restarting configuration.*

To describe the window content of an automaton within a particular configuration, the *prefix function* $\pi_k : \Sigma^* \to \Sigma^*$ over the alphabet Σ is introduced:

$$\pi_k(w_1 w_2 \ldots w_l) = \begin{cases} w_1 w_2 \ldots w_l, & \text{if } l < k, \\ w_1 w_2 \ldots w_k, & \text{otherwise,} \end{cases}$$

with $w_1, w_2, \ldots, w_l \in \Sigma$. Figure 4.2 shows how π_k is used to denote the currently read window content of a component for a given configuration. If the window size

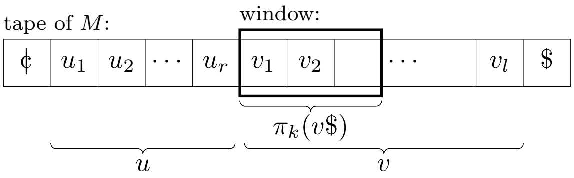

Figure 4.2: The prefix function π_k determines the currently read window content of the component M for the current configuration $¢uqv\$$ (q is the current state).

is not given explicitly, then we also use π_M for an automaton M.

For two configurations κ_1 and κ_2 of an automaton M, $\kappa_1 \vdash_M \kappa_2$ is a *computation step*, if one of the following five conditions holds:

1. Move-right step: the automaton moves its window exactly one position to the right and changes from state p into state q, that is, $\kappa_1 = \alpha pa\beta$, $\kappa_2 = \alpha aq\beta$, and $(q, \mathsf{MVR}) \in \delta(p, \pi_k(a\beta))$ with either

   (a) $\alpha = \varepsilon$, $a = ¢$, and $\beta \in \Gamma^* \cdot \{\$\}$; or

   (b) $\alpha \in \{¢\} \cdot \Gamma^*$, $a \in \Gamma$, and $\beta \in \Gamma^* \cdot \{\$\}$.

   Observe that the automaton is not allowed to move the window to the right if it is positioned on the rightmost border of the tape ($a = \$$ and $\beta = \varepsilon$).

2. Move-left step: the automaton moves its window exactly one position to the left and changes from state p into state q, that is, $\kappa_1 = \alpha ap\beta$, $\kappa_2 = \alpha qa\beta$, and $(q, \mathsf{MVL}) \in \delta(p, \pi_k(\beta))$ with either

   (a) $\alpha = \varepsilon$, $a = ¢$, and $\beta \in \Gamma^* \cdot \{\$\}$; or

   (b) $\alpha \in \{¢\} \cdot \Gamma^*$, $a \in \Gamma$, and $\beta \in \Gamma^* \cdot \{\$\}$.

   Particularly the automaton is not allowed to move the window over the left border of the tape ($\alpha = a = \varepsilon$ and $\beta \in \{¢\} \cdot \Gamma^* \cdot \{\$\}$).

3. Rewrite step: the automaton replaces the currently read window content by a shorter string, changes from state p into state q, and sets the window directly to the right of the previously written substring, that is, $\kappa_1 = \alpha p\beta_1\beta_2$, $\kappa_2 = \alpha\beta_1' q\beta_2$, and $(q, \beta_1') \in \delta(p, \beta_1)$ with $|\beta_1'| < |\beta_1| = k$, and either

   a) $\alpha = \varepsilon$, $\beta_1 \in \{¢\} \cdot \Gamma^*$, $\beta_2 \in \Gamma^* \cdot \{\$\}$, and $\beta_1' \in \{¢\} \cdot \Gamma^*$; or

   b) $\alpha \in \{¢\} \cdot \Gamma^*$, $\beta_1 \in \Gamma^*$, $\beta_2 \in \Gamma^* \cdot \{\$\}$, and $\beta_1' \in \Gamma^*$.

   If the rewriting takes place at the right end of the tape (the \$-symbol appears in the window), then the window may not move over the \$-symbol afterwards. Thus, it is positioned directly on the \$-symbol. That is, $\kappa_1 = \alpha p\beta\$$,

$\kappa_2 = \alpha\beta' q\$$, and $(q, \beta'\$) \in \delta(p, \beta\$)$ with $1 \leq |\beta| \leq k-1$, $|\beta'| < |\beta|$, and either

c) $\alpha = \varepsilon$ and $\beta, \beta' \in \{¢\} \cdot \Gamma^*$, or

d) $\alpha \in \{¢\} \cdot \Gamma^*$ and $\beta, \beta' \in \Gamma^*$.

It is further required that between two rewrite steps a restart operation is executed.

4. Restart step: the automaton can perform a restart operation, whereby the finite control is set into the initial state q_0 and the window is placed back on the leftmost position of the tape, that is, $\kappa_1 = \alpha p \beta$, $\kappa_2 = q_0 \alpha\beta$, and $\mathsf{Restart} \in \delta(p, \pi_k(\beta))$.

5. Accept step: the automaton can accept the input word by changing into the accepting configuration, that is, $\kappa_1 = \alpha p \beta$, $\kappa_2 = \mathsf{Accept}$, and $\mathsf{Accept} \in \delta(p, \pi_k(\beta))$.

The reflexive and transitive closure of the binary relation $\vdash_M$ describes the set of all possible *computations* of M of an arbitrary length and is denoted by $\vdash_M^*$. To denote that a computation consists of exactly r computation steps we use $\vdash_M^r$. If it is clear, to which automaton the relation is referring, then the name of the automaton in the subscript is omitted.

If in a particular situation δ is not defined for the current state or window content, then the automaton halts and therewith rejects the input. We call such configurations *rejecting configurations*. Thus, a computation can finish in two possible ways: accept the input word by reaching the accepting configuration or reject the input by reaching a rejecting configuration. Both kinds of configurations are also called *halting configurations*. Now we can define the language $L(M)$ that is accepted by an automaton M. It contains all words over the input alphabet Σ for which there exists a computation that starts with the corresponding initial configuration and reaches the accepting configuration:

$$L(M) = \{w \in \Sigma^* \mid q_0 ¢ w \$ \vdash_M^* \mathsf{Accept}\}.$$

Observe that in the rewrite step the new string β_1' has to be shorter than the old string β_1. This is called the *length reducing property* and ensures that the tape gets shorter with each rewrite step.

In principle, the behaviour of an automaton M can be described as follows: Beginning in a restarting configuration $q_0 ¢ u \$$ for any $u \in \Gamma^*$ (this includes particularly the initial configurations), M can move its window over the tape, then rewrites a part of the tape, can move the window again, and at last M performs a restart operation. Afterwards M is again in a restarting configuration and can thus repeat this sequence until it reaches an accepting or a rejecting configuration. The described sequence is called a *cycle*, and as mentioned above, in each cycle exactly one rewrite step is performed. Thus, a cycle can be formally written as a computation

$$q_0 ¢ u\$ = \kappa_1 \vdash_M^* \kappa_2 \vdash_M \kappa_3 \vdash_M^* \kappa_4 \vdash_M \kappa_5 = q_0 ¢ u'\$,$$

where the first part of the computation $\kappa_1 \vdash_M^* \kappa_2$ contains only move right steps and move left steps, then a rewrite step $\kappa_2 \vdash_M \kappa_3$ follows, the computation $\kappa_3 \vdash_M^* \kappa_4$ contains only move right steps and move left steps again, and the last step $\kappa_4 \vdash_M \kappa_5$ is a restart step. We use the notation $q_0 ¢ u\$ \vdash_M^c q_0 ¢ u'\$$, when the restarting configuration $q_0 ¢ u'\$$ is reached in one cycle from the restarting configuration $q_0 ¢ u\$$. For multiple cycles we use $\vdash_M^{c*}$. The *tail* of a computation is the last part of the computation that begins with a restarting configuration and reaches a halting configuration without performing a restart operation. Observe the similarity between a cycle of an automaton and a reduction step of the above explained technique 'analysis by reduction'.

In the next example an RLWW-automaton is presented that accepts the language

$$L_{a^nb^n} = \{a^nb^n \mid n \geq 0\}.$$

Example 8. An automaton M that accepts the language $L_{a^nb^n}$ can behave in the following way: It moves the window to the right over the ¢-symbol and the a's until it has found the substring ab. Then it deletes the ab from the tape and checks whether there are only b's to the right of the deleted substring. For that purpose it just moves the window to the right. Reaching the end of the tape (the \$-symbol appears in the window), it performs a restart operation and starts a new cycle with the shorter word. This continues until only the sentinels are left on the tape. Thus, if M reads ¢\$, then it accepts the input word. Formally, M is given by

$$M = (Q, \Sigma, \Gamma, ¢, \$, q_0, 2, \delta)$$

with $Q = \{q_0, q_1\}$ and $\Sigma = \{a, b\}$. Since M does not need auxiliary symbols, we take $\Gamma = \Sigma$. The initial state is q_0. The automaton needs to read and replace only substrings of length two in one step, therefore the window size is set to 2. At last,

δ is given by:

$$\begin{array}{ll}
\delta(q_0, \cent a) = (q_0, \mathsf{MVR}), & \delta(q_1, bb) = (q_1, \mathsf{MVR}), \\
\delta(q_0, aa) = (q_0, \mathsf{MVR}), & \delta(q_1, b\$) = (q_1, \mathsf{MVR}), \\
\delta(q_0, ab) \ = (q_1, \varepsilon), & \delta(q_1, \$) \ = \mathsf{Restart}, \\
\delta(q_0, \cent\$) = \mathsf{Accept}. &
\end{array}$$

A computation on the input word $aaabbb$ is formally written as follows (where the computation starts in the first column, proceeds downwards, is continued in the second column, and so forth):

$$\begin{array}{rl|rl|rl|rl}
 & q_0\cent aaabbb\$ & \vdash_M & q_0\cent aabb\$ & \vdash_M & q_0\cent ab\$ & \vdash_M & q_0\cent\$ \\
\vdash_M & \cent q_0aaabbb\$ & \vdash_M & \cent q_0aabb\$ & \vdash_M & \cent q_0ab\$ & \vdash_M & \mathsf{Accept.} \\
\vdash_M & \cent aq_0aabbb\$ & \vdash_M & \cent aq_0abb\$ & \vdash_M & \cent q_1\$ & & \\
\vdash_M & \cent aaq_0abbb\$ & \vdash_M & \cent aq_1b\$ & & & & \\
\vdash_M & \cent aaq_1bb\$ & \vdash_M & \cent abq_1\$ & & & & \\
\vdash_M & \cent aabq_1b\$ & & & & & & \\
\vdash_M & \cent aabbq_1\$ & & & & & &
\end{array}$$

Each of the first three columns of the computation represents one cycle and the last column corresponds to the tail of the computation.
If the input word is not of the form a^nb^n, that is, it either is not of the form a^*b^* (the a's and b's are mixed) or the numbers of a's and b's are different, then the automaton behaves as follows. In the first case it rejects the input in the first cycle because it reads an a when expecting only b's or vice versa. In the second case at one point of the computation only a's or only b's are on the tape and the automaton halts and rejects by reading $a\$$ or $\cent b$, respectively. □

Now, we give the different types of the restarting automaton model that result by adding several restrictions to the general model above. An RRWW-automaton is an RLWW-automaton that is only allowed to move its window to the right but not to the left. Therefore this type is also called *one-way restarting automaton.* If we claim that after a rewrite step a restart operation has to be applied immediately, then we obtain an RWW-automaton. In other words, this type of automaton is not allowed to move the window between a rewrite and a restart step. In some definitions for this type of automaton the rewrite and restart operations are merged to a single operation.

Furthermore, the rewrite operation can be restricted. An RLWW-automaton (RRWW-, RWW-automaton) is an RLW-automaton (RRW-, RW-automaton) if no auxiliary symbols are used, that is, if $\Sigma = \Gamma$. At last, an RLW-automaton (RRW-, RW-automaton) is an RL-automaton (RR-, R-automaton) if each rewrite operation is of the form $(q, \beta') \in \delta(p, \beta)$, where $|\beta'| < |\beta|$ and β' is a scattered subword of β. That means that β' can be obtained by deleting one or more symbols from β. For the simplification of further notations, we classify the types as follows:

$$\begin{aligned}\mathcal{T} &= \{RLWW, RLW, RL, RRWW, RRW, RR, RWW, RW, R\} \text{ and}\\ \mathcal{T}_R &= \{RRWW, RRW, RR, RWW, RW, R\}.\end{aligned}$$

All these types refer to nondeterministic automata. For deterministic automata we use the prefix 'det'. Thus, a deterministic automaton of type $\mathsf{X} \in \mathcal{T}$ is a det-X-automaton. The automaton that is given in Example 8 is of type det-RR. First, M is a one-way automaton that moves the window only to the right but not to the left. Second, it moves the window between the rewrite and the restart operations, thus it is not of the type RWW. Third, since no auxiliary symbols are used and each rewrite step is just a deletion, it is of the type RR. As the transition relation is a function and therewith M can perform in each situation at most one possible operation, it is deterministic.

Historically, the development of the various types of restarting automata was the other way round in contrast to our above explanation. The model that was introduced by Jancăr et al. in 1995 was exactly the R-automaton [JMPV95]. Then, in 1998 the usage of auxiliary symbols (RWW-automaton) and the separation of the rewrite step and the restart step (RRWW-automaton) were introduced by the same authors [JMPV98]. The two-way restarting automaton was then given by Martin Plátek in 2001 [Plá01].

The class of languages that are accepted by any automaton of type $(\mathsf{det}\text{-})\mathsf{X}$, $\mathsf{X} \in \mathcal{T}$, is denoted by $\mathcal{L}((\mathsf{det}\text{-})\mathsf{X})$. Sometimes it is useful to define the language class not only depending on the particular type but also on the window size. The set of languages accepted by restarting automata of type $(\mathsf{det}\text{-})\mathsf{X}$ with a window size of at most k is denoted by $\mathcal{L}((\mathsf{det}\text{-})\mathsf{X}(k))$.

A more compact representation of restarting automata is given in [NO01] by so-called *meta-instructions*. A meta-instruction is a triple of the form $(E_1, u \to v, E_2)$, where E_1 and E_2 are regular languages, and $u \to v$ is a rewriting rule with $|u| > |v|$, so that the substring u is rewritten by the word v in one rewrite step. This representation is based on the observation that an RRWW-automaton first reads the left part of the tape checking a regular constraint E_1, then performs a rewrite step, in which a word u is replaced by a shorter word v, and at last it can move the window over the remaining right part of the tape checking the regular constraint E_2. So a meta-instruction describes a cycle of an automaton. An accepting tail (rejecting tails need not be described) can be described with a meta-instruction of the form $(\cent E\$, \mathsf{Accept})$, where E is a regular language and we assume w.l.o.g. that the automaton accepts only while reading the right sentinel \$. Since an RWW-automaton performs a restart immediately after the rewrite step, the second constraint E_2 can be omitted for this type of automata. The automaton of Example 8 is described by the following two meta-instructions:

$$\begin{aligned}&(\cent a^*, ab \to \varepsilon, b^*\$),\\&(\cent\$, \mathsf{Accept}).\end{aligned}$$

Now, consider the following computation for an automaton M:

$$q_0\cent u_1v_1w_1\$ \vdash_M^c q_0\cent u_2v_2w_2\$ \vdash_M^c \ldots \vdash_M^c q_0\cent u_rv_rw_r\$.$$

We assume that the substring v_i is the part of the tape that is rewritten in the current cycle. We say that the computation is *monotone* if the right distances $D_i = |v_iw_i\$|$ never increase for $i = 1, 2, \ldots, r$. The tail of the computation is not of importance for monotonicity. An automaton is monotone if each computation beginning from an initial configuration is monotone. By mon-X (det-mon-X) we denote the type of (deterministic) monotone automata of type $\mathsf{X} \in \mathcal{T}$.

According to the computation given in Example 8, we have

$$q_0\cent aaabbb\$ \vdash_M^c q_0\cent aabb\$ \vdash_M^c q_0\cent ab\$ \vdash_M^c q_0\cent\$$$

with $D_1 = |abbb\$| = 5$, $D_2 = |abb\$| = 4$, and $D_3 = |ab\$| = 3$. Because of $D_1 \geq D_2 \geq D_3$, it follows that this computation is monotone. In fact, each computation of M, and thus M itself, is monotone. Hence, M is of the type det-mon-RR.

The property of monotonicity for restarting automata was defined in [JMPV95] and investigated in, e.g. [JMPV98, JMPV99, JMPV07, MO06, MPJV97]. Monotone automata gained an important role within the field of restarting automata, as they yield alternative characterizations of the context-free languages [JMPV99] and the deterministic context-free languages [JMPV97, JMPV99].

It turns out that some types of restarting automata characterize well-known language classes. We want to give some examples for the types presented above. Automata of type det-RR, R, RW, and RWW that have window size 1 accept exactly the set of regular languages [Mrá01, Rei07]:

$$\mathcal{L}(\mathsf{det\text{-}RR}(1)) = \mathcal{L}(\mathsf{R}(1)) = \mathcal{L}(\mathsf{RW}(1)) = \mathcal{L}(\mathsf{RWW}(1)) = \mathsf{REG}.$$

At first sight, this seems obvious due to the similarity between finite automata and the fact that auxiliary symbols do not help in the case of window size 1. Particularly, the restart operation is of no benefit in these cases. Then, Jančar, Mráz, Plátek, and Vogel showed in [JMPV97] and [JMPV99] that all deterministic and monotone one-way restarting automata accept exactly the class of deterministic context-free languages:

$$\mathcal{L}(\mathsf{det\text{-}mon\text{-}R}) = \mathcal{L}(\mathsf{det\text{-}mon\text{-}RRWW}) = \mathsf{DCFL}.$$

Omitting monotonicity and allowing the use of auxiliary symbols it turns out that

$$\mathcal{L}(\mathsf{det\text{-}RWW}) = \mathcal{L}(\mathsf{det\text{-}RRWW}) = \mathsf{CRL},$$

where CRL is the set of the Church-Rosser languages [NO03]. Further, the context-free languages (CFL) are characterized by nondeterministic monotone restarting automata with auxiliary symbols [JMPV99, Plá01]:

$$\mathcal{L}(\mathsf{mon\text{-}RWW}) = \mathcal{L}(\mathsf{mon\text{-}RRWW}) = \mathcal{L}(\mathsf{mon\text{-}RLWW}) = \mathsf{CFL}.$$

Moreover, it is known that the class of the growing context-sensitive languages (GCSL) is characterized by so-called weakly monotone RWW-, RRWW-, and RLWW-automata[2]. An automaton is called *weakly monotone* if there exists a constant such that the right distances of all computations increase by at most this constant (see [JLNO04] for more details).

All types of restarting automata are linearly bounded. Thus, all languages accepted by any type of restarting automata have to be contained in the set of the context-sensitive languages:

$$\mathcal{L}(\mathsf{RLWW}) \subseteq \mathsf{CSL}.$$

It is still an open question whether this inclusion is proper or not. For further information about the relationship between the language classes accepted by the various types of restarting automata, [Ott06] gives a useful overview and references.

Another property of restarting automata that we will use here is the *nonforgetting* property. Normally, after a restart operation a restarting automaton is reset into its initial state and the window is positioned back on the left side of the tape. Thus, after a restart, the automaton has 'forgotten' everything about the previous cycle. In contrast, a nonforgetting automaton is allowed to switch to any (determined) state during a restart operation. For this purpose, a restart

[2] For RWW- and RRWW-automata this was proved in [JLNO04]. For RLWW-automata this follows additionally from the proof of $\mathcal{L}(\mathsf{RRWW}){=}\mathcal{L}(\mathsf{RLWW})$ in [Plá01].

transition is extended as follows:

$$(\mathsf{Restart}, q) \in \delta(p, \alpha),$$

which means that if the automaton is in state p and reads currently the string α with its window, then it changes into state q and the window is positioned back on the left side of the tape. We will have a closer look at this kind of automata and the according language classes in the section ‘systems of nonforgetting restarting automata’.

5 Systems of parallel communicating restarting automata

5.1 Definitions

5.1.1 Systems of parallel communicating restarting automata

In this section systems of parallel communicating restarting automata and their functionality are defined and explained. Whenever it seems useful, reasons for and consequences of particular technical details are commented on.

Basically, a *parallel communicating restarting automata system* (PCRA system for short) of degree n is an n-tuple

$$\mathcal{M} = (M_1, M_2, \ldots, M_n),$$

where $M_1, M_2, \ldots, M_n$ are restarting automata which are the components of the system:

$$\begin{aligned} M_1 &= (Q_1, \Sigma, \Gamma_1, ¢, \$, q_1, k_1, \delta_1), \\ M_2 &= (Q_2, \Sigma, \Gamma_2, ¢, \$, q_2, k_2, \delta_2), \\ &\vdots \\ M_n &= (Q_n, \Sigma, \Gamma_n, ¢, \$, q_n, k_n, \delta_n). \end{aligned}$$

Let us explain how the components work together within a system. Following the classroom model described in Section 1, all components start the computation with the same task, i.e. the same input word on the tape. They perform local operations independently from each other just as it is usual for a single restarting automaton. At some point in time a component may need some help from another component, i.e. it requests some information, or it wants to inform another component about the results that it has achieved by its own previous local computation, i.e. it gives a response containing some information to a requesting component. This is exactly how the components work together within the system: they communicate with each other. After a communication between two components, they continue their local computations independently. In general, there is no restriction on how often and with which other components a component is allowed to communicate.

We now define the communication protocol that we use for PCRA systems and that is closely connected with the described interaction of components within the classroom model. Basically, the communication is realized by particular types of communication states that are included in the sets of states $Q_1, Q_2, \ldots, Q_n$: request states, response states, receive states, and acknowledge states. Formally, we denote the states as follows:

- request states (req^i_d): By entering the request state req^i_d, a component can send a communication request to the component M_i. The subscript d is a local information that allows to have more than one request state per component.
- response states ($\mathsf{res}^i_{d,c}$): By entering the response state $\mathsf{res}^i_{d,c}$, a component can answer to a request of M_i. The message can contain any information c with a constant length. The subscript d allows to enter different response states, but still sending the same answer to the same component.
- receive states ($\mathsf{rec}^i_{d,c}$): A component M_j is set into the receive state $\mathsf{rec}^i_{d,c}$ if it entered the request state req^i_d before and the component M_i has reached a response state $\mathsf{res}^j_{d',c}$.
- acknowledge states ($\mathsf{ack}^i_{d,c}$): A component M_j is set into the acknowledge state $\mathsf{ack}^i_{d,c}$ if the component M_i has reached a request state $\mathsf{req}^j_{d'}$ and M_j entered the response state $\mathsf{res}^i_{d,c}$.

A *communication step* can be executed if and only if a component M_i reaches a request state req^j_d through its local computation and the corresponding component M_j reaches a corresponding response state $\mathsf{res}^i_{d',c}$. For this purpose we introduce transitions of the form

$$\mathsf{req}^j_d \in \delta_i(q, \alpha) \quad \text{and} \quad \mathsf{res}^i_{d',c} \in \delta_j(q, \alpha).$$

A request state and response state correspond to each other if the superscript is the index of the corresponding other component. Whenever a component enters a request or response state, it waits for an answer from the communication partner. Hereby it is not important whether the request state or the response state is reached first. If a component is in a communication state but does never get a corresponding answer, i.e. the communication partner does not reach a corresponding communication state, then that component is blocked for the rest of the computation. Also with one or more blocked components the system continues the computation with the non-blocked components.

If two components reach corresponding communication states, then the communication step can be executed. We also say: the communication is resolved.

This means that the component M_i being in state req^j_d is set into the receive state $\mathsf{rec}^j_{d,c}$ (because it has *received* the requested message from M_j) and the component M_j being in state $\mathsf{res}^i_{d',c}$ is set into the acknowledge state $\mathsf{ack}^i_{d',c}$ (the receipt of the message is *acknowledged*). The c in the subscript of the receive state signalizes the receipt of the information c that was sent by M_j through entering the response state $\mathsf{res}^i_{d',c}$. In general, the receive and acknowledge states cannot be reached through a local computation, i.e. there do not exist transitions of the form $\mathsf{rec}^j_{d,c} \in \delta(q, \alpha)$ or $\mathsf{ack}^i_{d,c} \in \delta(q, \alpha)$, and no transitions of the form $A \in \delta(\mathsf{req}^j_d, \alpha)$ or $A \in \delta(\mathsf{res}^i_{d',c}, \alpha)$ are allowed. After M_i and M_j are set into the receive and acknowledge state, both components continue their independent local computations. The receive and acknowledge states are somehow successor states for the request and response states that are used for the formal description of a communication step.

The usage of the subscript d of the communication states has a particular reason. Consider, e.g. a communication step in a parallel communicating finite automata system (see Section 3.2 for further details), i.e. a component A_i has reached the communication state K_j and thus is set into the current state of A_j, say state q. Possibly, A_i can reach the state K_j for diffent prefixes of the input. Thus, after the communication, A_i is in state q and the information about the particular previously read part of the input is lost. However, according to the classroom model it seems somehow natural that a component can keep the whole information about its previous computation also in a communication step, such that it can combine the result of its own computation with the communicated information. For this, in PCRA systems the local information d is used.

Figure 5.1 demonstrates a communication between the components M_1 and M_2, where the downward arrows denote timelines. There, M_1 and M_2 perform their local computations independently of each other, which is denoted by the continuous downward arrows. At some point in time, M_1 reaches the request state req^2_d that can be interpreted as sending a request message to M_2 and keeping the local information d. On the other hand, M_2 reaches the response state $\mathsf{res}^1_{d',c}$ through its local computation. We can imagine, that with this M_2 sends a response message with the content c to M_1 and keeps the local information d'. Since both components reach corresponding communication states, the communication can be resolved and M_1 and M_2 are set into the states $\mathsf{rec}^2_{d,c}$ and $\mathsf{ack}^1_{d',c}$, respectively. If we imagine that the operation of setting M_1 into the receive state is connected with sending an acknowledge message to M_2, then we can observe a similarity to the so-called *three-way handshake* mechanism that is applied in communication networks for synchronization and secure communication, where secure means that both communication partners know whether the information was transmitted correctly or not. However, after the successful communication, M_1 and M_2 continue their local computations independently.

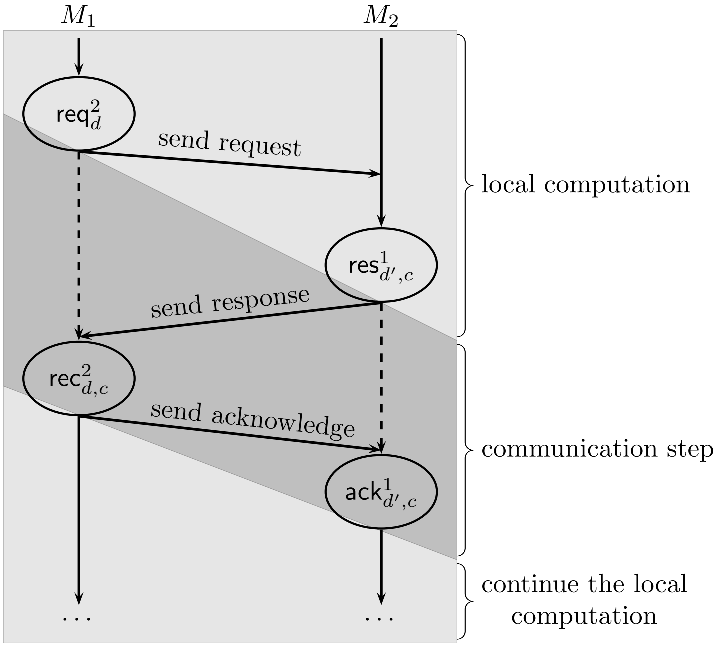

Figure 5.1: A communication between the two components M_1 and M_2.

As mentioned above, receive and acknowledge states cannot be reached through local computations in general. Nevertheless in some situations of proofs and constructions it may be useful to allow the direct reachability of receive and acknowledge states in a local computation without performing a communication step. Although this contradicts the basic idea of our communication concept, it does not effect a change of the computational power. A transition $\mathsf{rec}_{d,c}^i \in \delta_j(p, \alpha)$ or $\mathsf{ack}_{d,c}^i \in \delta_j(p, \alpha)$ can simply be replaced by the transitions $A \in \delta_j(p, \alpha)$ for all $A \in \delta_j(\mathsf{rec}_{d,c}^i, \alpha)$, for all $A \in \delta_j(\mathsf{ack}_{d,c}^i, \alpha)$ respectively. All transitions of the form $A \in \delta_j(\mathsf{rec}_{d,c}^i, \alpha)$, $A \in \delta_j(\mathsf{ack}_{d,c}^i, \alpha)$ respectively, have to be kept since the receive and acknowledge state can be reached even through a communication step.

A fundamental difference between our communication protocol and those of other PC systems is that there is no *global clock* synchronizing the components of the system. A global clock is an external mechanism that forces all components to execute exactly one computation step together in each unit of time. This is applied to mostly all PC systems. Nevertheless we have decided to avoid using it since it can be seen as *implicit communication* (in contrast to the defined communication steps as *explicit communication*). Look at the following example: in [MMM02] a PCFA system with three components is given that accepts the language $L_{a^nb^nc^n}$ without processing any explicit communication. Certainly, since this language is not even context-free, it cannot be accepted by a single finite automaton and hence not by a system of arbitrary many components without *any* communication - including synchronization. This shows that synchronization alone can increase the computational power of such systems without any explicit communication. Here it is legitimate to ask what amount of explicit communication would really be needed to realize a global clock, and by how much does it slowdown the computation of the system? Another reason for avoiding a global clock is that it contradicts the notion of distributed computing, where no global control mechanism exists. All in all, we will not use any global clock in our definition of PCRA systems such that the components are loosely connected only by explicit communications.

For the reason of abbreviation we define the following notations for a system $\mathcal{M} = (M_1, M_2, \dots, M_n)$ and a component $M_i = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, $1 \leq i \leq n$:

- $\mathsf{REQ}(M_i) = \{\mathsf{req}_d^j \mid \mathsf{req}_d^j \in Q, 1 \leq j \leq n, \text{ and string } d\}$,
- $\mathsf{RES}(M_i) = \{\mathsf{res}_{d,c}^j \mid \mathsf{res}_{d,c}^j \in Q, 1 \leq j \leq n, \text{ and strings } d, c\}$,
- $\mathsf{REC}(M_i) = \{\mathsf{rec}_{d,c}^j \mid \mathsf{rec}_{d,c}^j \in Q, 1 \leq j \leq n, \text{ and strings } d, c\}$,
- $\mathsf{ACK}(M_i) = \{\mathsf{ack}_{d,c}^j \mid \mathsf{ack}_{d,c}^j \in Q, 1 \leq j \leq n, \text{ and strings } d, c\}$,
- $\mathsf{COM}(M_i) = \mathsf{REQ}(M_i) \cup \mathsf{RES}(M_i) \cup \mathsf{REC}(M_i) \cup \mathsf{ACK}(M_i)$,
- $\mathsf{REQ} = \bigcup_{1 \leq i \leq n} \mathsf{REQ}(M_i)$,
- $\mathsf{RES} = \bigcup_{1 \leq i \leq n} \mathsf{RES}(M_i)$,
- $\mathsf{REC} = \bigcup_{1 \leq i \leq n} \mathsf{REC}(M_i)$,
- $\mathsf{ACK} = \bigcup_{1 \leq i \leq n} \mathsf{ACK}(M_i)$, and
- $\mathsf{COM} = \bigcup_{1 \leq i \leq n} \mathsf{COM}(M_i)$.

The type of the components determines the type of the system. If the components are restarting automata of type X for $X \in \mathcal{T}$, then the system is of type PC-X and is called a PC-X-system. With PC-X(n)-system we mean a system of type X and degree n. Moreover, if the maximal window size is bounded by a constant k, we write PC-X(n, k)-system.

For a formalization of the behaviour of a PCRA system we need to define the concepts of configurations and computations. A *configuration* K of a PCRA system of degree n is an n-tuple, which contains a configuration of each component:

$$K = (\kappa_1, \kappa_2, \dots, \kappa_n),$$

where κ_i $(1 \leq i \leq n)$ is either Accept or it is of the form $u_i q_i v_i$, where $q_i \in Q_i$ and either $u_i = ¢\alpha$ and $v_i = \beta\$$ or $u_i = \varepsilon$ and $v_i = ¢\beta\$$ $(\alpha, \beta \in \Gamma_i^*)$. For an input word w and initial states q_i $(1 \leq i \leq n)$, the initial configuration of a PCRA system is:

$$K_0 = (q_1 ¢ w\$, q_2 ¢ w\$, \ldots, q_n ¢ w\$).$$

A *computation step* of a system $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ can be described by the binary relation $\vdash_{\mathcal{M}}$. Let K and K' be two configurations with $K = (\kappa_1, \kappa_2, \ldots, \kappa_n)$ and $K' = (\kappa'_1, \kappa'_2, \ldots, \kappa'_n)$. 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. $\kappa_i \vdash_{M_i} \kappa'_i$ (local computation step),
2. $\exists j \in \{1, 2, \ldots, n\} \setminus \{i\} : \kappa_i = u_i \mathsf{req}^j_{d_i} v_i, \quad \kappa_j = u_j \mathsf{res}^i_{d_j,c} v_j,$
   $\kappa'_i = u_i \mathsf{rec}^j_{d_i,c} v_i, \ \kappa'_j = u_j \mathsf{ack}^i_{d_j,c} v_j$ (communication),
3. $\exists j \in \{1, 2, \ldots, n\} \setminus \{i\} : \kappa_i = u_i \mathsf{res}^j_{d_i,c} v_i, \quad \kappa_j = u_j \mathsf{req}^i_{d_j} v_j,$
   $\kappa'_i = u_i \mathsf{ack}^j_{d_i,c} v_i, \ \kappa'_j = u_j \mathsf{rec}^i_{d_j,c} v_j$ (communication),
4. $\kappa_i = \kappa'_i$, $\kappa_i \neq \mathsf{Accept}$, and no local operation (MVR, MVL, rewrite, restart) or communication of M_i is possible.

Whenever a local computation step of a component is possible, then it is executed immediately and independently of the other components (1). If two components have reached corresponding communication states, then this communication is resolved immediately (2 and 3). If no local transition is possible anymore, then the component is blocked and remains in its current configuration (4). If a component is in a communication state but the communication partner is (still) not in the corresponding communication state, then it remains in its current configuration and waits for the communication answer (4).

At first sight the definition of a computation step may indicate the usage of a global clock that actually should be avoided. In fact, this definition just supports a formal and unique representation of a system's computation[1] but does not influence the synchronization of the components (that is realized only by explicit communication). Particularly the fourth item of the definition contradicts the mechanism of a global clock, since a component can wait arbitrarily long for the execution of a communication. Due to this fact, it is really unimportant for the cooperation of the components within the system how long the local computations between two communication steps are and whether each step of each component is executed in parallel in the same unit of time. Moreover, in contrast to PC systems

[1] At this point it is important to distinguish between the real behaviour of the system and the representation of the behaviour.

with a global clock, it is not guaranteed that two components execute the same number of computation steps between two communications.

The reflexive and transitive closure of the relation $\vdash_{\mathcal{M}}$ is expressed by $\vdash^*_{\mathcal{M}}$ and describes a *computation* of $\mathcal{M}$. Then, the *accepted language* of a PCRA system $\mathcal{M}$ over an input alphabet Σ is

$$L(\mathcal{M}) = \{w \in \Sigma^* \mid (q_1 \text{¢} w \$, q_2 \text{¢} w \$, \ldots, q_n \text{¢} w \$) \vdash^*_{\mathcal{M}} (\kappa_1, \kappa_2, \ldots, \kappa_n), \\ \{\kappa_1, \kappa_2, \ldots, \kappa_n\} \cap \{\mathsf{Accept}\} \neq \emptyset\},$$

where $q_1, q_2, \ldots, q_n$ are the initial states of the components. Moreover, any configuration that includes Accept is called an *accepting configuration.*

A PCRA system is called *nondeterministic* if there is at least one component that is nondeterministic. If all components are deterministic, the system is called *locally deterministic* and gets the prefix det-local. In general, there is no restriction according to the accepting component. That allows the system to accept the input with whatever component reaches the accepting configuration. A more strict definition of determinism in PCRA systems is that the system accepts if and only if the first component accepts. A locally deterministic system that accepts if and only if the first component accepts is called *globally deterministic* and gets the prefix det-global.

In addition, if a system consists only of monotone restarting automata of type $\mathsf{X} \in \mathcal{T}$, then we denote the type of that system by mon-PC-X in the nondeterministic case. For deterministic and monotone components we obtain systems of type det-global-mon-PC-X and det-local-mon-PC-X, respectively.

The class of languages that are accepted by systems of automata of type $\mathsf{X} \in \mathcal{T}$ is denoted by $\mathcal{L}(\mathsf{PC\text{-}X})$ in the nondeterministic case. In the local-deterministic and global-deterministic cases it is denoted by $\mathcal{L}(\mathsf{det\text{-}local\text{-}PC\text{-}X})$ and $\mathcal{L}(\mathsf{det\text{-}global\text{-}PC\text{-}X})$, respectively. Further, the classes of languages that are accepted by systems of monotone restarting automata are denoted by $\mathcal{L}(\mathsf{mon\text{-}PC\text{-}X})$, $\mathcal{L}(\mathsf{det\text{-}local\text{-}mon\text{-}PC\text{-}X})$, and $\mathcal{L}(\mathsf{det\text{-}global\text{-}mon\text{-}PC\text{-}X})$.[2]

Let us consider a first example. Here we use the well-known copy language with a middle marker:

$$L_{w\#w} = \{w\#w \mid w \in \{a,b\}^*\}.$$

Example 9. A PCRA system $\mathcal{M}_{w\#w}$ that accepts the language $L_{w\#w}$ consists of two components M_1 and M_2: $\mathcal{M}_{w\#w} = (M_1, M_2)$. At the beginning of a

[2] Remark: Concerning the notation of the different types and language classes, we differ from those of other PC systems. Instead, we follow the notation that is usually used for CD systems of restarting automata and for individual restarting automata.

computation the input word is placed on the tapes of both components enclosed by the left and right sentinels ¢ and \$. The basic idea is that M_1 moves its window over the first syllable and reads the first symbol of the second syllable, while M_2 reads the first symbol of the first syllable. Then, they compare the symbols using a communication. If both symbols are equal, then they are deleted and a restart operation is applied immediately to both M_1 and M_2. If the symbols are different, then M_1 gets stuck directly after the communication because of a missing applicable transition and the system does not accept the input. If all symbols are processed and both syllables are of the same length, then M_1 reads the \$-symbol and M_2 reads the #-symbol. In this situation M_1 accepts after a communication, and thus the system $\mathcal{M}_{w\#w}$ accepts. Formally, M_1 and M_2 are defined as follows:

$$\begin{aligned}
M_1 &= (\{q_0, \mathsf{req}^2, \mathsf{rec}^2_a, \mathsf{rec}^2_b, \mathsf{rec}^2_\#, q_r\}, \{a, b, \#\}, \{a, b, \#\}, ¢, \$, q_0, 2, \delta_1),\\
M_2 &= (\{q_0, \mathsf{res}^1_a, \mathsf{res}^1_b, \mathsf{res}^1_\#, \mathsf{ack}^1_a, \mathsf{ack}^1_b, \mathsf{ack}^1_\#, q_r\}, \{a, b, \#\}, \{a, b, \#\}, ¢, \$, q_0, 2, \delta_2),
\end{aligned}$$

where, for all $\alpha \in \{¢, a, b\} \cdot \{a, b, \#\}$, $\beta \in \{a, b, \$\}$, and $\gamma \in \mathcal{PC}^{(2)}$,

$$\begin{array}{llll}
\delta_1(q_0, \alpha) & = (q_0, \mathsf{MVR}), & \delta_2(q_0, ¢a) & = \mathsf{res}^1_a,\\
\delta_1(q_0, \#\beta) & = \mathsf{req}^2, & \delta_2(q_0, ¢b) & = \mathsf{res}^1_b,\\
\delta_1(\mathsf{rec}^2_a, \#a) & = (q_r, \#), & \delta_2(q_0, ¢\#) & = \mathsf{res}^1_\#,\\
\delta_1(\mathsf{rec}^2_b, \#b) & = (q_r, \#), & \delta_2(\mathsf{ack}^1_a, ¢a) & = (q_r, ¢),\\
\delta_1(\mathsf{rec}^2_\#, \#\$) & = \mathsf{Accept}, & \delta_2(\mathsf{ack}^1_b, ¢b) & = (q_r, ¢),\\
\delta_1(q_r, \gamma) & = \mathsf{Restart}, & \delta_2(q_r, \gamma) & = \mathsf{Restart}.
\end{array}$$

The superscripts of the communication states of M_1 are always 2 because M_2 is the only component M_1 can communicate with. Particularly it makes no sense for a component to communicate with itself. Accordingly, the superscripts of the communication states of M_2 are always 1. Whenever the communication partner is unique as in this case, then we will omit the superscript of the communication states in further considerations. Moreover, the response, receive, and acknowledge states in our example have only one entry within the subscripts, and the request state of M_1 has not any subscript. The reason for this is that no local information about the previous computation steps of the current cycle needs to be stored. In addition, if also the message is unimportant, that is, if only the fact is interesting whether a communication takes place or not, then we even omit the message in the subscript. In the extreme we could have communication states like req, res, rec, and ack without any annotation.

Let us now consider how the system behaves for a particular input. A computation for $\mathcal{M}_{w\#w}$ and the input word $ab\#ab$ is written in the following way, where the computation begins in the first column, proceeds downwards, and is continued in the second column:

$$\begin{array}{lll}
 & (q_0\text{¢}ab\#ab\$, & q_0\text{¢}ab\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}q_0ab\#ab\$, & \mathsf{res}^1_a\text{¢}ab\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}aq_0b\#ab\$, & \mathsf{res}^1_a\text{¢}ab\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}abq_0\#ab\$, & \mathsf{res}^1_a\text{¢}ab\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}ab\mathsf{req}^2\#ab\$, & \mathsf{res}^1_a\text{¢}ab\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}ab\mathsf{rec}^2_a\#ab\$, & \mathsf{ack}^1_a\text{¢}ab\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}ab\#q_rb\$, & \text{¢}q_rb\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (q_0\text{¢}ab\#b\$, & q_0\text{¢}b\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}q_0ab\#b\$, & \mathsf{res}^1_b\text{¢}b\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}aq_0b\#b\$, & \mathsf{res}^1_b\text{¢}b\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}abq_0\#b\$, & \mathsf{res}^1_b\text{¢}b\#ab\$)
\end{array}$$

$$\begin{array}{lll}
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}ab\mathsf{req}^2\#b\$, & \mathsf{res}^1_b\text{¢}b\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}ab\mathsf{rec}^2_b\#b\$, & \mathsf{ack}^1_b\text{¢}b\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}ab\#q_r\$, & \text{¢}q_r\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (q_0\text{¢}ab\#\$, & q_0\text{¢}\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}q_0ab\#\$, & \mathsf{res}^1_\#\text{¢}\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}aq_0b\#\$, & \mathsf{res}^1_\#\text{¢}\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}abq_0\#\$, & \mathsf{res}^1_\#\text{¢}\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}ab\mathsf{req}^2\#\$, & \mathsf{res}^1_\#\text{¢}\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\text{¢}ab\mathsf{rec}^2_\#\#\$, & \mathsf{ack}^1_\#\text{¢}\#ab\$) \\
\vdash_{\mathcal{M}_{w\#w}} & (\mathsf{Accept}, & \mathsf{ack}^1_\#\text{¢}\#ab\$)
\end{array}$$

The components of $\mathcal{M}$ are deterministic monotone R-automata and, moreover, only M_1 is able to accept the input. Thus, the system $\mathcal{M}$ is of type det-global-mon-PC-R(2). □

The copy language in Example 9 is not even growing context sensitive and thus cannot be accepted by any single deterministic R-automaton, so this example gives a first impression of how communication increases the computational power of restarting automata. Moreover, we will see later in this thesis that even the copy language without a middle marker can be accepted by a globally deterministic system of three components with a window size of one.

In some situations it will be useful to describe the progress of the configurations of a particular component M in a system $\mathcal{M}$. Therefore, the binary relation $\vdash_{M,\mathcal{M}}$ is introduced. It is somehow an extension of $\vdash_M$ that includes the communication within the system. If $\mathcal{M}$ is of degree n, κ and κ' are two configurations, and M is the i-th component of $\mathcal{M}$, then $\kappa \vdash_{M,\mathcal{M}} \kappa'$ holds if and only if there exist configurations κ_j, κ'_j for all $j \in \{1,\ldots,n\} \setminus \{i\}$ with

$$(\kappa_1,\ldots,\kappa_{i-1},\kappa,\kappa_{i+1},\ldots,\kappa_n) \vdash_{\mathcal{M}} (\kappa'_1,\ldots,\kappa'_{i-1},\kappa',\kappa'_{i+1},\ldots,\kappa'_n).$$

Moreover, with $\kappa \vdash^*_{M,\mathcal{M}} \kappa'$ we denote a computation of M within $\mathcal{M}$, i.e. $\kappa \vdash^*_{M,\mathcal{M}} \kappa'$ if and only if there exist configurations κ_j, κ'_j for all $j \in \{1,\ldots,n\} \setminus \{i\}$ such that

$$(\kappa_1,\ldots,\kappa_{i-1},\kappa,\kappa_{i+1},\ldots,\kappa_n) \vdash^*_{\mathcal{M}} (\kappa'_1,\ldots,\kappa'_{i-1},\kappa',\kappa'_{i+1},\ldots,\kappa'_n).$$

Observe that in contrast to the relation $\vdash_M$, the relation $\vdash^*_{M,\mathcal{M}}$ is not the reflexive and transitive closure of $\vdash_{M,\mathcal{M}}$.

In the remaining subsections some more technical machinery is developed and first results are derived.

5.1.2 Communicational equivalence between components of a system

We start this topic with the following observation: Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PCRA system. Replacing a component M_i $(1 \leq i \leq n)$ by an equivalent automaton M_i' leads to the system $\mathcal{M}' = (M_1, \ldots, M_{i-1}, M_i', M_{i+1}, \ldots, M_n)$. Usually $\mathcal{M}$ and $\mathcal{M}'$ are not equivalent anymore as can be easily seen from the next example:

Example 10. $\mathcal{M} = (M_1, M_2)$ with $M_1 = (\{q_1, \mathsf{res}^2, \mathsf{ack}^2\}, \{a\}, \{a\}, ¢, \$, q_1, 1, \delta_1)$ and $M_2 = (\{q_2, \mathsf{req}^1, \mathsf{rec}^1\}, \{a\}, \{a\}, ¢, \$, q_2, 1, \delta_2)$, where

$$\begin{aligned} \delta_1(q_1, ¢) &= \{\mathsf{res}^2\}, \\ \delta_2(q_2, ¢) &= \{\mathsf{req}^1\}, \\ \delta_2(\mathsf{rec}^1, ¢) &= \{\mathsf{Accept}\}. \end{aligned}$$

Then $L(M_1) = L(M_2) = \emptyset$ and $L(\mathcal{M}) = \{a\}^*$. Let $M_1' = (\{q_1'\}, \{a\}, \{a\}, ¢, \$, q_1', 1, \delta_1')$ with $\delta_1'(q_1', ¢) = \emptyset$. Then it follows that $L(M_1') = \emptyset = L(M_1)$. But for $\mathcal{M}' = (M_1', M_2)$ it holds that $L(\mathcal{M}') = \emptyset \neq L(\mathcal{M})$. □

Though the component M_1 in the system $\mathcal{M}$ was replaced by an equivalent automaton M_1', the resulting system $\mathcal{M}'$ behaves quite differently from the original system. The reason for this is the fact that not only the behaviour of the local computations of the various components, but also the behaviour with respect to communication influences the work of the whole system. The next definition deals with this situation and gives a stronger interpretation of equivalence.

Definition 2 (Communicational equivalence). Let $M_1 = (Q_1, \Sigma, \Gamma_1, ¢, \$, q_0^{(1)}, k, \delta_1)$ and $M_2 = (Q_2, \Sigma, \Gamma_2, ¢, \$, q_0^{(2)}, k, \delta_2)$ be two components such that $\mathsf{COM}(M_1) = \mathsf{COM}(M_2)$. Further, let

$$\begin{aligned} C_1 &= \mathsf{REQ}(M_1) \cup \mathsf{RES}(M_1) = \mathsf{REQ}(M_2) \cup \mathsf{RES}(M_2) \text{ and} \\ C_2 &= \mathsf{REC}(M_1) \cup \mathsf{ACK}(M_1) = \mathsf{REC}(M_2) \cup \mathsf{ACK}(M_2). \end{aligned}$$

Then M_1 and M_2 are called *communicational equivalent* (denoted by $M_1 \equiv_c M_2$) if they are equivalent $(L(M_1) = L(M_2))$ and the following three conditions hold:

1. for all $w \in \Sigma^*$ and $p \in C_1$,

$$q_0^{(1)}¢w\$ \vdash_{M_1}^* upv \quad \Leftrightarrow \quad q_0^{(2)}¢w\$ \vdash_{M_2}^* upv,$$

2. for all $p \in C_2$ and $q \in C_1$,

$$upv \vdash_{M_1}^{*} u'qv' \quad \Leftrightarrow \quad upv \vdash_{M_2}^{*} u'qv',$$

3. for all $p \in C_2$,

$$upv \vdash_{M_1}^{*} \mathsf{Accept} \quad \Leftrightarrow \quad upv \vdash_{M_2}^{*} \mathsf{Accept}.$$

□

Observe that the assumption $\mathsf{COM}(M_1) = \mathsf{COM}(M_2)$ implies that $\mathsf{REQ}(M_1) = \mathsf{REQ}(M_2)$, $\mathsf{RES}(M_1) = \mathsf{RES}(M_2)$, $\mathsf{REC}(M_1) = \mathsf{REC}(M_2)$, and $\mathsf{ACK}(M_1) = \mathsf{ACK}(M_2)$. Moreover, this is no real restriction, because whenever a communication state $q \in \mathsf{COM}(M_1) \setminus \mathsf{COM}(M_2)$ or $q \in \mathsf{COM}(M_2) \setminus \mathsf{COM}(M_1)$ is reached by a component, then one of the three conditions does not hold anyway. The communicational equivalence ensures that two automata have the same 'global behaviour' within a system. It means that both components react on communications in the same manner. Now we prove that two systems with communicational equivalent components are indeed equivalent.

Lemma 1. *Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PCRA system, $M_j = (Q_j, \Sigma, \Gamma_j, ¢, \$, q_0^{(j)}, k, \delta_j)$ for all $j \in \{1, \ldots, n\}$, $M_i' = (Q_i', \Sigma, \Gamma_i', ¢, \$, q_0^{(i)}, k, \delta_i')$ an automaton with $M_i' \equiv_c M_i$ for an arbitrary $i \in \{1, \ldots, n\}$, and $\mathcal{M}' = (M_1, \ldots, M_{i-1}, M_i', M_{i+1}, \ldots, M_n)$. Then for all $w \in \Sigma^*$, $uv \in \{¢\} \cdot (\Gamma_i \cup \Gamma_i')^* \cdot \{\$\}$, and $q \in \mathsf{REQ}(M_i) \cup \mathsf{RES}(M_i)$,*

$$q_0^{(i)} ¢ w \$ \vdash_{M_i, \mathcal{M}}^{*} uqv \quad \Leftrightarrow \quad q_0^{(i)} ¢ w \$ \vdash_{M_i', \mathcal{M}'}^{*} uqv.$$

Proof. Let γ_1 be a computation of M_i in $\mathcal{M}$ such that $\gamma_1 = q_0^{(i)} ¢ w \$ \vdash_{M_i, \mathcal{M}}^{*} uqv$ for an arbitrary input $w \in \Sigma^*$, $uv \in \{¢\} \cdot (\Gamma_i \cup \Gamma_i')^* \cdot \{\$\}$, and $q \in \mathsf{REQ}(M_i) \cup \mathsf{RES}(M_i)$. Further, let $S(\gamma_1) = (p_1, p_2, \ldots, p_t, q)$ be the sequence of request and response states that are reached by M_i in the computation γ_1 in chronological order.

By induction on the length of $S(\gamma_1)$, i.e. the number of reached communication states, we show that there exists a computation γ_2 of M_i' in $\mathcal{M}'$ such that $\gamma_2 = q_0^{(i)} ¢ w \$ \vdash_{M_i', \mathcal{M}'}^{*} uqv$ with $S(\gamma_2) = S(\gamma_1)$. In particular, all communications of M_i' in γ_2 can be resolved, since the same communication states are reached as in the computation γ_1 of M_i and all the other components behave in $\mathcal{M}'$ in the same way as in $\mathcal{M}$.

Basis: For $S(\gamma_1) = (q)$ the assumption follows directly from the first item of the Definition 2.

Inductive step: For $S(\gamma_1) = (p_1, \ldots, p_t, q)$ with $t > 0$ it holds that

$$\begin{array}{lll l} q_0^{(i)}¢w\$ & \vdash^*_{M_i,\mathcal{M}} & u'p_tv'(=\gamma_3) & (1) \\ u'p_tv' & \vdash_{M_i,\mathcal{M}} & u'p'_tv' & (2) \\ u'p'_tv' & \vdash^*_{M_i,\mathcal{M}} & uqv(=\gamma_4) & (3) \end{array}$$

where $p_t = \mathsf{req}^j_d$ and $p'_t = \mathsf{rec}^j_{d,c}$ or $p_t = \mathsf{res}^j_{d,c}$ and $p'_t = \mathsf{ack}^j_{d,c}$, $S(\gamma_3) = (p_1, \ldots, p_t)$, and $S(\gamma_4) = (q)$. From line (1) and the induction hypothesis it follows that

$$q_0^{(i)}¢w\$ \vdash^*_{M'_i,\mathcal{M}'} u'p_tv'(=\gamma_5)$$

with $S(\gamma_5) = S(\gamma_3) = (p_1, \ldots, p_t)$. The communication step in line (2) can be executed in $\mathcal{M}'$ as well, since all previous communications are resolved (induction hypothesis) and the other components behave in the same way as in $\mathcal{M}$. From line (3) and the second item of Definition 2 it follows that

$$u'p'_tv' \vdash^*_{M'_i,\mathcal{M}'} uqv(=\gamma_6)$$

with $S(\gamma_6) = S(\gamma_4) = (q)$, i.e. γ_6 is a local computation of M'_i. Altogether we have

$$q_0^{(i)}¢w\$ \vdash^*_{M'_i,\mathcal{M}'} uqv(=\gamma_2)$$

with $S(\gamma_2) = S(\gamma_1) = (p_1, \ldots, p_t, q)$. The opposite direction can be shown in the same way because of the equivalences of Definition 2. □

Corollary 1. *Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ and $\mathcal{M}' = (M'_1, M'_2, \ldots, M'_n)$ be two PCRA systems with $M_i \equiv_c M'_i$ for all $i \in \{1, \ldots, n\}$. Then $L(\mathcal{M}) = L(\mathcal{M}')$ holds.*

Proof. This results immediately from Lemma 1 and the third item of Definition 2. □

The next corollary is sometimes useful for technical reasons.

Corollary 2. *For every restarting automaton with a communication state as the initial state, there exists a communicational equivalent restarting automaton with an initial state that is not a communication state.*

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q, k, \delta)$ be a restarting automaton with $q \in \mathsf{REQ}(M) \cup \mathsf{RES}(M)$. Construct a restarting automaton $M' = (Q \cup \{p\}, \Sigma, \Gamma, ¢, \$, p, k, \delta')$, where $p \notin Q$ and $\delta' = \delta \cup \{(p, \alpha, q) \mid \alpha \in \mathcal{PC}^{(k)}\}$. Thus, every computation of M' starts with $p¢w\$ \vdash_{M'} q¢w\$$ for any input word $w \in \Sigma^*$. Hence, $M \equiv_c M'$. □

5.1.3 Restarting automata with state-change-only transitions

Usually in a restarting automaton a change into a particular state is connected with a move of the window on the tape (MVL, MVR) or a replace operation. In some proofs it is useful to define restarting automata in such a way that the state can be changed without performing any operation on the tape. Here we call transitions of the form $q \in \delta(p, u)$ *state-change-only transitions* (SCO transitions for short), where p and q are states and u is a part of the tape content that could possibly be read by the window. This is quite similar to spontaneous transitions for nondeterministic finite automata or pushdown automata.

Definition 3 (SCO transition). Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an arbitrary restarting automaton of any type. Then an SCO transition is of the form $q \in \delta(p, u)$, where $p, q \in Q$, and $u \in \mathcal{PC}^{(k)}$. This means that M, being in state p and reading the word u on the tape, may change into the state q, without changing the tape content, keeping the window at the current position. □

If M is a component within a PC system of restarting automata, transitions of the form $q \in \delta(p, u)$ are not called SCO transitions if q is a communication state. When M changes into a communication state, this is not just a change of the current state, but it is also connected with a communication operation. Now we show that restarting automata with SCO transitions are equivalent to restarting automata without SCO transitions with respect to their computational power. Since it is clear that restarting automata without SCO transitions cannot be more powerful than those with SCO transitions, it remains to show the opposite direction. The first proof deals only with restarting automata not contained within a PC system. Thereafter, it is shown that SCO transitions do not result in an increase of the computational power even in PC systems.

Theorem 1. *Let M be an arbitrary restarting automaton of any type with SCO transitions. Then there exists an automaton M' of the same type without SCO transitions such that $L(M) = L(M')$.*

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be a restarting automaton of any type with SCO transitions. The following algorithm constructs an automaton $M' = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta')$ of the same type without using SCO transitions such that $L(M) = L(M')$.

(1) Let P_1 be the set of all SCO transitions of M:

$$P_1 := \{(p, \alpha, q) \mid q \in \delta(p, \alpha)\}.$$

(2) Remove every (p, α, q) from P_1 for which $\delta(q, \alpha) = \emptyset$ holds:

$$P_2 := P_1 \setminus \{(p, \alpha, q) \mid \delta(q, \alpha) = \emptyset\}.$$

(3) Determine the transitive closure of P_2 which is the set of computation paths only containing SCO transitions:

$$\begin{aligned} P_3 := \{(p, \alpha, q) \mid (p, \alpha, q) \in P_2 \text{ or } \exists n \geq 0 : \exists q_1, q_2, \ldots, q_n \in Q :\\ (p, \alpha, q_1), (q_1, \alpha, q_2), \ldots, (q_n, \alpha, q) \in P_2\}. \end{aligned}$$

(4) Remove every (p, α, p) from P_3:

$$P_4 := P_3 \setminus \{(p, \alpha, p) \mid (p, \alpha, p) \in P_3\}.$$

(5) For each $A \in \delta(p, \alpha)$ with $A \neq q$ for any $q \in Q$, take $A \in \delta'(p, \alpha)$.

(6) While $P_4 \neq \emptyset$ do:

(6.1) Choose a $(p, \alpha, q) \in P_4$.

(6.2) For each $A \in \delta(q, \alpha)$, define $A \in \delta'(p, \alpha)$.

(6.3) Remove (p, α, q) from P_4.

The basic idea of the algorithm is to shorten the computations of M by eliminating the SCO steps successively. Now, it remains to show that for any input word $w \in \Sigma^*$ holds: $q_0¢w\$ \vdash_M^* \mathsf{Accept}$ if and only if $q_0¢w\$ \vdash_{M'}^* \mathsf{Accept}$.

„$\Rightarrow$ “: An SCO sequence of length r is a computation consisting of r SCO transitions. Formally this can be written as $uq_1v \vdash uq_2v \vdash \cdots \vdash uq_rv \vdash uq_{r+1}v$. By induction on the number of SCO sequences it can be shown that $\kappa \vdash_M^* \mathsf{Accept} \Rightarrow \kappa \vdash_{M'}^* \mathsf{Accept}$ holds for any configuration κ that is not reached directly from an SCO transition. If the computation $\kappa \vdash_M^* \mathsf{Accept}$ does not contain an SCO sequence, then $\kappa \vdash_{M'}^* \mathsf{Accept}$ follows directly from line (5) of the algorithm. Let $\kappa \vdash_M^* \mathsf{Accept}$ contain $s + 1$ SCO sequences ($s \geq 0$). Then there exist states $q_1, q_2, \ldots, q_r, q$ and tape contents u, v, u', v' with

$$\kappa \vdash_M^* uq_1v \vdash_M uq_2v \vdash_M \ \ldots \vdash_M uq_rv \vdash_M u'qv' \vdash_M^* \mathsf{Accept}.$$

W.l.o.g. $\kappa \vdash_M^* uq_1v$ contains no SCO sequence and $uq_1v \vdash_M uq_2v \vdash_M \ \ldots \vdash_M uq_rv$ is the first SCO sequence (of length $r-1$) within the computation $\kappa \vdash_M^* \mathsf{Accept}$. Moreover, $uq_rv \vdash_M u'qv'$ is not an SCO transition (otherwise $u'qv'$ would belong to the SCO sequence), and $q_i \neq q_{i+1}$ for all $1 \leq i < r$. The last part of the computation

$(u'qv' \vdash_M^* \mathsf{Accept})$ contains the remaining s SCO sequences. Now it follows that $q_{i+1} \in \delta(q_i, \pi_k(v))$ for all $1 \leq i < r$ and $(q, \mathsf{MVR}) \in \delta(q_r, \pi_k(v))$ (if $uq_rv \vdash_M^{MVR} u'qv'$; MVL, rewrite, restart, and Accept similar). Hence $(q_i, \pi_k(v), q_{i+1}) \in P_1$ and $(q_i, \pi_k(v), q_{i+1}) \in P_2$ for all $1 \leq i < r$ because of line (1) and (2) of the algorithm. Due to line (3) $(q_1, \pi_k(v), q_r) \in P_3$. If $q_1 \neq q_r$, then $(q_1, \pi_k(v), q_r) \in P_4$. (Otherwise there exists a computation $\kappa \vdash_M^* uq_1v \vdash_M u'qv' \vdash_M^* \mathsf{Accept}$ with s SCO sequences and the induction hypothesis can be applied.). Furthermore, $(q, \mathsf{MVR}) \in \delta'(q_1, \pi_k(v))$ because of line (6.2). Subsequently it follows that

$$\kappa \vdash_{M'}^* uq_1v \vdash_{M'} u'qv' \vdash_{M'}^* \mathsf{Accept},$$

where $\kappa \vdash_{M'}^* uq_1v$ follows from line (5) of the algorithm, and $u'qv' \vdash_{M'}^* \mathsf{Accept}$ results from the induction hypothesis. In particular, if $\kappa = q_0 ¢ w \$$ for some input word $w \in \Sigma^*$, then $q_0 ¢ w \$ \vdash_M^* \mathsf{Accept}$ implies $q_0 ¢ w \$ \vdash_{M'}^* \mathsf{Accept}$.

„$\Leftarrow$ “: By induction on the length of the computations of M' it can be shown that $\kappa \vdash_{M'}^* \mathsf{Accept} \Rightarrow \kappa \vdash_M^* \mathsf{Accept}$ holds for any configuration κ. For zero computation steps $\kappa = \mathsf{Accept}$ and the assumption holds. Let $\kappa \vdash_{M'}^{n+1} \mathsf{Accept}$ for a configuration $\kappa = uqv = uqv_1 \ldots v_t$. Then there exists a state q' such that

$$uqv_1 \ldots v_t \vdash_{M'}^{MVR} uv_1q'v_2 \ldots v_t \vdash_{M'}^n \mathsf{Accept}$$

(MVL, rewrite, and restart similar), and it follows that

$$\begin{array}{rll} uqv_1 \ldots v_t & \vdash_M^* uv_1q'v_2 \ldots v_t & (1) \\ & \vdash_M^* \mathsf{Accept} & (2) \end{array}$$

Line (1) holds because:

$$\begin{aligned}
& uqv_1 \dots v_t \vdash_{M'}^{MVR} uv_1q'v_2 \dots v_t \\
\Rightarrow\ & (q', \mathsf{MVR}) \in \delta'(q, \pi_k(v)) \\
\Rightarrow\ & (q', \mathsf{MVR}) \in \delta(q, \pi_k(v)) \text{ or } \exists q'' \in Q : \\
& \quad (q, \pi_k(v), q'') \in P_4 \text{ and } (q', \mathsf{MVR}) \in \delta(q'', \pi_k(v)) \\
\Rightarrow\ & (q', \mathsf{MVR}) \in \delta(q, \pi_k(v)) \text{ or } \exists q'' \in Q : \\
& \quad (q, \pi_k(v), q'') \in P_3 \text{ and } (q', \mathsf{MVR}) \in \delta(q'', \pi_k(v)) \text{ and } q'' \neq q \\
\Rightarrow\ & (q', \mathsf{MVR}) \in \delta(q, \pi_k(v)) \text{ or } \exists q'' \in Q : \exists r \geq 1 : \exists q_1, q_2, \dots, q_r \in Q : \\
& \quad (q, \pi_k(v), q_1) \in P_2, (q_i, \pi_k(v), q_{i+1}) \in P_2 \text{ for all } 0 < i < r, q_r = q'', \\
& \quad \text{and } (q', \mathsf{MVR}) \in \delta(q'', \pi_k(v)) \\
\Rightarrow\ & (q', \mathsf{MVR}) \in \delta(q, \pi_k(v)) \text{ or } \exists q'' \in Q : \exists r \geq 1 : \exists q_1, q_2, \dots, q_r \in Q : \\
& \quad (q, \pi_k(v), q_1) \in P_1, (q_i, \pi_k(v), q_{i+1}) \in P_1 \text{ for all } 0 < i < r, q_r = q'', \\
& \quad \text{and } (q', \mathsf{MVR}) \in \delta(q'', \pi_k(v)) \\
\Rightarrow\ & (q', \mathsf{MVR}) \in \delta(q, \pi_k(v)) \text{ or } \exists q'' \in Q : \exists r \geq 1 : \exists q_1, q_2, \dots, q_r \in Q : \\
& \quad q_1 \in \delta(q, \pi_k(v)), q_{i+1} \in \delta(q_i, \pi_k(v)) \text{ for all } 0 < i < r, q_r = q'', \\
& \quad \text{and } (q', \mathsf{MVR}) \in \delta(q'', \pi_k(v)) \\
\Rightarrow\ & uqv_1 \dots v_t \vdash_M uv_1q'v_2 \dots v_t \text{ or } \exists q'' \in Q : \exists r \geq 1 : \exists q_1, q_2, \dots, q_r \in Q : \\
& \quad uqv_1 \dots v_t \vdash_M uq_1v_1 \dots v_t \vdash_M \dots \vdash_M uq_rv_1 \dots v_t = uq''v_1 \dots v_t \\
& \qquad \vdash_M uv_1q'v_2 \dots v_t
\end{aligned}$$

Line (2) holds because of the induction hypothesis. In particular, if $q_0¢w\$ \vdash_{M'}^* \mathsf{Accept}$, then $q_0¢w\$ \vdash_M^* \mathsf{Accept}$. Hence, it follows that $L(M) = L(M')$. □

The algorithm above terminates for every input, and it needs time $O(n^4)$ and space $\mathcal{O}(n^2)$, where n is the number of transitions of M. Lines (1), (2), and (5) need linear time and space. Lines (3) and (4) need time $\mathcal{O}(n^3)$ and space $\mathcal{O}(n^2)$. For this, we can use the Floyd-Warshall algorithm [Flo62, War62]. Finally, line (6) needs time $\mathcal{O}(n^4)$ and space $\mathcal{O}(n^2)$.

The next theorem indicates that the use of SCO transitions has no effect on the computational power even in PC systems of restarting automata. To show this the concept of communicational equivalence is used.

Theorem 2. *Let M be a component within a PC system of restarting automata of an arbitrary type that includes* SCO *transitions. Then there exists a communicational equivalent component M' of the same type without* SCO *transitions ($M \equiv_c M'$).*

Proof. The proof is nearly the same as the one of Theorem 1. By replacing κ with the configurations $q_0¢w\$$ or xqy for an input word w and a communication state $q \in \mathsf{REC}(M) \cup \mathsf{ACK}(M)$ and replacing Accept with xpy for a communication state $p \in \mathsf{REQ}(M) \cup \mathsf{RES}(M)$, it can be seen that the three conditions (beside the condition $L(M) = L(M'))$ of Definition 2 hold. □

One problem dealing with SCO steps are loops in the computation that cannot occur in computations of restarting automata that use only MVR steps. Such loops can result in infinitely long computations for a given input word. But this is no real disadvantage, since the algorithm above can be used to decide whether an automaton can reach a loop (the automaton can reach an SCO loop if and only if P_3 contains at least one triple of the type (p, α, p)) and moreover (due to Theorems 1 and 2) a (communicational) equivalent automaton without SCO transitions can be constructed effectively.

Furthermore, the construction within the proof of Theorem 2 has only an effect on the currently considered component, but not on the other components of the system. Thus, by replacing the components with SCO transitions by components without SCO transitions successivley, we obtain the following corollary.

Corollary 3. *Every system that includes components with* SCO *transitions can be effectively transformed into an equivalent system without* SCO *transitions.*

5.1.4 Multicast and broadcast

Consider the language

$$L_{c\text{-}copy} = \{w(\#w)^c \mid w \in \{a,b\}^+\},$$

where c is a constant positive integer. An intuitive way to construct a system for this language would be to have $c+1$ components each working on one of the $c+1$ syllables quite similar to the copy language of Example 9 (although $L_{c\text{-}copy}$ can be accepted by a system with only two components as we will see within the section „Further examples“). Comparing the first symbols of all syllables by one-to-one communications can be very expensive. Rather a one-to-many communication would be useful, so that one particular component sends its first symbol to all other components that now can compare the information received with their own currently read symbol.

Now we introduce an operation that allows a component to send a message not only to one but to many components within the system. Therefore, the present communication concept is extended by multicast and broadcast communication operations. While in a broadcast a certain information is sent to all other components, in the multicast operation the message is sent to a determined subset of all components (except the sending component itself). In this sense the broadcast is a special case of the multicast. Usually, if a member of a (communication) network sends a broadcast message, it does not necessarily require an acknowledgement from the addressees. Here we limit ourself to broadcasts and multicasts with ‘echo’, where each addressee sends an acknowledgement to the sender. Thus, the sender knows which members have received the original message and can send the message again if there are some members that did not obtain the message. In the context of the PCRA systems this means that the component sending the multicast waits until every addressee reaches a corresponding communication state. The next definitions give a formal description of broad- and multicasts within PCRA systems.

Definition 4 (Multicast communication step). Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PCRA system of an arbitrary type. Then, a multicast operation is a transition of the form

$$\mathsf{res}_{d,c}^{\{i_1,i_2,\ldots,i_r\}} \in \delta(q, \alpha),$$

where d is a local information, c is a message, and $\{i_1, i_2, \ldots, i_r\} \subset \{1, 2, \ldots, n\}$ are the indices of the addressees not containing the index of the sender. □

Using broadcast communication we notate this also by

$$\mathsf{res}_{d,c}^{*} \in \delta(q, \alpha),$$

since the message is sent to *all* other components. Now, we extend the definition of a computation step of a system.

Definition 5. Let K and K' be two configurations with $K = (\kappa_1, \kappa_2, \ldots, \kappa_n)$ and $K' = (\kappa'_1, \kappa'_2, \ldots, \kappa'_n)$. Then $K \vdash_{\mathcal{M}} K'$ iff for all $i \in \{1, 2, \ldots, n\}$ one of the following conditions holds:

1. $\kappa_i \vdash_{M_i} \kappa'_i$ (local computation step),

2. $\exists j \in \{1, 2, \ldots, n\} \setminus \{i\} : \kappa_i = u_i \mathsf{req}_{d_i}^{j} v_i, \quad \kappa_j = u_j \mathsf{res}_{d_j,c}^{i} v_j,$
$\kappa'_i = u_i \mathsf{rec}_{d_i,c}^{j} v_i, \ \kappa'_j = u_j \mathsf{ack}_{d_j,c}^{i} v_j$ (communication),

3. $\exists j \in \{1, 2, \ldots, n\} \setminus \{i\} : \kappa_i = u_i \mathsf{res}_{d_i,c}^{j} v_i, \quad \kappa_j = u_j \mathsf{req}_{d_j}^{i} v_j,$
$\kappa'_i = u_i \mathsf{ack}_{d_i,c}^{j} v_i, \ \kappa'_j = u_j \mathsf{rec}_{d_j,c}^{i} v_j$ (communication),

4. $\exists j_1, \ldots, j_r \in \{1, 2, \ldots, n\} \setminus \{i\}$:
$\kappa_i = u_i \mathsf{res}_{d_i,c}^{\{j_1,\ldots,j_r\}} v_i, \ \forall j \in \{j_1, \ldots, j_r\} : \kappa_j = u_j \mathsf{req}_{d_j}^{i} v_j,$
$\kappa'_i = u_i \mathsf{ack}_{d_i,c}^{\{j_1,\ldots,j_r\}} v_i, \forall j \in \{j_1, \ldots, j_r\} : \kappa'_j = u_j \mathsf{rec}_{d_j,c}^{i} v_j$ (multicast),

5. $\kappa_i = \kappa'_i$, $\kappa_i \neq \mathsf{Accept}$, and no local operation (MVR, MVL, $\mathsf{Restart}$, replacement) or communication (including multicasts) of M_i is possible. □

Although the defined multicast operation can be a useful simplification for the construction of a system, it does not increase the computational power, as the next theorem shows.

Theorem 3. *For every PCRA $\mathcal{M}$ of an arbitrary type that is allowed to use multicast operations, there exists a PCRA system $\mathcal{M}'$ of the same type that does not use any multicast operations and accepts exactly the same language as $\mathcal{M}$.*

Proof. Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PCRA system of an arbitrary type that is allowed to use multicast operations. We construct a system $\mathcal{M}' = (M_1', M_2', \ldots, M_n')$ of the same type with $L(\mathcal{M}) = L(\mathcal{M}')$. Therefore every multicast operation that appears in any component of $\mathcal{M}$ is eliminated. Let

$$\mathsf{res}_{d,c}^{\{j_1,\ldots,j_r\}} \in \delta_i(q, \alpha) \text{ and } A \in \delta(\mathsf{ack}_{d,c}^{\{j_1,\ldots,j_r\}}, \alpha)$$

be an arbitrary transition of a multicast operation from the component M_i. These are replaced with the following communication transitions in M_i':

$$\begin{aligned}
\mathsf{res}_{d,c}^{j_1} &\in \delta_i'(q, \alpha),\\
\mathsf{res}_{d,c}^{j_2} &\in \delta_i'(\mathsf{ack}_{d,c}^{j_1}, \alpha),\\
\mathsf{res}_{d,c}^{j_3} &\in \delta_i'(\mathsf{ack}_{d,c}^{j_2}, \alpha),\\
&\vdots\\
\mathsf{res}_{d,c}^{j_r} &\in \delta_i'(\mathsf{ack}_{d,c}^{j_{r-1}}, \alpha),\\
A &\in \delta_i'(\mathsf{ack}_{d,c}^{j_r}, \alpha).
\end{aligned}$$

If any of the states $\mathsf{res}_{d,c}^{j_1}$, $\mathsf{ack}_{d,c}^{j_1}$, $\mathsf{res}_{d,c}^{j_2}$, $\mathsf{ack}_{d,c}^{j_2}$, $\ldots$, $\mathsf{res}_{d,c}^{j_r}$, $\mathsf{ack}_{d,c}^{j_r}$ already appears in the set of states of M_i, then the local information d has to be chosen in a way that makes the states unique. If there are several multicast operations defined for the same state q and tape content α, then the new transitions are defined for each multicast operation. If the initial state of a component M_i is of the form $\mathsf{res}_{d,c}^{\{j_1,\ldots,j_r\}}$, then the initial state of M_i' is $\mathsf{res}_{d,c}^{j_1}$ and the new transitions above are included. If the initial state of M_i is not a multicast communication state, then the initial state in M_i' is the same as in M_i.

To show that $\mathcal{M}$ and $\mathcal{M}'$ accept the same language, we consider the following multicast step within a computation of $\mathcal{M}$, where the order of the components M_i, M_{j_1}, M_{j_2}, $\ldots$, M_{j_r} is irrelevant:

$$\begin{array}{rlllll}
 & (\ldots, & \kappa_i, & \kappa_{j_1}, & \ldots, & \kappa_{j_r}, \quad \ldots) \\
= & (\ldots, & u_i q_i v_i, & u_{j_1} q_{j_1} v_{j_1}, & \ldots, & u_{j_r} q_{j_r} v_{j_r}, \quad \ldots) \\
\vdash_{\mathcal{M}} & (\ldots, & u_i \mathsf{res}_{d,c}^{\{j_1,\ldots,j_r\}} v_i, & u_{j_1} \mathsf{req}_{d_1}^{i} v_{j_1}, & \ldots, & u_{j_r} \mathsf{req}_{d_r}^{i} v_{j_r}, \quad \ldots) \\
\vdash_{\mathcal{M}} & (\ldots, & u_i \mathsf{ack}_{d,c}^{\{j_1,\ldots,j_r\}} v_i, & u_{j_1} \mathsf{rec}_{d_1,c}^{i} v_{j_1}, & \ldots, & u_{j_r} \mathsf{rec}_{d_r,c}^{i} v_{j_r}, \quad \ldots) \\
\vdash_{\mathcal{M}} & (\ldots, & \kappa'_i, & \kappa'_{j_1}, & \ldots, & \kappa'_{j_r}, \quad \ldots).
\end{array}$$

With the above described modifications the system $\mathcal{M}'$ performs the following computation. The symbol $*$ denotes an arbitrary successor configuration of the according component.

$$\begin{array}{rllllllll}
 & (\ldots, & \kappa_i, & \kappa_{j_1}, & \kappa_{j_2}, & \ldots, & \kappa_{j_{r-1}}, & \kappa_{j_r}, & \ldots) \\
= & (\ldots, & u_i q_i v_i, & u_{j_1} q_{j_1} v_{j_1}, & u_{j_2} q_{j_2} v_{j_2}, & \ldots, & u_{j_{r-1}} q_{j_{r-1}} v_{j_{r-1}}, & u_{j_r} q_{j_r} v_{j_r}, & \ldots) \\
\vdash_{\mathcal{M}'} & (\ldots, & u_i \mathsf{res}_{d,c}^{j_1} v_i, & u_{j_1} \mathsf{req}_{d_1}^{i} v_{j_1}, & u_{j_2} \mathsf{req}_{d_2}^{i} v_{j_2}, & \ldots, & u_{j_{r-1}} \mathsf{req}_{d_{r-1}}^{i} v_{j_{r-1}}, & u_{j_r} \mathsf{req}_{d_r}^{i} v_{j_r}, & \ldots) \\
\vdash_{\mathcal{M}'} & (\ldots, & u_i \mathsf{ack}_{d,c}^{j_1} v_i, & u_{j_1} \mathsf{rec}_{d_1,c}^{i} v_{j_1}, & u_{j_2} \mathsf{req}_{d_2}^{i} v_{j_2}, & \ldots, & u_{j_{r-1}} \mathsf{req}_{d_{r-1}}^{i} v_{j_{r-1}}, & u_{j_r} \mathsf{req}_{d_r}^{i} v_{j_r}, & \ldots) \\
\vdash_{\mathcal{M}'} & (\ldots, & u_i \mathsf{res}_{d,c}^{j_2} v_i, & \kappa'_{j_1}, & u_{j_2} \mathsf{req}_{d_2}^{i} v_{j_2}, & \ldots, & u_{j_{r-1}} \mathsf{req}_{d_{r-1}}^{i} v_{j_{r-1}}, & u_{j_r} \mathsf{req}_{d_r}^{i} v_{j_r}, & \ldots) \\
\vdash_{\mathcal{M}'} & (\ldots, & u_i \mathsf{ack}_{d,c}^{j_2} v_i, & *, & u_{j_2} \mathsf{rec}_{d_2,c}^{i} v_{j_2}, & \ldots, & u_{j_{r-1}} \mathsf{req}_{d_{r-1}}^{i} v_{j_{r-1}}, & u_{j_r} \mathsf{req}_{d_r}^{i} v_{j_r}, & \ldots) \\
\vdash_{\mathcal{M}'} & (\ldots, & u_i \mathsf{res}_{d,c}^{j_3} v_i, & *, & \kappa'_{j_2}, & \ldots, & u_{j_{r-1}} \mathsf{req}_{d_{r-1}}^{i} v_{j_{r-1}}, & u_{j_r} \mathsf{req}_{d_r}^{i} v_{j_r}, & \ldots) \\
 & & & & \vdots & & & & \\
\vdash_{\mathcal{M}'} & (\ldots, & u_i \mathsf{res}_{d,c}^{j_r} v_i, & *, & *, & \ldots, & \kappa'_{j_{r-1}}, & u_{j_r} \mathsf{req}_{d_r}^{i} v_{j_r}, & \ldots) \\
\vdash_{\mathcal{M}'} & (\ldots, & u_i \mathsf{ack}_{d,c}^{j_r} v_i, & *, & *, & \ldots, & *, & u_{j_r} \mathsf{rec}_{d_r,c}^{i} v_{j_r}, & \ldots) \\
\vdash_{\mathcal{M}'} & (\ldots, & \kappa'_i, & *, & *, & \ldots, & *, & \kappa'_{j_r}, & \ldots).
\end{array}$$

Hence, each component of $\mathcal{M}'$ performs the same computation as the original component with respect to some additional communication steps. Particularly, for each component M_i, $1 \leq i \leq n$,

$$q_0^{(i)} ¢ w \$ \vdash^*_{M_i,\mathcal{M}} \kappa \quad \text{if and only if} \quad q_0^{(i)} ¢ w \$ \vdash^*_{M'_i,\mathcal{M}'} \kappa$$

for each configuration κ that does not contain a communication state. Since this holds for $\kappa = \mathsf{Accept}$ in particular, $L(\mathcal{M}) = L(\mathcal{M}')$ follows. □

The following example demonstrates how multicasts can be applied usefully. For this purpose we use our language $L_{\textit{c-copy}}$ defined above.

Example 11. We construct a system $\mathcal{M}_{\textit{c-copy}}$ that accepts the language $L_{\textit{c-copy}}$ with $c+1$ components M_1, M_2, $\ldots$, M_{c+1}. Basically, each component M_i, $1 \leq i \leq$

$c+1$, works on the i-th syllable of the input word. The first component M_1 moves over the ¢-symbol and then communicates each symbol via multicast to the other components $M_2, \ldots, M_{c+1}$ until it reads the #-symbol. Reading the first #-symbol signalizes the end of the first syllable. Each other component M_i, $2 \le i \le c+1$, moves its window to the right until it passes the $(i-1)$-th #-symbol, positions the window on the first symbol of the i-th syllable, and requests the symbol that is currently read by the first component. When receiving the corresponding symbol from M_1, M_i compares it with the own currently read symbol. If both symbols match, then M_i moves the window one step to the right and requests the next symbol from M_1. At the end of a computation, that is, when each of the first c components has reached the #-symbol, and M_{c+1} has reached the $-symbol, a last multicast communication step takes place to verify that indeed all the components have read their whole syllable. Only if this communication can be resolved, then all syllables of the input word are identical and M_1 accepts.
We define the components of $\mathcal{M}_{\textit{c-copy}}$ by

$$M_i = (Q_i, \{a, b, \#\}, \{a, b, \#\}, \text{¢}, \$, q_0, 1, \delta_i), 1 \le i \le c+1,$$

with

- $Q_1 = \{q_0, \mathsf{res}^*_a, \mathsf{ack}^*_a, \mathsf{res}^*_b, \mathsf{ack}^*_b, \mathsf{res}^*_\#, \mathsf{ack}^*_\#, \mathsf{res}^*_{\mathsf{acc}}, \mathsf{ack}^*_{\mathsf{acc}}\}$,
- $Q_i = \{q_0, \mathsf{req}^1, \mathsf{rec}^1_a, \mathsf{rec}^1_b, \mathsf{rec}^1_\#\} \cup \{q_j \mid 1 \le j \le i-1\}$ for all $2 \le i \le c+1$,

and

$$\begin{array}{llllll}
\delta_1(q_0, \text{¢}) & = (q_0, \mathsf{MVR}) & \delta_i(q_0, \text{¢}) & = (q_0, \mathsf{MVR}), & \\
\delta_1(q_0, a) & = \mathsf{res}^*_a, & \delta_i(q_j, a) & = (q_j, \mathsf{MVR}) & \text{for all } 0 \le j \le i-2, \\
\delta_1(q_0, b) & = \mathsf{res}^*_b, & \delta_i(q_j, b) & = (q_j, \mathsf{MVR}) & \text{for all } 0 \le j \le i-2, \\
\delta_1(q_0, \#) & = \mathsf{res}^*_\#, & \delta_i(q_j, \#) & = (q_{j+1}, \mathsf{MVR}) & \text{for all } 0 \le j \le i-2, \\
\delta_1(\mathsf{ack}^*_a, a) & = (q_0, \mathsf{MVR}), & \delta_i(q_{i-1}, a) & = \mathsf{req}^1, & \\
\delta_1(\mathsf{ack}^*_b, b) & = (q_0, \mathsf{MVR}), & \delta_i(q_{i-1}, b) & = \mathsf{req}^1, & \\
\delta_1(\mathsf{ack}^*_\#, \#) & = \mathsf{res}^*_{\mathsf{acc}}, & \delta_i(q_{i-1}, \#) & = \mathsf{req}^1 & \text{if } i < c+1, \\
\delta_1(\mathsf{ack}^*_{\mathsf{acc}}, \#) & = \mathsf{Accept}, & \delta_i(q_{i-1}, \$) & = \mathsf{req}^1 & \text{if } i = c+1, \\
 & & \delta_i(\mathsf{rec}^1_a, a) & = (q_{i-1}, \mathsf{MVR}), & \\
 & & \delta_i(\mathsf{rec}^1_b, b) & = (q_{i-1}, \mathsf{MVR}), & \\
 & & \delta_i(\mathsf{rec}^1_\#, \#) & = \mathsf{req}^1 & \text{if } i < c+1, \\
 & & \delta_i(\mathsf{rec}^1_\#, \$) & = \mathsf{req}^1 & \text{if } i = c+1.
\end{array}$$

Observe that no rewrite operations and no restart operations are used within the definition of the components. An example computation of $\mathcal{M}_{\textit{c-copy}}$ for $c = 2$ and the input word $ab\#ab\#ab$ is:

$$
\begin{array}{lllllll}
(& q_0\text{¢}ab\#ab\#ab\$ & , & q_0\text{¢}ab\#ab\#ab\$ & , & q_0\text{¢}ab\#ab\#ab\$ &) \\
\vdash (& \text{¢}q_0ab\#ab\#ab\$ & , & \text{¢}q_0ab\#ab\#ab\$ & , & \text{¢}q_0ab\#ab\#ab\$ &) \\
\vdash (& \text{¢}\mathsf{res}^*_a ab\#ab\#ab\$ & , & \text{¢}aq_0b\#ab\#ab\$ & , & \text{¢}aq_0b\#ab\#ab\$ &) \\
\vdash (& \text{¢}\mathsf{res}^*_a ab\#ab\#ab\$ & , & \text{¢}abq_0\#ab\#ab\$ & , & \text{¢}abq_0\#ab\#ab\$ &) \\
\vdash (& \text{¢}\mathsf{res}^*_a ab\#ab\#ab\$ & , & \text{¢}ab\#q_1ab\#ab\$ & , & \text{¢}ab\#q_1ab\#ab\$ &) \\
\vdash (& \text{¢}\mathsf{res}^*_a ab\#ab\#ab\$ & , & \text{¢}ab\#\mathsf{req}^1ab\#ab\$ & , & \text{¢}ab\#aq_1b\#ab\$ &) \\
\vdash (& \text{¢}\mathsf{res}^*_a ab\#ab\#ab\$ & , & \text{¢}ab\#\mathsf{req}^1ab\#ab\$ & , & \text{¢}ab\#abq_1\#ab\$ &) \\
\vdash (& \text{¢}\mathsf{res}^*_a ab\#ab\#ab\$ & , & \text{¢}ab\#\mathsf{req}^1ab\#ab\$ & , & \text{¢}ab\#ab\#q_2ab\$ &) \\
\vdash (& \text{¢}\mathsf{res}^*_a ab\#ab\#ab\$ & , & \text{¢}ab\#\mathsf{req}^1ab\#ab\$ & , & \text{¢}ab\#ab\#\mathsf{req}^1ab\$ &) \\
\vdash (& \text{¢}\mathsf{ack}^*_a ab\#ab\#ab\$ & , & \text{¢}ab\#\mathsf{rec}^1_a ab\#ab\$ & , & \text{¢}ab\#ab\#\mathsf{rec}^1_a ab\$ &) \\
\vdash (& \text{¢}aq_0b\#ab\#ab\$ & , & \text{¢}ab\#aq_1b\#ab\$ & , & \text{¢}ab\#ab\#aq_2b\$ &) \\
\vdash (& \text{¢}a\mathsf{res}^*_b b\#ab\#ab\$ & , & \text{¢}ab\#a\mathsf{req}^1b\#ab\$ & , & \text{¢}ab\#ab\#a\mathsf{req}^1b\$ &) \\
\vdash (& \text{¢}a\mathsf{ack}^*_b b\#ab\#ab\$ & , & \text{¢}ab\#a\mathsf{rec}^1_b b\#ab\$ & , & \text{¢}ab\#ab\#a\mathsf{rec}^1_b b\$ &) \\
\vdash (& \text{¢}abq_0\#ab\#ab\$ & , & \text{¢}ab\#abq_1\#ab\$ & , & \text{¢}ab\#ab\#abq_2\$ &) \\
\vdash (& \text{¢}ab\mathsf{res}^*_{\#}\#ab\#ab\$ & , & \text{¢}ab\#ab\mathsf{req}^1\#ab\$ & , & \text{¢}ab\#ab\#ab\mathsf{req}^1\$ &) \\
\vdash (& \text{¢}ab\mathsf{ack}^*_{\#}\#ab\#ab\$ & , & \text{¢}ab\#ab\mathsf{rec}^1_{\#}\#ab\$ & , & \text{¢}ab\#ab\#ab\mathsf{rec}^1_{\#}\$ &) \\
\vdash (& \text{¢}ab\mathsf{res}^*_{\mathsf{acc}}\#ab\#ab\$ & , & \text{¢}ab\#ab\mathsf{req}^1\#ab\$ & , & \text{¢}ab\#ab\#ab\mathsf{req}^1\$ &) \\
\vdash (& \text{¢}ab\mathsf{ack}^*_{\mathsf{acc}}\#ab\#ab\$ & , & \text{¢}ab\#ab\mathsf{rec}^1_{\mathsf{acc}}\#ab\$ & , & \text{¢}ab\#ab\#ab\mathsf{rec}^1_{\mathsf{acc}}\$ &) \\
\vdash (& \mathsf{Accept} & , & \text{¢}ab\#ab\mathsf{rec}^1_{\mathsf{acc}}\#ab\$ & , & \text{¢}ab\#ab\#ab\mathsf{rec}^1_{\mathsf{acc}}\$ &)
\end{array}
$$

If the input word is not of the correct form, that is, either the number of #-symbols is not equal to c or some syllables differ from each other in at least one symbol, then the following happens. In the first case, at least M_{c+1} gets stuck being in a state q_j with $0 \leq j \leq (i-2)$ and reading the \$-symbol or being in state q_c reading the #-symbol. Subsequently, at least one component cannot reach a communication state and no communication of M_1 is resolved. Whenever one syllable differs in at least one symbol from the first syllable, the corresponding component will get stuck, because $\delta_i(\mathsf{rec}^1_x, y)$ is not defined for two different tape symbols x and y. Thus, in either case M_1 cannot reach the accepting configuration, and the input word is rejected. □

5.2 Further examples

In Section 5.1.1 a first example for the PCRA systems was given that accepts the copy language with a middle marker. Now, some more examples are presented for typical languages appearing in formal language contexts and computational linguistic:

- Gladkij language:

$$L_{\mathit{Gladkij}} = \{w\#w^R\#w \mid w \in \{a,b\}^*\},$$

- language of multiple agreements:

$$L_{a^nb^nc^nd^n} = \{a^nb^nc^nd^n \mid n \geq 1\},$$

- copy language for constant many copies (c is a constant integer with $c \geq 1$):

$$L_{\mathit{c\text{-}copy}} = \{w(\#w)^c \mid w \in \{a,b\}^+\},$$

- copy language without middle marker (also called duplication language):

$$L_{ww} := \{ww \mid w \in \{a,b\}^*\},$$

- exponential language:

$$L_{\mathit{expo}} = \{a^{2^n} \mid n \geq 0\}.$$

The systems for these example languages will be explained rather detailed in order to illustrate how restarting automata work together within a PCRA system solving various tasks.

Example 12 (Gladkij language). A PCRA system that accepts the Gladkij language L_{Gladkij} consists of two det-mon-R-automata with window size three: $\mathcal{M}_{\mathit{Gladkij}} = (M_1, M_2)$. In each cycle component M_1 moves its window to the right until the first #-symbol occurs in the middle of the window. If it reads $a\#a$ or $b\#b$, this string is replaced by # followed by an immediate restart. Thus, M_1 checks whether the second syllable is the reversal of the first syllable. The second component M_2 behaves similarly, checking whether the third syllable is the reversal of the second one. When both automata are successful, i.e. M_1 reads ¢## and M_2 reads ##\$, then the one and only communication step takes place, whereafter M_1 and therefore the system $\mathcal{M}_{\mathit{Gladkij}}$ accepts the input word. If the input is not of the correct form, then M_1 or M_2 (or both) do not reach the communication state, the communication does not happen, and the system does not accept. Formally M_1 and M_2 are given by:

$$M_1 = (\{q_0, q_r, \mathsf{req}, \mathsf{rec}\}, \{a, b, \#\}, \{a, b, \#\}, ¢, \$, q_0, 3, \delta_1)$$

with

$$\begin{array}{lll}
\delta_1(q_0, \alpha) & = (q_0, \mathsf{MVR}) & \text{for all } \alpha \in \{¢, a, b\} \cdot \{a, b\} \cdot \{a, b, \#\}, \\
\delta_1(q_0, \alpha\#\alpha) & = (q_r, \#) & \text{for all } \alpha \in \{a, b\}, \\
\delta_1(q_r, \alpha) & = \mathsf{Restart} & \text{for all } \alpha \in \mathcal{PC}^{(3)}, \\
\delta_1(q_0, ¢\#\#) & = \mathsf{req}, & \\
\delta_1(\mathsf{rec}, ¢\#\#) & = \mathsf{Accept}, &
\end{array}$$

and

$$M_2 = (\{q_0, q_1, q_r, \mathsf{res}, \mathsf{ack}\}, \{a, b, \#\}, \{a, b, \#\}, ¢, \$, q_0, 3, \delta_2)$$

with

$$\begin{array}{lll}
\delta_2(q_0, \alpha) & = (q_0, \mathsf{MVR}) & \text{for all } \alpha \in \{¢, a, b\} \cdot \{a, b\} \cdot \{a, b, \#\}, \\
\delta_2(q_0, \alpha\#\#) & = (q_1, \mathsf{MVR}) & \text{for all } \alpha \in \{¢, a, b\}, \\
\delta_2(q_0, \alpha\#\alpha) & = (q_1, \mathsf{MVR}) & \text{for all } \alpha \in \{a, b\}, \\
\delta_2(q_1, \alpha) & = (q_1, \mathsf{MVR}) & \text{for all } \alpha \in \{a, b, \#\} \cdot \{a, b\} \cdot \{a, b, \#\}, \\
\delta_2(q_1, \alpha\#\alpha) & = (q_r, \#) & \text{for all } \alpha \in \{a, b\}, \\
\delta_2(q_r, \alpha) & = \mathsf{Restart} & \text{for all } \alpha \in \mathcal{PC}^{(3)}, \\
\delta_2(q_1, \#\#\$) & = \mathsf{res}. &
\end{array}$$

Observe that each computation of $\mathcal{M}_{Gladkij}$ contains at most one communication step, and this communication can only be executed in one particular situation (when M_1 reads $¢\#\#$ and M_2 reads $\#\#\$$). Thus, no local information has to be kept during the communication and no particular message has to be sent. Only the fact is important that some communication takes place. Therefore, the subscripts of the communication states can be omitted. Moreover, since for systems of two components the communication partner is always unique, the superscripts of the communication states can also be left out.

An example computation of $\mathcal{M}_{Gladkij}$ for the input word $ab\#ba\#ab$ is given as follows:

$$\begin{array}{ll|l}
 & (q_0\cent ab\#ba\#ab\$,\ q_0\cent ab\#ba\#ab\$) & \vdash (\mathsf{req}\cent\#\#ab\$,\ \cent abq_1\#b\#b\$) \\
\vdash & (\cent q_0ab\#ba\#ab\$,\ \cent q_0ab\#ba\#ab\$) & \vdash (\mathsf{req}\cent\#\#ab\$,\ \cent ab\#q_1b\#b\$) \\
\vdash & (\cent aq_0b\#ba\#ab\$,\ \cent aq_0b\#ba\#ab\$) & \vdash (\mathsf{req}\cent\#\#ab\$,\ \cent ab\#\#q_r\$) \\
\vdash & (\cent a\#q_ra\#ab\$,\quad \cent abq_1\#ba\#ab\$) & \vdash (\mathsf{req}\cent\#\#ab\$,\ q_0\cent ab\#\#\$) \\
\vdash & (q_0\cent a\#a\#ab\$,\quad \cent ab\#q_1ba\#ab\$) & \vdash (\mathsf{req}\cent\#\#ab\$,\ \cent q_0ab\#\#\$) \\
\vdash & (\cent q_0a\#a\#ab\$,\quad \cent ab\#bq_1a\#ab\$) & \vdash (\mathsf{req}\cent\#\#ab\$,\ \cent aq_0b\#\#\$) \\
\vdash & (\cent\#q_r\#ab\$,\qquad \cent ab\#b\#q_rb\$) & \vdash (\mathsf{req}\cent\#\#ab\$,\ \cent abq_1\#\#\$) \\
\vdash & (q_0\cent\#\#ab\$,\qquad q_0\cent ab\#b\#b\$) & \vdash (\mathsf{req}\cent\#\#ab\$,\ \cent ab\mathsf{res}\#\#\$) \\
\vdash & (\mathsf{req}\cent\#\#ab\$,\quad \cent q_0ab\#b\#b\$) & \vdash (\mathsf{rec}\cent\#\#ab\$,\ \cent ab\mathsf{ack}\#\#\$) \\
\vdash & (\mathsf{req}\cent\#\#ab\$,\quad \cent aq_0b\#b\#b\$) & \vdash (\mathsf{Accept},\qquad \cent ab\mathsf{ack}\#\#\$).
\end{array}$$

□

The following example shows how a system of three simple automata accepts the language of multiple agreements. Similarly to the previous example the components work mainly independently of each other and only at the end of a computation two communications are needed.

Example 13 ($L_{a^nb^nc^nd^n}$). Let $\mathcal{M}_{a^nb^nc^nd^n} = (M_1, M_2, M_3)$ be a PCRA system that accepts the language of multiple agreements

$$L_{a^nb^nc^nd^n} = \{a^nb^nc^nd^n \mid n \geq 1\}.$$

All components work independently of each other in the following way: M_1 checks whether the input is of the form $a^nb^nc^+d^+$, M_2 verifies that the input is of the form $a^+b^nc^nd^+$, and M_3 checks whether the input is of the form $a^+b^+c^nd^n$. If the input satisfies all three conditions, then two communications between M_1 and M_2 and between M_1 and M_3 take place, whereafter M_1 accepts. If the input word is not of the correct form, then at least one of the components halts without reaching the necessary communication state, the corresponding communication does not take place, and finally M_1 and thus the system $\mathcal{M}_{a^nb^nc^nd^n}$ do not accept. The components are now given by the following three det-R-automata with window size four:

$$M_1 = (\{q_0, q_1, q_r, \mathsf{req}^2, \mathsf{rec}^2, \mathsf{req}^3, \mathsf{rec}^3\}, \{a,b,c,d\}, \{a,b,c,d\}, \cent, \$, q_0, 4, \delta_1) \text{ with:}$$

$$\begin{aligned}
\delta_1(q_0,\alpha) &= (q_0,\mathsf{MVR}) && \text{f.a. } \alpha\in\{\cent a^3, \cent a^2b, a^4, a^3b\},\\
\delta_1(q_0,aabb) &= (q_r,ab),\\
\delta_1(q_0,\cent abc) &= (q_1,\mathsf{MVR}),\\
\delta_1(q_1,\alpha) &= (q_1,\mathsf{MVR}) && \text{f.a. } \alpha\in\{abc^2, abcd, bc^3, bc^2d, bcd^2, c^4, c^3d, c^2d^2, cd^3, d^4\},\\
\delta_1(q_1,\alpha) &= \mathsf{req}^2 && \text{f.a. } \alpha\in\{bcd\$, cd^2\$, d^3\$\},\\
\delta_1(q_r,\alpha) &= \mathsf{Restart} && \text{f.a. } \alpha\in\mathcal{PC}^{(4)},\\
\delta_1(\mathsf{rec}^2,\alpha) &= \mathsf{req}^3 && \text{f.a. } \alpha\in\{bcd\$, cd^2\$, d^3\$\},\\
\delta_1(\mathsf{rec}^3,\alpha) &= \mathsf{Accept} && \text{f.a. } \alpha\in\{bcd\$, cd^2\$, d^3\$\},
\end{aligned}$$

$$M_2 = (\{q_0,q_1,q_r,\mathsf{res}^1,\mathsf{ack}^1\},\{a,b,c,d\},\{a,b,c,d\},\cent,\$,q_0,4,\delta_2) \text{ with:}$$

$$\begin{aligned}
\delta_2(q_0,\alpha) &= (q_0,\mathsf{MVR}) && \text{f.a. } \alpha\in\{\cent a^3, \cent a^2b, \cent ab^2, \cent abc, a^4, a^3b, a^2b^2, a^2bc,\\
&&& \qquad ab^3, ab^2c, b^4, b^3c\},\\
\delta_2(q_0,bbcc) &= (q_r,bc),\\
\delta_2(q_0,abcd) &= (q_1,\mathsf{MVR}),\\
\delta_2(q_1,\alpha) &= (q_1,\mathsf{MVR}) && \text{f.a. } \alpha\in\{bcd^2, cd^3, d^4\},\\
\delta_2(q_1,\alpha) &= \mathsf{res}^1 && \text{f.a. } \alpha\in\{bcd\$, cd^2\$, d^3\$\},\\
\delta_2(q_r,\alpha) &= \mathsf{Restart} && \text{f.a. } \alpha\in\mathcal{PC}^{(4)},
\end{aligned}$$

$$M_3 = (\{q_0,q_r,\mathsf{res}^1,\mathsf{ack}^1\},\{a,b,c,d\},\{a,b,c,d\},\cent,\$,q_0,4,\delta_3) \text{ with:}$$

$$\begin{aligned}
\delta_3(q_0,\alpha) &= (q_0,\mathsf{MVR}) && \text{f.a. } \alpha\in\{\cent a^3, \cent a^2b, \cent abc, a^4, a^3b, a^2b^2, ab^3, a^2bc, b^4,\\
&&& \qquad ab^2c, b^3c, b^2c^2, b^2cd, bc^3, c^4, bc^2d, abcd, c^3d\},\\
\delta_3(q_0,ccdd) &= (q_r,cd),\\
\delta_3(q_0,bcd\$) &= \mathsf{res}^1,\\
\delta_3(q_r,\alpha) &= \mathsf{Restart} && \text{f.a. } \alpha\in\mathcal{PC}^{(4)}.
\end{aligned}$$

For the input $a^2b^2c^2d^2$ the system $\mathcal{M}_{a^nb^nc^nd^n}$ performs the following computation:

$$\begin{array}{rllll}
 & (q_0\text{¢}a^2b^2c^2d^2\$, & q_0\text{¢}a^2b^2c^2d^2\$, & q_0\text{¢}a^2b^2c^2d^2\$ &) \\
\vdash & (\text{¢}q_0a^2b^2c^2d^2\$, & \text{¢}q_0a^2b^2c^2d^2\$, & \text{¢}q_0a^2b^2c^2d^2\$ &) \\
\vdash & (\text{¢}abq_rc^2d^2\$ & ,\text{¢}aq_0ab^2c^2d^2\$, & \text{¢}aq_0ab^2c^2d^2\$ &) \\
\vdash & (q_0\text{¢}abc^2d^2\$ & ,\text{¢}a^2q_0b^2c^2d^2\$, & \text{¢}a^2q_0b^2c^2d^2\$ &) \\
\vdash & (\text{¢}q_1abc^2d^2\$ & ,\text{¢}a^2bcq_rd^2\$ & ,\text{¢}a^2bq_0bc^2d^2\$ &) \\
\vdash & (\text{¢}aq_1bc^2d^2\$ & ,q_0\text{¢}a^2bcd^2\$ & ,\text{¢}a^2b^2q_0c^2d^2\$ &) \\
\vdash & (\text{¢}abq_1c^2d^2\$ & ,\text{¢}q_0a^2bcd^2\$ & ,\text{¢}a^2b^2cdq_r\$ &) \\
\vdash & (\text{¢}abcq_1cd^2\$ & ,\text{¢}aq_0abcd^2\$ & ,q_0\text{¢}a^2b^2cd\$ &) \\
\vdash & (\text{¢}abc\mathsf{req}^2cd^2\$, & \text{¢}a^2q_1bcd^2\$ & ,\text{¢}q_0a^2b^2cd\$ &) \\
\vdash & (\text{¢}abc\mathsf{req}^2cd^2\$, & \text{¢}a^2bq_1cd^2\$ & ,\text{¢}aq_0ab^2cd\$ &) \\
\vdash & (\text{¢}abc\mathsf{req}^2cd^2\$, & \text{¢}a^2b\mathsf{res}^1cd^2\$ & ,\text{¢}a^2q_0b^2cd\$ &) \\
\vdash & (\text{¢}abc\mathsf{rec}^2cd^2\$, & \text{¢}a^2b\mathsf{ack}^1cd^2\$, & \text{¢}a^2bq_0bcd\$ &) \\
\vdash & (\text{¢}abc\mathsf{req}^3cd^2\$, & \text{¢}a^2b\mathsf{ack}^1cd^2\$, & \text{¢}a^2b\mathsf{res}^1bcd\$ &) \\
\vdash & (\text{¢}abc\mathsf{rec}^3cd^2\$, & \text{¢}a^2b\mathsf{ack}^1cd^2\$, & \text{¢}a^2b\mathsf{ack}^1bcd\$ &) \\
\vdash & (\mathsf{Accept} & ,\text{¢}a^2b\mathsf{ack}^1cd^2\$, & \text{¢}a^2b\mathsf{ack}^1bcd\$ &)
\end{array}$$

□

In Section 5.1.4 the language $L_{\textit{c-copy}}$ was used as an appropriate example for broadcast communication, where each of $c+1$ components work on its own part of the input and all syllables are compared with each other concurrently. Now we give a system with only two components with window size one. In contrast to the previous two examples, the components of the next system work closely together and use a high degree of communication.

Example 14 (det-global-PC-R(2,1)-system for $L_{\textit{c-copy}}$). The language

$$L_{\textit{c-copy}} = \{w(\#w)^c \mid w \in \{a,b\}^+\}$$

with constant $c \geq 1$ contains words that start with a non-empty string w over $\{a,b\}$ followed by c identical copies of w. Each copy is separated by the marker #. A system can be defined consisting of two components with window size one:

$$\mathcal{M}_{\textit{c-copy}} = (M_1, M_2)$$

with

$$\begin{aligned}
M_1 &= (Q_1, \{a,b,\#\}, \{a,b,\#\}, \text{¢}, \$, q_0, 1, \delta_1), \\
M_2 &= (Q_2, \{a,b,\#\}, \{a,b,\#\}, \text{¢}, \$, q_0, 1, \delta_2),
\end{aligned}$$

where

$$\begin{aligned} Q_1 &= \{q_j, \mathsf{req}_j, \mathsf{rec}_{j,a}, \mathsf{rec}_{j,b} \mid 0 \leq j < c\} \cup \{\mathsf{rec}_{j,\#} \mid 0 \leq j < c-1\} \cup \{\mathsf{rec}_{c-1,\$}\}, \\ Q_2 &= \{q_0, q_1\} \cup \{\mathsf{res}_x, \mathsf{ack}_x \mid x \in \{a, b, \#, \$\}\}, \end{aligned}$$

$$\begin{aligned} &\delta_1(q_0, ¢) = (q_0, \mathsf{MVR}), \\ &\delta_1(q_j, a) = \delta_1(q_j, b) = \delta_1(q_j, \#) = \mathsf{req}_j && \text{for all } 0 \leq j \leq c-1, \\ &\delta_1(\mathsf{rec}_{j,a}, a) = \delta_1(\mathsf{rec}_{j,b}, b) = (q_j, \mathsf{MVR}) && \text{for all } 0 \leq j \leq c-1, \\ &\delta_1(\mathsf{rec}_{j,\#}, \#) = (q_{j+1}, \mathsf{MVR}) && \text{for all } 0 \leq j \leq c-2, \\ &\delta_1(\mathsf{rec}_{c-1,\$}, \#) = \mathsf{Accept}, \end{aligned}$$

and

$$\begin{aligned} &\delta_2(q_0, ¢) = \delta_2(q_0, a) = \delta_2(q_0, b) = (q_0, \mathsf{MVR}), \\ &\delta_2(q_0, \#) = (q_1, \mathsf{MVR}), \\ &\delta_2(q_1, a) = \mathsf{res}_a, \\ &\delta_2(q_1, b) = \mathsf{res}_b, \\ &\delta_2(q_1, \#) = \mathsf{res}_\#, \\ &\delta_2(\mathsf{ack}_a, a) = \delta_2(\mathsf{ack}_b, b) = \delta_2(\mathsf{ack}_\#, \#) = (q_1, \mathsf{MVR}), \\ &\delta_2(q_1, \$) = \mathsf{res}_\$. \end{aligned}$$

First, M_2 moves its window to the right over the first syllable until it reads the first #-symbol. Then, both components move their windows synchronously to the right while comparing the currently read symbols through a communication in each step. Meanwhile, M_1 counts the number of #-symbols and stores it in the index of its state. If, at the end, each syllable is equal to its neighbouring syllable and, moreover, the number of counted #-symbols is equal to c, then M_1 accepts. Otherwise, if there occurs a mismatch of two currently read symbols, then the corresponding communication cannot be resolved, and thus, M_1 cannot reach the accepting configuration. Additionally, if the number of syllables is less than c, then M_1 rejects the input in a state $\mathsf{rec}_{j,\$}$ reading # for some $j < c-1$. If there are more than c syllables, then M_1 reaches the situation $(\mathsf{rec}_{c-1,\#}, \#)$ such that no transition is applicable. Thus, in this situation M_1 rejects, too. For an input w of the correct form, i.e. $w = u(\#u)^c = u_1u_2 \ldots u_l(\#u_1u_2 \ldots u_l)^c$ the system executes the following computation:

$$\begin{array}{llll}
 & (q_0\text{¢}u_1 \dots u_l\# \dots \#u\$ & , q_0\text{¢}u_1 \dots u_l\# \dots \#u\$ &) \\
\vdash & (\text{¢}q_0u_1 \dots u_l\# \dots \#u\$ & , \text{¢}q_0u_1 \dots u_l\# \dots \#u\$ &) \\
\vdash & (\text{¢}\mathsf{req}_0u_1 \dots u_l\# \dots \#u\$ & , \text{¢}u_1q_0u_2 \dots u_l\# \dots \#u\$ &) \\
\vdash^{l-1} & (\text{¢}\mathsf{req}_0u_1 \dots u_l\# \dots \#u\$ & , \text{¢}u_1 \dots u_lq_0\# \dots \#u\$ &) \\
\vdash & (\text{¢}\mathsf{req}_0u_1 \dots u_l\# \dots \#u\$ & , \text{¢}u\#q_1u_1 \dots u_l\# \dots \#u\$ &) \\
\vdash & (\text{¢}\mathsf{req}_0u_1 \dots u_l\# \dots \#u\$ & , \text{¢}u\#\mathsf{res}_{u_1}u_1 \dots u_l\# \dots \#u\$ &) \\
\vdash & (\text{¢}\mathsf{rec}_{0,u_1}u_1 \dots u_l\# \dots \#u\$ & , \text{¢}u\#\mathsf{ack}_{u_1}u_1 \dots u_l\# \dots \#u\$ &) \\
\vdash & (\text{¢}u_1q_0u_2 \dots u_l\# \dots \#u\$ & , \text{¢}u\#u_1q_1u_2 \dots u_l\# \dots \#u\$ &) \\
\vdash & (\text{¢}u_1\mathsf{req}_0u_2 \dots u_l\# \dots \#u\$ & , \text{¢}u\#u_1\mathsf{res}_{u_2}u_2 \dots u_l\# \dots \#u\$ &) \\
\vdash^{3(l-1)} & (\text{¢}u_1 \dots u_l\mathsf{req}_0\# \dots \#u\$ & , \text{¢}u\#u_1 \dots u_l\mathsf{res}_{\#}\# \dots \#u\$ &) \\
\vdash & (\text{¢}u_1 \dots u_l\mathsf{rec}_{0,\#}\# \dots \#u\$ & , \text{¢}u\#u_1 \dots u_l\mathsf{ack}_{\#}\# \dots \#u\$ &) \\
\vdash & (\text{¢}u_1 \dots u_l\#q_1u_1 \dots u_l\# \dots \#u\$, & \text{¢}u\#u_1 \dots u_l\#q_1u_1 \dots u_l\# \dots \#u\$ &) \\
\vdash^{(c-1)(3l+3)-3} & (\text{¢} \dots \#u_1 \dots u_l\mathsf{req}_{c-1}\#u\$ & , \text{¢}u\# \dots \#u_1 \dots u_l\mathsf{res}_{\$}\$ &) \\
\vdash & (\text{¢} \dots \#u_1 \dots u_l\mathsf{rec}_{c-1,\$}\#u\$ & , \text{¢}u\# \dots \#u_1 \dots u_l\mathsf{ack}_{\$}\$ &) \\
\vdash & (\mathsf{Accept} & , \text{¢}u\# \dots \#u_1 \dots u_l\mathsf{ack}_{\$}\$ &)
\end{array}$$

Hence $L(\mathcal{M}_{c\text{-}copy}) = L_{c\text{-}copy}$. □

The systems that are used to accept the languages $L_{w\#w}$ and $L_{c\text{-}copy}$ are globally deterministic, since each component is deterministic and the system accepts if and only if the first component accepts. At first sight it seems that the marker # is responsible for the fact that determinism suffices here. But the next example shows that this marker is not necessary.

Example 15 (Copy language without middle marker L_{ww}). The copy language without a middle marker

$$L_{ww} = \{ww \mid w \in \{a, b\}^*\}$$

is accepted by a det-global-PC-R-system $\mathcal{M}_{ww}$ consisting of three components M_1, M_2, and M_3 with window size one:

$$\begin{aligned}
\delta_1(q_0, \text{¢}) &= (q_0, \mathsf{MVR}),\\
\delta_1(q_0, a) &= \mathsf{req}^2,\\
\delta_1(q_0, b) &= \mathsf{req}^2,\\
\delta_1(q_0, \$) &= \mathsf{Accept},\\
\delta_1(\mathsf{rec}^2_{\mathsf{mvr}}, a) &= (q_0, \mathsf{MVR}),\\
\delta_1(\mathsf{rec}^2_{\mathsf{mvr}}, b) &= (q_0, \mathsf{MVR}),\\
\delta_1(\mathsf{rec}^2_{\mathsf{end}}, a) &= \mathsf{req}^3,\\
\delta_1(\mathsf{rec}^2_{\mathsf{end}}, b) &= \mathsf{req}^3,\\
\delta_1(\mathsf{rec}^3_{a}, a) &= (q_1, \mathsf{MVR}),\\
\delta_1(\mathsf{rec}^3_{b}, b) &= (q_1, \mathsf{MVR}),\\
\delta_1(q_1, a) &= \mathsf{req}^3,\\
\delta_1(q_1, b) &= \mathsf{req}^3,\\
\delta_1(q_1, \$) &= \mathsf{Accept},
\end{aligned}$$

$$\begin{aligned}
\delta_2(q_0, \text{¢}) &= (q_0, \mathsf{MVR}),\\
\delta_2(q_0, a) &= (q_1, \mathsf{MVR}),\\
\delta_2(q_0, b) &= (q_1, \mathsf{MVR}),\\
\delta_2(q_1, a) &= \mathsf{res}^1_{\mathsf{mvr}},\\
\delta_2(q_1, b) &= \mathsf{res}^1_{\mathsf{mvr}},\\
\delta_2(\mathsf{ack}^1_{\mathsf{mvr}}, a) &= (q_0, \mathsf{MVR}),\\
\delta_2(\mathsf{ack}^1_{\mathsf{mvr}}, b) &= (q_0, \mathsf{MVR}),\\
\delta_2(q_0, \$) &= \mathsf{res}^1_{\mathsf{end}},
\end{aligned}$$

$$\begin{aligned}
\delta_3(q_0, \text{¢}) &= (q_0, \mathsf{MVR}),\\
\delta_3(q_0, a) &= \mathsf{res}^1_{a},\\
\delta_3(q_0, b) &= \mathsf{res}^1_{b},\\
\delta_3(\mathsf{ack}^1_{a}, a) &= (q_0, \mathsf{MVR}),\\
\delta_3(\mathsf{ack}^1_{b}, b) &= (q_0, \mathsf{MVR}).
\end{aligned}$$

First, M_2 helps M_1 to find the middle of the input word. For doing so, M_2 moves to the right, and after every second move-right step, it sends a message to M_1. On receiving this message M_1 moves its window one step to the right. When M_2 reaches the right delimiter \$, it sends a message to M_1 telling it that it has reached the middle of the word. Then the window of M_1 is positioned on the first letter of the second half of the input, and the window of M_3 is positioned on the first symbol of the input. Now M_1 and M_3 read their current symbols and compare them through a communication. If these symbols coincide, then M_1 and M_3 both move their windows one position to the right. This continues until either a mismatch is found, in which case the corresponding communication cannot be resolved and the system halts without accepting, or until M_1 reaches the right delimiter \$, which means that the input was of the form ww, and then M_1 (and therewith the system) accepts. If the input word has odd length, then M_2 will detect this when encountering the \$-symbol being in the state q_1 instead of q_0, and the system gets stuck as well, since there is no transition defined for this situation.

For an input word $w = w_1 \dots w_{2l} = uu$, where $u \in \{a, b\}^*$, the system $\mathcal{M}_{ww}$ executes the following computation:

$$\begin{array}{lllll}
 & (q_0 ¢ w_1...w_{2l}\$ & , q_0 ¢ w_1...w_{2l}\$ & , q_0 ¢ w_1...w_{2l}\$ &) \\
\vdash & (¢ q_0 w_1...w_{2l}\$ & , ¢ q_0 w_1...w_{2l}\$ & , ¢ q_0 w_1...w_{2l}\$ &) \\
\vdash & (¢ \mathsf{req}^2 w_1...w_{2l}\$ & , ¢ w_1 q_1 w_2...w_{2l}\$ & , ¢ \mathsf{res}^1_{w_1} w_1...w_{2l}\$ &) \\
\vdash & (¢ \mathsf{req}^2 w_1...w_{2l}\$ & , ¢ w_1 \mathsf{res}^1_{\mathsf{mvr}} w_2...w_{2l}\$ & , ¢ \mathsf{res}^1_{w_1} w_1...w_{2l}\$ &) \\
\vdash & (¢ \mathsf{rec}^2_{\mathsf{mvr}} w_1...w_{2l}\$ & , ¢ w_1 \mathsf{ack}^1_{\mathsf{mvr}} w_2...w_{2l}\$ & , ¢ \mathsf{res}^1_{w_1} w_1...w_{2l}\$ &) \\
\vdash & (¢ w_1 q_0 w_2...w_{2l}\$ & , ¢ w_1 w_2 q_0 w_3...w_{2l}\$ & , ¢ \mathsf{res}^1_{w_1} w_1...w_{2l}\$ &) \\
\vdash^{4l-4} & (¢ w_1...w_l q_0 w_{l+1}...w_{2l}\$ & , ¢ w_1...w_{2l} q_0 \$ & , ¢ \mathsf{res}^1_{w_1} w_1...w_{2l}\$ &) \\
\vdash & (¢ w_1...w_l \mathsf{req}^2 w_{l+1}...w_{2l}\$ & , ¢ w_1...w_{2l} \mathsf{res}^1_{\mathsf{end}} \$ & , ¢ \mathsf{res}^1_{w_1} w_1...w_{2l}\$ &) \\
\vdash & (¢ w_1...w_l \mathsf{rec}^2_{\mathsf{end}} w_{l+1}...w_{2l}\$ & , ¢ w_1...w_{2l} \mathsf{ack}^1_{\mathsf{end}} \$ & , ¢ \mathsf{res}^1_{w_1} w_1...w_{2l}\$ &) \\
\vdash & (¢ w_1...w_l \mathsf{req}^3 w_{l+1}...w_{2l}\$ & , ¢ w_1...w_{2l} \mathsf{ack}^1_{\mathsf{end}} \$ & , ¢ \mathsf{res}^1_{w_1} w_1...w_{2l}\$ &) \\
\vdash & (¢ w_1...w_l \mathsf{rec}^3_{w_1} w_{l+1}...w_{2l}\$ & , ¢ w_1...w_{2l} \mathsf{ack}^1_{\mathsf{end}} \$ & , ¢ \mathsf{ack}^1_{w_1} w_1...w_{2l}\$ &) \\
\vdash & (¢ w_1...w_{l+1} q_1 w_{l+2}...w_{2l}\$ & , ¢ w_1...w_{2l} \mathsf{ack}^1_{\mathsf{end}} \$ & , ¢ w_1 q_0 w_2...w_{2l}\$ &) \\
\vdash & (¢ w_1...w_{l+1} \mathsf{req}^3 w_{l+2}...w_{2l}\$ & , ¢ w_1...w_{2l} \mathsf{ack}^1_{\mathsf{end}} \$ & , ¢ w_1 \mathsf{res}^1_{w_2} w_2...w_{2l}\$ &) \\
\vdash & (¢ w_1...w_{l+1} \mathsf{rec}^3_{w_2} w_{l+2}...w_{2l}\$ & , ¢ w_1...w_{2l} \mathsf{ack}^1_{\mathsf{end}} \$ & , ¢ w_1 \mathsf{ack}^1_{w_2} w_2...w_{2l}\$ &) \\
\vdash & (¢ w_1...w_{l+2} q_1 w_{l+3}...w_{2l}\$ & , ¢ w_1...w_{2l} \mathsf{ack}^1_{\mathsf{end}} \$ & , ¢ w_1 w_2 q_0 w_3...w_{2l}\$ &) \\
\vdash^{3l-6} & (¢ w_1...w_{2l} q_1 \$ & , ¢ w_1...w_{2l} \mathsf{ack}^1_{\mathsf{end}} \$ & , ¢ w_1...w_l q_0 w_{l+1}...w_{2l}\$ &) \\
\vdash & (\mathsf{Accept} & , ¢ w_1...w_{2l} \mathsf{ack}^1_{\mathsf{end}} \$ & , ¢ w_1...w_l \mathsf{res}^1_{w_{l+1}} w_{l+1}...w_{2l}\$ &)
\end{array}$$

□

All the example languages that we considered above are semi-linear. Now we define a PCRA system for the exponential language as a representative for non semi-linear languages.

Example 16 (Exponential language L_{expo}). Consider the language

$$L_{expo} = \{a^{2^n} \mid n \geq 0\}$$

that consists of all words with an exponential number of a's. This language can be accepted by a single det-RRWW-automaton given by the following meta-instructions:

$$\begin{array}{l}
(¢X^*, aa \to X, a^*\$), \\
(¢a^*, XX \to a, X^*\$), \\
(¢X\$, \mathsf{Accept}), \\
(¢a\$, \mathsf{Accept}).
\end{array}$$

However, the language L_{expo} cannot be accepted by any restarting automaton without auxiliary symbols[3]. This can easily be achieved using the correctness pre-

[3] Even for nonforgetting R- and RW-automata this can be found in [MO06].

serving property for a contradiction. Assume there would exist an RLW-automaton M accepting exactly the language L_{expo}. Then M must perform at least one rewrite step for a sufficiently large input word. Moreover, in each rewrite step M can just delete at most constantly many a's depending on the size of the window (no auxiliary symbols are allowed). But for each constant window size k, there exists an input word $w \in L_{expo}$ that is larger than $2k$ and thus cannot be reduced to one half in one rewrite step. Now, independently of how many symbols are deleted in one rewrite step, the resulting tape content w' is not contained in L_{expo}. Since M accepts L_{expo} and $w \in L_{expo}$, there must be an accepting computation for w. But then w' is also accepted by M, although $w' \notin L_{expo}$. This contradicts the fact that M accepts exactly L_{expo}. Hence, there cannot exist an RLW-automaton accepting L_{expo}.

Now we construct a PCRA system $\mathcal{M}_{expo} = (M_1, M_2)$ of the type det-global-PC-R with two components and window size two accepting L_{expo}. Basically, both components work in two alternating phases: 1) read the input and 2) halve the input. The first component starts with reading the input from left to right and for every second input symbol it informs the other component to delete one symbol. Thus, when M_1 has read its whole input word, then M_2 has halved the word on its working tape. Then both components swap their role and work in the according other phase. Now, M_2 reads the input from left to right and for each read symbol, M_1 deletes one a. After M_1 has worked exactly one time in the reading phase and in the halving phase and M_2 has in parallel worked in the halving phase and in the reading phase, both components have reduced the words on their working tapes exactly by one half. Now the same computation is done again with the remaining part of the input. At the end, M_1 accepts if and only if exactly one a remains on the tape.

Technically, at the end of each reading phase one symbol is deleted to perform a restart operation. This must be taken into account in the subsequent halving phase. For an input word a^{16} the sequence of the phases can be given as follows:

$$\begin{array}{l} M_1 : a^{16} \xrightarrow{count} a^{15} \xrightarrow{delete} a^{8} \xrightarrow{count} a^{7} \xrightarrow{delete} a^{4} \xrightarrow{count} a^{3} \xrightarrow{delete} a^{2} \xrightarrow{count} a^{1} \xrightarrow{delete} a^{1} \\ M_2 : a^{16} \xrightarrow{delete} a^{9} \xrightarrow{count} a^{8} \xrightarrow{delete} a^{5} \xrightarrow{count} a^{4} \xrightarrow{delete} a^{3} \xrightarrow{count} a^{2} \xrightarrow{delete} a^{2} \xrightarrow{count} a^{1}. \end{array}$$

Moreover, some communication overhead is necessary for agreeing to the role of each component. Particularly, after each performed restart operation a component has to obtain the information about its current role from the other component. Now we give the formal definition of $\mathcal{M}_{expo}$, and afterwards some example computations are given for a better understanding of the system. The symbols used within the subscripts have the following meaning: W stands for the question

‘What to do?’. It signalizes that the roles of the components have to be agreed on. The symbol R is used as a local information or is sent as a message. In the former case it means that the first component is still in the reading (counting) phase and in the latter case M_2 informs M_1 about its current role. The D as a local information means that M_1 is currently in the deleting phase. As a message it is a call for deleting a symbol a. The message Ch is used to signalize the change of the phases, i.e. M_1 changes from the reading phase into the deleting phase.

$$\begin{aligned} M_1 &= (Q_1, \{a\}, \{a\}, \text{¢}, \$, \mathsf{res}_W, 2, \delta_1), \\ M_2 &= (Q_2, \{a\}, \{a\}, \text{¢}, \$, \mathsf{req}_R, 2, \delta_2), \end{aligned}$$

where

$$\begin{aligned} Q_1 &= \{\mathsf{res}_W, \mathsf{ack}_W, \mathsf{req}, \mathsf{rec}_R, \mathsf{rec}_D, p_0, p_1, p_2, p_r, \mathsf{res}_D, \mathsf{ack}_D, \mathsf{res}_{Ch}, \mathsf{ack}_{Ch}\}, \\ Q_2 &= \{\mathsf{req}_R, \mathsf{rec}_{R,W}, \mathsf{rec}_{R,D}, \mathsf{rec}_{R,Ch}, \mathsf{res}_R, \mathsf{ack}_R, \mathsf{res}_D, \mathsf{ack}_D, \mathsf{req}_D, \mathsf{rec}_{D,W}, p_1, p_2, p_r\}, \end{aligned}$$

and

$$\begin{array}{llllll}
\delta_1(\mathsf{ack}_W, \text{¢}a) & = \mathsf{req}, & & \delta_2(\mathsf{rec}_{R,W}, \text{¢}a) & = \mathsf{res}_R, \\
\delta_1(\mathsf{rec}_R, \text{¢}a) & = (p_0, \mathsf{MVR}), & & \delta_2(\mathsf{ack}_R, \text{¢}a) & = \mathsf{req}_R, \\
\delta_1(p_0, aa) & = (p_2, \mathsf{MVR}), & & \delta_2(\mathsf{rec}_{R,Ch}, \text{¢}a) & = (p_1, \mathsf{MVR}), \\
\delta_1(p_0, a\$) & = \mathsf{Accept}, & & \delta_2(p_1, aa) & = (p_2, \mathsf{MVR}), \\
\delta_1(p_1, aa) & = (p_2, \mathsf{MVR}), & & \delta_2(p_2, aa) & = \mathsf{req}_D, \\
\delta_1(p_2, aa) & = \mathsf{res}_D, & & \delta_2(\mathsf{rec}_{D,W}, aa) & = \mathsf{res}_D, \\
\delta_1(\mathsf{ack}_D, aa) & = (p_1, \mathsf{MVR}), & & \delta_2(\mathsf{ack}_D, aa) & = (p_2, \mathsf{MVR}), \\
\delta_1(p_2, a\$) & = \mathsf{res}_{Ch}, & & \delta_2(p_2, a\$) & = (p_r, \$), \\
\delta_1(\mathsf{ack}_{Ch}, a\$) & = (p_r, \$), & & \delta_2(\mathsf{rec}_{R,D}, \text{¢}a) & = (p_r, \text{¢}), \\
\delta_1(\mathsf{rec}_D, \text{¢}a) & = (p_r, \text{¢}), & & \delta_2(p_r, \alpha) & = \mathsf{Restart}, \\
\delta_1(p_r, \alpha) & = \mathsf{Restart}, & & &
\end{array}$$

for all $\alpha \in \{\text{¢}\$, \text{¢}a, aa, a\$, \$\}$. For small inputs $\mathcal{M}_{expo}$ behaves as follows. If the input is ε, then both components get stuck immediately, since no transition can be applied while reading ¢\$. If the input is $a = a^{2^0}$ or $aa = a^{2^1}$, then the following computations are executed:

$$
\begin{array}{rllll|rlllll}
 & (& \mathsf{res}_W\text{¢}a\$\,, & \mathsf{req}_R\text{¢}a\$ &) & & (& \mathsf{res}_W\text{¢}aa\$\;, & \mathsf{req}_R\text{¢}aa\$ &) & \\
\vdash & (& \mathsf{ack}_W\text{¢}a\$, & \mathsf{rec}_{R,W}\text{¢}a\$ &) & \vdash & (& \mathsf{ack}_W\text{¢}aa\$\,, & \mathsf{rec}_{R,W}\text{¢}aa\$ &) & \\
\vdash & (& \mathsf{req}\text{¢}a\$ \;\;, & \mathsf{res}_R\text{¢}a\$ &) & \vdash & (& \mathsf{req}\text{¢}aa\$ \;\;, & \mathsf{res}_R\text{¢}aa\$ &) & \\
\vdash & (& \mathsf{rec}_R\text{¢}a\$\,, & \mathsf{ack}_R\text{¢}a\$ &) & \vdash & (& \mathsf{rec}_R\text{¢}aa\$\;, & \mathsf{ack}_R\text{¢}aa\$ &) & \\
\vdash & (& \text{¢}p_0a\$ \;\;, & \mathsf{req}_R\text{¢}a\$ &) & \vdash & (& \text{¢}p_0aa\$ \;\;, & \mathsf{req}_R\text{¢}aa\$ &) & \\
\vdash & (& \mathsf{Accept} \;\;, & \mathsf{req}_R\text{¢}a\$ &) & \vdash & (& \text{¢}ap_2a\$ \;\;, & \mathsf{req}_R\text{¢}aa\$ &) & \\
 & & & & & \vdash & (& \text{¢}a\mathsf{res}_{Ch}a\$\,, & \mathsf{req}_R\text{¢}aa\$ &) & \\
 & & & & & \vdash & (& \text{¢}a\mathsf{ack}_{Ch}a\$, & \mathsf{rec}_{R,Ch}\text{¢}aa\$ &) & \\
 & & & & & \vdash & (& \text{¢}ap_r\$ \;\;, & \text{¢}p_1aa\$ &) & \\
 & & & & & \vdash & (& \mathsf{res}_W\text{¢}a\$ \;\;, & \text{¢}ap_2a\$ &) & \\
 & & & & & \vdash & (& \mathsf{res}_W\text{¢}a\$ \;\;, & \text{¢}ap_r\$ &) & \\
 & & & & & \vdash & (& \mathsf{res}_W\text{¢}a\$ \;\;, & \mathsf{req}_R\text{¢}a\$ &) & \\
 & & & & & \vdash^* & (& \mathsf{Accept} \;\;, & \mathsf{req}_R\text{¢}a\$ &) & (*),
\end{array}
$$

where the part of the computation marked with $(*)$ is the same as that for the input a given at the left-hand side. For input words of the form a^{2^r} with $r \geq 2$, the system performs the following computation, where the vertical dots signalize that the parts above marked with $(*)$ are executed all in all $2^{r-1} - 2$ times. The left-hand part of the computation describes the reading phase of component M_1 (where the component M_2 is in the deleting phase), and the right-hand part of the computation describes M_1's deleting phase (where M_2 is in the reading/counting phase):

$$\begin{array}{lll}
 & (\mathsf{res}_W\text{¢}a^{2^r}\$ & ,\mathsf{req}_R\text{¢}a^{2^r}\$\qquad) \\
\vdash & (\mathsf{ack}_W\text{¢}a^{2^r}\$ & ,\mathsf{rec}_{R,W}\text{¢}a^{2^r}\$\qquad) \\
\vdash & (\mathsf{req}\text{¢}a^{2^r}\$ & ,\mathsf{res}_R\text{¢}a^{2^r}\$\qquad) \\
\vdash & (\mathsf{rec}_R\text{¢}a^{2^r}\$ & ,\mathsf{ack}_R\text{¢}a^{2^r}\$\qquad) \\
\vdash & (\text{¢}p_0a^{2^r}\$ & ,\mathsf{req}_R\text{¢}a^{2^r}\$\qquad) \\
\vdash & (\text{¢}ap_2a^{2^r-1}\$ & ,\mathsf{req}_R\text{¢}a^{2^r}\$\qquad) \\
\vdash & (\text{¢}a\mathsf{res}_Da^{2^r-1}\$ & ,\mathsf{req}_R\text{¢}a^{2^r}\$\qquad) \\
\vdash & (\text{¢}a\mathsf{ack}_Da^{2^r-1}\$ & ,\mathsf{rec}_{R,D}\text{¢}a^{2^r}\$\qquad) \\
\vdash & (\text{¢}a^2p_1a^{2^r-2}\$ & ,\text{¢}p_ra^{2^r-1}\$\qquad)(*) \\
\vdash & (\text{¢}a^3p_2a^{2^r-3}\$ & ,\mathsf{req}_R\text{¢}a^{2^r-1}\$\qquad)(*) \\
\vdash & (\text{¢}a^3\mathsf{res}_Da^{2^r-3}\$ & ,\mathsf{req}_R\text{¢}a^{2^r-1}\$\qquad)(*) \\
\vdash & (\text{¢}a^3\mathsf{ack}_Da^{2^r-3}\$ & ,\mathsf{rec}_{R,D}\text{¢}a^{2^r-1}\$\qquad)(*) \\
 & \vdots & \\
\vdash & (\text{¢}a^{2^r-3}\mathsf{ack}_Da^3\$ & ,\mathsf{rec}_{R,D}\text{¢}a^{2^{r-1}+2}\$\qquad) \\
\vdash & (\text{¢}a^{2^r-2}p_1a^2\$ & ,\text{¢}p_ra^{2^{r-1}+1}\$\qquad) \\
\vdash & (\text{¢}a^{2^r-1}p_2a\$ & ,\mathsf{req}_R\text{¢}a^{2^{r-1}+1}\$\qquad) \\
\vdash & (\text{¢}a^{2^r-1}\mathsf{res}_{Ch}a\$ & ,\mathsf{req}_R\text{¢}a^{2^{r-1}+1}\$\qquad) \\
\vdash & (\text{¢}a^{2^r-1}\mathsf{ack}_{Ch}a\$ & ,\mathsf{rec}_{R,Ch}\text{¢}a^{2^{r-1}+1}\$) \\
\vdash & (\text{¢}a^{2^r-1}p_r\$ & ,\text{¢}p_1a^{2^{r-1}+1}\$\qquad) \\
\vdash & (\mathsf{res}_W\text{¢}a^{2^r-1}\$ & ,\text{¢}ap_2a^{2^{r-1}}\$\qquad)
\end{array}$$

$$\begin{array}{lll}
\vdash & (\mathsf{res}_W\text{¢}a^{2^r-1}\$ & ,\text{¢}a\mathsf{req}_Da^{2^{r-1}}\$\qquad) \\
\vdash & (\mathsf{ack}_W\text{¢}a^{2^r-1}\$ & ,\text{¢}a\mathsf{rec}_{D,W}a^{2^{r-1}}\$\qquad) \\
\vdash & (\mathsf{req}\text{¢}a^{2^r-1}\$ & ,\text{¢}a\mathsf{res}_Da^{2^{r-1}}\$\qquad)(*) \\
\vdash & (\mathsf{rec}_D\text{¢}a^{2^r-1}\$ & ,\text{¢}a\mathsf{ack}_Da^{2^{r-1}}\$\qquad)(*) \\
\vdash & (\text{¢}p_ra^{2^r-2}\$ & ,\text{¢}a^2p_2a^{2^{r-1}-1}\$\qquad)(*) \\
\vdash & (\mathsf{res}_W\text{¢}a^{2^r-2}\$ & ,\text{¢}a^2\mathsf{req}_Da^{2^{r-1}-1}\$\qquad)(*) \\
\vdash & (\mathsf{ack}_W\text{¢}a^{2^r-2}\$ & ,\text{¢}a^2\mathsf{rec}_{D,W}a^{2^{r-1}-1}\$)(*) \\
 & \vdots & \\
\vdash & (\mathsf{ack}_W\text{¢}a^{2^{r-1}+1}\$ & ,\text{¢}a^{2^{r-1}-1}\mathsf{rec}_{D,W}a^2\$) \\
\vdash & (\mathsf{req}\text{¢}a^{2^{r-1}+1}\$ & ,\text{¢}a^{2^{r-1}-1}\mathsf{res}_Da^2\$\qquad) \\
\vdash & (\mathsf{rec}_D\text{¢}a^{2^{r-1}+1}\$ & ,\text{¢}a^{2^{r-1}-1}\mathsf{ack}_Da^2\$\qquad) \\
\vdash & (\text{¢}p_ra^{2^{r-1}}\$ & ,\text{¢}a^{2^{r-1}}p_2a\$\qquad) \\
\vdash & (\mathsf{res}_W\text{¢}a^{2^{r-1}}\$ & ,\text{¢}a^{2^{r-1}}p_r\$\qquad) \\
\vdash & (\mathsf{res}_W\text{¢}a^{2^{r-1}}\$ & ,\mathsf{req}_R\text{¢}a^{2^{r-1}}\$\qquad)\ .
\end{array}$$

If the input is not of the form a^{2^r}, then the configuration $(\mathsf{res}_W\text{¢}a^t\$, \mathsf{req}_R\text{¢}a^t\$)$ appears at a point during the computation with an odd number t of a's on the working tape. But then M_1 will get stuck while reading $a\$$ in state p_1 at some time. For that case M_1 cannot accept the input, and the whole system does not accept, since M_2 cannot reach the accepting configuration for itself. Thus, it follows that $L(\mathcal{M}_{expo}) = L_{expo}$. □

5.3 Centralized versus non-centralized PCRA systems

Regarding distributed computing the question of an appropriate communication structure is important. One crucial aspect is the issue of whether the communication is centralized or non-centralized. All components of a centralized system are only allowed to communicate with a distinguished master component, whereas in a non-centralized communication structure each component can communicate with all other components.

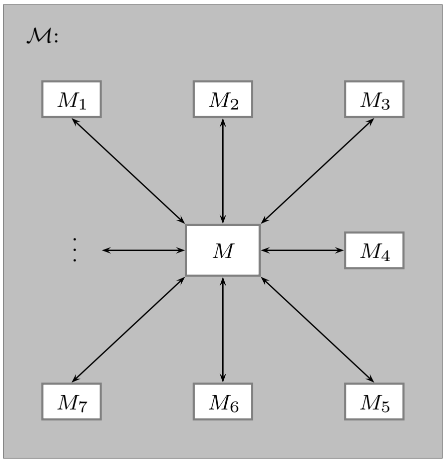

Figure 5.2: A centralized system $\mathcal{M}$ with the master component M and client components M_1, M_2, etc. The arrows show the allowed connections of communication.

That non-centralized systems are at least as powerful as centralized systems results from the fact that a centralized system can be seen as a special case of a non-centralized system (for systems of finite and pushdown automata see e.g. [CMMV00, MMM02]). But does the centralization as a restriction of the communication structure yield a proper decrease in computational power? For example, this happens for deterministic parallel communicating finite automata [BKM08] and PC grammar systems with regular or linear components [CDKP94]. Within this section it will be shown that in the case of PCRA systems, centralized and non-centralized systems have the same computational power independent of the type of the components.

A *centralized* PC-X-system ($\mathsf{X} \in \mathcal{T}$), cPC-X-system for short, is a PC-X-system in which every component is only allowed to communicate with the first component (the master component). Thereby it is not important whether the communication is initiated by the master or a client, that is, whether the master or a client sends a request. In centralized systems the superscript of the communication states (that denotes the receiver) of the components can be omitted (except for the master).

The set of languages accepted by cPC-X-systems is denoted by $\mathcal{L}$(cPC-X).

Theorem 4. *$\mathcal{L}$(cPC-X) = $\mathcal{L}$(PC-X) for all $\mathsf{X} \in \mathcal{T}_R$.*

Proof. Let $\mathsf{X} \in \mathcal{T}_R$ be a type of restarting automaton that does not allow to use MVL operations. $\mathcal{L}$(cPC-X) $\subseteq \mathcal{L}$(PC-X) obviously holds. It remains to show that $\mathcal{L}$(PC-X) $\subseteq \mathcal{L}$(cPC-X). We prove that for every PCRA system there exists an equivalent centralized system of the same type. Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be an arbitrary PC-X-system of degree n, and let $M_i = (Q_i, \ \Sigma, \Gamma_i, ¢, \$, q_0^{(i)}, k, \delta_i)$ $(1 \leq i \leq n)$. We construct a centralized PC-X-system $\mathcal{M}' = (M, M_1', \ M_2', \ldots, M_n')$ with $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$, where M is a new master component that solely controls all communications. While M_1,...,M_n can communicate with each other, M_1',...,M_n' are not allowed to do so. Now, we first define the clients formally and then give a description of their behaviour referring to the particular aspects of the construction. Afterwards we do the same with the new master component. A new component M_i' $(1 \leq i \leq n)$ is obtained by modifying the transition relation of M_i as follows ($q, q' \in Q_i$; $\alpha \in \mathcal{PC}^{(k)}$; $j \in \{1, 2, \ldots, n\}$; c, d are strings of constant length):

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

A2. for all $\delta_i(q, \alpha) = \emptyset$, $\delta_i'(q, \alpha) = \{\mathsf{res}_{[\perp]}\}$,

A3. for all $\mathsf{Accept} \in \delta_i(q, \alpha)$,

   (a) $\mathsf{res}_{[Accept]} \in \delta_i'(q, \alpha)$, if $\mathcal{M}$ is not globally deterministic or if $i = 1$,

   (b) $\mathsf{res}_{[\perp]} \in \delta_i'(q, \alpha)$, if $\mathcal{M}$ is globally deterministic and $i > 1$,

A4. 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 is not sent to the master. After telling the master the current communication state, M_i' requests the information from the master whether the communication partner has sent a corresponding communication state. If the communication partner does not or will not, M_i' is stuck like M_i in this situation.),

A5. for $A \in \delta_i(\mathsf{rec}_{d,c}^j, \alpha)$, $A \in \delta_i'(\mathsf{rec}_{[req(j,d)],[res(j,c)]}, \alpha)$
(The action A can be performed by M_i' when the master tells M_i' that

the communication partner M'_j has sent the corresponding communication state.),

A6. 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
- $\mathsf{req}_{[res(j,d,c)]} \in \delta'_i(\mathsf{ack}_{[res(j,d,c)],[res(j,c)]}, \alpha)$

(Similar to A4.),

A7. for $A \in \delta_i(\mathsf{ack}^j_{d,c}, \alpha)$, $A \in \delta'_i(\mathsf{rec}_{[res(j,d,c)],[req(j)]}, \alpha)$
(Similar to A5.).

The operations MVR, $\mathsf{Restart}$, and rewrite are retained unchanged (A1). Furthermore, observe that there are exactly three ways in which a local computation of a component can finish: accept (A3), being stuck (A2), or reaching a communication state (A4, A6). For these situations the component sends a corresponding information to the master by entering a response state: $[Accept]$, $[\perp]$, $[req(j)]$ or $[res(j, c)]$. With respect to acceptance, two cases have to be distinguished: If the accepting component is not the first one within a globally deterministic system, this is similar to the case of being stuck, and therefore the information $[\perp]$ is sent. Otherwise the system should accept, hence the information $[Accept]$ is posted. After sending the response, the component either is stuck (if it has sent $[Accept]$ or $[\perp]$) or enters a request state to await information from the master about the currently simulated communication step, in particular if the communication partner reaches the corresponding communication state. Moreover, the original communication states are included in the subscript (the local information) of the new communication states in order to simulate the communication step in a unique manner.
The sets of states of the modified components are given indirectly through the definition of the transition function. The initial state of M'_i is the same as that of M_i except if it is a communication state. If the initial state of M_i is req^j_d ($\mathsf{res}^j_{d,c}$), then the initial state of M'_i is $\mathsf{res}_{[req(j,d)],[req(j)]}$ ($\mathsf{res}_{[res(j,d,c)],[res(j,c)]}$).

The master component M has only communication states. In their indices (as local information) these states contain a tuple of situations, one for each component. E.g. the tuple $\langle *, \perp, res(1, c)\rangle$ describes the fact that the master does not (yet) know the current situation of M'_1 (the master still has to ask for it), M'_2 is stuck, and M'_3 wants to response to M'_1 with the information c.
The initial state of the master component is $\mathsf{req}^1_{\langle *,\dots,*\rangle}$. Thus, the master does not have any information about the current situations of the other components, and

first M'_1 is asked for the result of its local computation. Observe that the index of the communication partner in the superscript of the communication states indeed refers to the index of the corresponding component and not to its position within the tuple of the system. In the following computation the master cyclically asks all components that did not send the information $[\perp]$ before (see B2) or wait in a communication situation. This means that only components are asked for which $*$ is listed in the situation tuple. In a situation $\langle t_1, t_2, \ldots, t_n\rangle$ the successor of a component M'_m is denoted with $M'_{m'}$, and is obtained as follows: Let $N_1 = \{i \mid m < i \leq n \wedge t_i = *\}$ 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 does not exist a successor (there is no $*$ left in the situation tuple, hence $N_1 = N_2 = \emptyset$), the system is stuck and rejects the input. In addition, observe that if the successor m' exists, it is obtained deterministically.

Since the communication in a centralized system is only allowed with the master component, the original communication steps between two arbitrary components have to be simulated. Now, the master controls and forwards all communication demands. The information about the former communication demand (a component sends a response with an information $[req(j)]$ or $[res(j, c)]$) is at first stored within the situation tuple of the master (see B3 'else', B6 'else'). When the master receives a corresponding message from the communication partner, it immediately informs both communication partners about the respective other message (see B3 'if'-B5, B6 'if'-B8). After performing the communication step, $*$ is stored in the situation tuple for both communication partners, and the computation goes on with asking the next component.

If the master receives the information $[Accept]$ from any component, it accepts the input itself, and therefore the whole system accepts (see B1). Now the formal definition of the transition function of the master component is given (the transitions are defined for all $1 \leq m \leq n$):

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

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

It remains to show that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$ holds. First, the behaviour of the modified components ($M_i', i \in \{1,2,\ldots,n\}$) according to the local computations is exactly like that of the original components (because of step A1 within the construction). This means that for all $i \in \{1,2,\ldots,n\}$ and all configurations κ, κ',

$$\kappa \vdash_{M_i} \kappa' \Leftrightarrow \kappa \vdash_{M_i'} \kappa',$$

and therefore

$$\kappa \vdash^*_{M_i} \kappa' \Leftrightarrow \kappa \vdash^*_{M_i'} \kappa',$$

if κ' is achieved from κ through MVR, $\mathsf{Restart}$, and rewrite operations. The communication between two components achieves the same result in $\mathcal{M}'$ as in $\mathcal{M}$. Let M_i and M_j be two components of $\mathcal{M}$ that perform a communication. A computation looks as follows, where $\kappa_{r,s}$ denotes a configuration (r and s stand for the computation step and the number of the component, respectively):

$$\begin{array}{ll} & (\kappa_{1,1},\kappa_{1,2},\ldots,\kappa_{1,n}) \\ \vdash_{\mathcal{M}} & (\kappa_{2,1},\kappa_{2,2},\ldots,\kappa_{2,i-1},u\mathsf{req}^{j}_{d}v,\kappa_{2,i+1},\ldots,\kappa_{2,j-1},u'\mathsf{res}^{i}_{d',c}v',\kappa_{2,j+1},\ldots,\kappa_{2,n}) \\ \vdash_{\mathcal{M}} & (\kappa_{3,1},\kappa_{3,2},\ldots,\kappa_{3,i-1},u\mathsf{rec}^{j}_{d,c}v,\kappa_{3,i+1},\ldots,\kappa_{3,j-1},u'\mathsf{ack}^{i}_{d',c}v',\kappa_{3,j+1},\ldots,\kappa_{3,n}) \\ \vdash_{\mathcal{M}} & (\kappa_{4,1},\kappa_{4,2},\ldots,\kappa_{4,i-1},K_i,\kappa_{4,i+1},\ldots,\kappa_{4,j-1},K_j,\kappa_{4,j+1},\ldots,\kappa_{4,n}). \end{array}$$

The system $\mathcal{M}'$ simulates this communication in the following way (The annotations A1, A2,…, B1,… refer to the construction of the modified components and the new master component above. The contents of the variables $t_1,\ldots,t_n$ used in the situation tuples can differ from computation step to computation step. Due to clarity it was avoided to write $t_{1,1},\ldots,t_{1,n},t_{2,1},\ldots,t_{2,n},\ldots$ for the different computation steps. Moreover, κ' is used instead of κ for separation from the communication step of the original system $\mathcal{M}$.):

$$
\begin{aligned}
&(\kappa'_{1,0}, \kappa'_{1,1}, \kappa'_{1,2}, \ldots, \kappa'_{1,n})\\
\vdash_{\mathcal{M}'}\ &(\kappa'_{2,0}, \kappa'_{2,1}, \ldots, \kappa'_{2,i-1}, u\mathsf{res}_{[req(j,d)],[req(j)]}v, \kappa'_{2,i+1}, \ldots, \kappa'_{2,n})\\
&\text{(because of A4)}\\
\vdash^*_{\mathcal{M}'}\ &(\mathsf{req}^i_{\langle t_1,\ldots,t_{i-1},*,t_{i+1},\ldots,t_n\rangle}\text{¢}w\$, \kappa'_{3,1}, \ldots, \kappa'_{3,i-1}, u\mathsf{res}_{[req(j,d)],[req(j)]}v, \kappa'_{3,i+1}, \ldots, \kappa'_{3,n})\\
\vdash_{\mathcal{M}'}\ &(\mathsf{rec}^i_{\langle t_1,\ldots,t_{i-1},*,t_{i+1},\ldots,t_n\rangle,[req(j)]}\text{¢}w\$, \kappa'_{4,1},\\
&\qquad \ldots, \kappa'_{4,i-1}, u\mathsf{ack}_{[req(j,d)],[req(j)]}v, \kappa'_{4,i+1}, \ldots, \kappa'_{4,n})\\
\vdash_{\mathcal{M}'}\ &(\mathsf{req}^m_{\langle t_1,\ldots,t_{i-1},req(j),t_{i+1},\ldots,t_n\rangle}\text{¢}w\$, \kappa'_{5,1}, \ldots, \kappa'_{5,i-1}, u\mathsf{req}_{[req(j,d)]}v, \kappa'_{5,i+1}, \ldots, \kappa'_{5,n})\\
&\text{(because of B3 and A4; } m \in \{1,2,\ldots,n\} \text{ is used here as the index of the}\\
&\text{successor of } M_i)\\
\vdash^*_{\mathcal{M}'}\ &(p\text{¢}w\$, \kappa'_{6,1}, \ldots, \kappa'_{6,i-1}, u\mathsf{req}_{[req(j,d)]}v, \kappa'_{6,i+1},\\
&\qquad \ldots, \kappa'_{6,j-1}, u'\mathsf{res}_{[res(i,d',c)],[res(i,c)]}v', \kappa'_{6,j+1}, \ldots, \kappa'_{6,n})\\
&\text{(because of A6; } p \text{ is a state of M with index } \langle t_1,\ldots,t_{i-1},req(j),t_{i+1},\ldots,t_n\rangle)\\
\vdash^*_{\mathcal{M}'}\ &(\mathsf{req}^j_{\langle t_1,\ldots,t_{i-1},req(j),t_{i+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle}\text{¢}w\$, \kappa'_{7,1}, \ldots, \kappa'_{7,i-1}, u\mathsf{req}_{[req(j,d)]}v, \kappa'_{7,i+1},\\
&\qquad \ldots, \kappa'_{7,j-1}, u'\mathsf{res}_{[res(i,d',c)],[res(i,c)]}v', \kappa'_{7,j+1}, \ldots, \kappa'_{7,n})\\
\vdash_{\mathcal{M}'}\ &(\mathsf{rec}^j_{\langle t_1,\ldots,t_{i-1},req(j),t_{i+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle,[res(i,c)]}\text{¢}w\$, \kappa'_{8,1},\\
&\qquad \ldots, \kappa'_{8,i-1}, u\mathsf{req}_{[req(j,d)]}v, \kappa'_{8,i+1}, \ldots, \kappa'_{8,j-1}, u'\mathsf{ack}_{[res(i,d',c)],[res(i,c)]}v', \kappa'_{8,j+1}, \ldots, \kappa'_{8,n})\\
\vdash_{\mathcal{M}'}\ &(\mathsf{res}^j_{\langle t_1,\ldots,t_{i-1},req(j),t_{i+1},\ldots,t_{j-1},res(i,c),t_{j+1},\ldots,t_n\rangle,[req(i)]}\text{¢}w\$, \kappa'_{9,1},\\
&\qquad \ldots, \kappa'_{9,i-1}, u\mathsf{req}_{[req(j,d)]}v, \kappa'_{9,i+1}, \ldots, \kappa'_{9,j-1}, u'\mathsf{req}_{[res(i,d',c)]}v', \kappa'_{9,j+1}, \ldots, \kappa'_{9,n})\\
&\text{(because of B6 and A6)}\\
\vdash_{\mathcal{M}'}\ &(\mathsf{ack}^j_{\langle t_1,\ldots,t_{i-1},req(j),t_{i+1},\ldots,t_{j-1},res(i,c),t_{j+1},\ldots,t_n\rangle,[req(i)]}\text{¢}w\$, \kappa'_{10,1},\\
&\qquad \ldots, \kappa'_{10,i-1}, u\mathsf{req}_{[req(j,d)]}v, \kappa'_{10,i+1}, \ldots, \kappa'_{10,j-1}, u'\mathsf{rec}_{[res(i,d',c)],[req(i)]}v', \kappa'_{10,j+1}, \ldots, \kappa'_{10,n})\\
\vdash_{\mathcal{M}'}\ &(\mathsf{res}^i_{\langle t_1,\ldots,t_{i-1},*,t_{i+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle,[res(j,c)]}\text{¢}w\$, \kappa'_{11,1},\\
&\qquad \ldots, \kappa'_{11,i-1}, u\mathsf{req}_{[req(j,d)]}v, \kappa'_{11,i+1}, \ldots, \kappa'_{11,j-1}, K'_j, \kappa'_{11,j+1}, \ldots, \kappa'_{11,n})\\
&\text{(because of B7 and A7)}\\
\vdash_{\mathcal{M}'}\ &(\mathsf{ack}^i_{\langle t_1,\ldots,t_{i-1},*,t_{i+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle,[res(j,c)]}\text{¢}w\$, \kappa'_{12,1},\\
&\qquad \ldots, \kappa'_{12,i-1}, u\mathsf{rec}_{[req(j,d)],[res(j,c)]}v, \kappa'_{12,i+1}, \ldots, \kappa'_{12,n})\\
\vdash_{\mathcal{M}'}\ &(\mathsf{req}^{m'}_{\langle t_1,\ldots,t_{i-1},*,t_{i+1},\ldots,t_{j-1},*,t_{j+1},\ldots,t_n\rangle}\text{¢}w\$, \kappa'_{13,1}, \ldots, \kappa'_{13,i-1}, K'_i, \kappa'_{13,i+1}, \ldots, \kappa'_{13,n})\\
&\text{(because of B8 and A5; } m' \in \{1,2,\ldots,n\} \text{ is the index of the successor of } M_i),
\end{aligned}
$$

where for K'_i one of the following conditions hold (similarly for K'_j):

- $K_i = K'_i$ if K_i is a configuration uqv and q is not a communication state,
- $K_i = \mathsf{Accept}$, and K'_i contains the state $\mathsf{res}_{[Accept]}$,
- K_i contains a request state req^j_d, and K'_i contains the state $\mathsf{res}_{[req(j,d)],[req(j)]}$, or
- K_i contains a response state $\mathsf{res}^j_{d,c}$, and K'_i contains the state $\mathsf{res}_{[res(j,d,c)],[res(j,c)]}$.

In this simulation it was assumed that the request is sent before the corresponding response and that the master communicates with the requesting component first. Of course, the other way around can occur. Then A6 would be interchanged with A4, B6 with B3, B7 with B4, A7 with A5, and B8 with B5. Moreover, the order of reaching the request state from the master or the communication state from the component can be changed without having an effect on the result of the communication. Furthermore, if a component is in a communication state but gets never an answer from the communication partner or no corresponding answer, its index does not appear within the set of successors anymore and it is stuck like in the original system.

An important fact is that there exists exactly one situation where the master component and therefore the whole system gets stuck. This happens if (and only if) all components are stuck because either the transition is not defined in the original component or a communication cannot be completed. Due to the definition of the successor component, the master asks cyclically every component for its current situation until it sends a stuck or communication message. Both are stored within the situation tuple of the master. Because of A2, a component cannot get stuck without sending a corresponding message. In particular, the master does not get stuck because some components are in a communication deadlock (no corresponding messages are sent).

Now it can be concluded that for all $i \in \{1, 2, \dots, n\}$, all input words w, initial states $q_0^{(j)}$ of M_j ($j \in \{1, 2, \dots, n\}$), initial states $p_0^{(j)}$ of M'_j, configurations κ_j, κ'_j of M_j, M'_j ($j \in \{1, 2, \dots, n\} \setminus \{i\}$), and keeping the same conditions for K_i and K'_i mentioned above,

$$(q_0^{(1)}\text{¢}w\$, q_0^{(2)}\text{¢}w\$, ..., q_0^{(n)}\text{¢}w\$) \vdash^*_{\mathcal{M}} (\kappa_1, \kappa_2, ..., \kappa_{i-1}, K_i, \kappa_{i+1}, ..., \kappa_n)$$

if and only if

$$(\mathsf{req}^1_{\langle *,\dots,* \rangle}\text{¢}w\$, p_0^{(1)}\text{¢}w\$, p_0^{(2)}\text{¢}w\$, ..., p_0^{(n)}\text{¢}w\$) \vdash^*_{\mathcal{M}'} (\kappa, \kappa'_1, \kappa'_2, ..., \kappa'_{i-1}, K'_i, \kappa'_{i+1}, ..., \kappa'_n).$$

Finally, this particularly applies to $K_i = \mathsf{Accept}$ and $K'_i = u\mathsf{res}_{[Accept]}v$, and thus,

$$(q_0^{(1)}\text{¢}w\$, q_0^{(2)}\text{¢}w\$, ..., q_0^{(n)}\text{¢}w\$) \vdash^*_{\mathcal{M}} (\kappa_1, \kappa_2, ..., \kappa_{i-1}, \mathsf{Accept}, \kappa_{i+1}, ..., \kappa_n)$$

if and only if

$$\begin{aligned}
& (\mathsf{req}^1_{\langle *,\dots,* \rangle}\text{¢}w\$, p_0^{(1)}\text{¢}w\$, p_0^{(2)}\text{¢}w\$, ..., p_0^{(n)}\text{¢}w\$) \\
\vdash^*_{\mathcal{M}'} \quad & (\kappa, \kappa'_{1,1}, \kappa'_{1,2}, ..., \kappa'_{1,i-1}, u\mathsf{res}_{[Accept]}v, \kappa'_{1,i+1}, ..., \kappa'_{1,n}) \\
\vdash^*_{\mathcal{M}'} \quad & (\mathsf{req}^i_{\langle t_1,\dots,t_{i-1},*,t_{i+1},\dots,t_n \rangle}\text{¢}w\$, \kappa'_{2,1}, ..., \kappa'_{2,i-1}, u\mathsf{res}_{[Accept]}v, \kappa'_{2,i+1}, ..., \kappa'_{2,n}) \\
\vdash^*_{\mathcal{M}'} \quad & (\mathsf{rec}^i_{\langle t_1,\dots,t_{i-1},*,t_{i+1},\dots,t_n \rangle,[Accept]}\text{¢}w\$, \kappa'_{3,1}, ..., \kappa'_{3,i-1}, u\mathsf{ack}_{[Accept]}v, \kappa'_{3,i+1}, ..., \kappa'_{3,n}) \\
\vdash^*_{\mathcal{M}'} \quad & (\mathsf{Accept}, \kappa'_{4,1}, ..., \kappa'_{4,i-1}, u\mathsf{ack}_{[Accept]}v, \kappa'_{4,i+1}, ..., \kappa'_{4,n}).
\end{aligned}$$

This means that $\mathcal{M}$ accepts an input word if and only if $\mathcal{M}'$ accepts it, hence $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$. □

Theorem 4 is restricted to restarting automata without MVL operations. Now systems are considered in which MVL operations are allowed. The use of MVL operations together with that of MVR operations may cause infinite loops within the computation of a component. This happens when the read/write window is moved over the tape without changing the content of the tape, and after a finite number of moves, the same local configuration is reached again. Such a situation can be responsible for a deadlock of the constructed system considered in the proof of Theorem 4. If a component is in an infinite loop, and the master sends a request to that component (because the master wants to know the result of its local computation), the whole system gets stuck since the request will never be answered. Subsequently even if some other component sends the accepting message, the master is not able to receive this and hence does not accept. Here the input is not accepted by the centralized system, although it is accepted by the original system. Observe that we constructed the centralized system in such a way that only the master is allowed to accept. This is not really necessary here but quite helpful for our further work. But even if we allowed each component to accept the input, the described deadlock could lead to a situation where no more communication steps can be applied because of the master's inability to act. The next theorem gives a solution that avoids the described deadlock situation.

Theorem 5. *$\mathcal{L}$(cPC-X) = $\mathcal{L}$(PC-X) holds true for all $X \in \mathcal{T}$.*

Proof. Take the construction of the proof of Theorem 4. A simple modification is necessary to avoid a deadlock of the whole system caused by an infinite loop of a local computation. Every loop consists of at least one MVR and one MVL step. Whenever a component wants to perform a MVL operation, it sends the message $[*]$ to the master and stores the successor state within the index of the communication state:

A8. for all $(q', \mathsf{MVL}) \in \delta_i(q, \alpha)$,

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

When the master receives the information $[*]$, it just goes on by asking the successor component:

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

Consider that $t_m = *$ is not changed. Therefore, M'_m is asked again after the master has communicated with the other components. In between M'_m continues performing local computation steps. With this modification the master has still the possibility to communicate with all components (that have not already sent $[\perp]$ or wait in a communication situation), although one of them (or more) is in an infinite loop of computation. Therefore, $\mathcal{M}'$ accepts an input iff $\mathcal{M}$ accepts it. Similar to a single automaton with MVL operations, also a PCRA system with MVL operations does not have to halt for all inputs. This happens if at least one component is in an infinite loop and none of the (other) components reaches the accepting configuration. Such an input is not a member of the language accepted by the system. According to the construction above, this means that if a component M in $\mathcal{M}$ can reach an infinite loop, then the component M' in $\mathcal{M}'$ does so as well, and moreover, M' communicates with the master component every time it applies a MVL step. □

Remark. *In Theorem 5 it would not be enough to replace the RLWW-components by equivalent RRWW-components (which exist because of $\mathcal{L}(\mathsf{RRWW}) = \mathcal{L}(\mathsf{RLWW})$, see [Plá01]), because it is not clear whether this equivalence carries over to PCRA systems, e.g. whether, for every RLWW-component, there exists a communicational equivalent RRWW-component. Moreover, for deterministic components it can be assumed that equivalent components without loops can be constructed [Sip78, GMP07].*

Remark. *Theorems 4 and 5 also hold for systems with other types of restarting automata (R, RR, RW, RRW, RWW; RL, RLW respectively) as well as for systems with deterministic or monotone components, because the new component M is still a deterministic monotone R-automaton, and all modifications of the components have only an effect on the communication. Therefore, the construction preserves the type of the components and their properties of being determinisic and/or monotone.*

The centralized system considered in Theorem 4 and extended in Theorem 5 contains one additional component (the master component) in comparison to the non-centralized system. But it can be observed that the master component applies only communications but no local operations (no MVL, MVR, rewrite, Restart). In other words, the computation of the master component never depends on the content of the tape. Thus, the idea seems obvious to merge the master component with some other component and to build the product automaton in a broader sense. But then, the current state of the master gets lost with every restart of the other component. To avoid this, the new resulting component has the nonforgetting property[4]. The next lemma describes the merging process in detail. Thereafter, Lemma 3 shows, how the nonforgetting property of the first component can be eliminated.

Lemma 2. *Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a centralized PCRA system of degree $n > 2$ that accepts if and only if the first component accepts, and the first component only performs communication steps and* SCO *transitions. Moreover, the transitions of M_1 do not depend on the tape content, that is,*

$$A \in \delta_1(p, \alpha) \Rightarrow (A \in Q_1 \cup \{\mathsf{Accept}\} \wedge \forall \beta \in \mathcal{PC}_{M_1} : A \in \delta_1(p, \beta)).$$

In other words, no MVR*,* MVL*,* $\mathsf{Restart}$*, or rewrite operation is allowed for M_1. Then there exists a centralized system $\mathcal{M}'$ of degree $(n-1)$ with $L(\mathcal{M}) = L(\mathcal{M}')$, and the first component of $\mathcal{M}'$ is allowed to use the nonforgetting property*[5]*.*

Proof. Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a centralized system of degree n with $n > 2$ that accepts if and only if M_1 accept, and M_1 only performs communication steps and SCO transitions. Moreover, the transitions of M_1 do not depend on the tape content. Since M_1 works independently from its tape, $\Gamma_1 = \Gamma_2$ can be assumed. Now, we merge $M_1 = (Q_1, \Sigma, \Gamma_1, ¢, \$, q_0^{(1)}, k, \delta_1)$ and $M_2 = (Q_2, \Sigma, \Gamma_2, ¢, \$, q_0^{(2)}, k, \delta_2)$ into a new component $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, which simulates M_1 and M_2 almost simultaneously:

- $Q = (Q_1 \times Q_2) \cup \mathsf{COM}(M_1) \cup \{\mathsf{req}^i_{[d,q]} \mid \mathsf{req}^i_d \in Q_1 \wedge q \in Q_2\} \cup \{\mathsf{res}^i_{[d,q],c} \mid \mathsf{res}^i_{d,c} \in Q_1 \wedge q \in Q_2\} \cup \{\mathsf{rec}^i_{[d,q],c} \mid \mathsf{rec}^i_{d,c} \in Q_1 \wedge q \in Q_2\} \cup \{\mathsf{ack}^i_{[d,q],c} \mid \mathsf{ack}^i_{d,c} \in Q_1 \wedge q \in Q_2\}$,
- $\Gamma = \Gamma_2$,
- $q_0 = (q_0^{(1)}, q_0^{(2)})$.

Due to Corollary 2 it can be assumed that $q_0^{(1)}$ and $q_0^{(2)}$ are not communication states. The construction of the transition relation δ of the new automaton M distinguishes mainly between three cases: 1) M_2 performs a local computation (MVR, MVL, rewrite, $\mathsf{Restart}$, or SCO transition), 2) M_1 wants to communicate with M_2, and 3) M_1 wants to communicate with another component than M_2. If M_1 is not in a request or response state (for the current state p of M_1 it holds that $p \notin \mathsf{REQ}(M_1) \cup \mathsf{RES}(M_1)$), then M simulates one computation step of both, M_1 and M_2:

[4] For further information about the nonforgetting property, see Sections 4 and 5.4.

[5] The nonforgetting property was shortly explained in Section 4 and will be studied in more detail in Section 5.4.

$$\begin{aligned}\delta((p,q),\alpha) = &\{\mathit{Accept} \mid \mathit{Accept} \in \delta_1(p,\alpha)\}\\ &\cup\{(p',q') \mid p' \in \delta_1(p,\alpha), q' \in \delta_2(q,\alpha)\}\\ &\cup\{((p',q'),\mathsf{MVR}) \mid p' \in \delta_1(p,\alpha), (q',\mathsf{MVR}) \in \delta_2(q,\alpha)\}\\ &\cup\{((p',q'),\mathsf{MVL}) \mid p' \in \delta_1(p,\alpha), (q',\mathsf{MVL}) \in \delta_2(q,\alpha)\}\\ &\cup\{((p',q'),\beta) \mid p' \in \delta_1(p,\alpha), (q',\beta) \in \delta_2(q,\alpha)\}\\ &\cup\{(\mathsf{Restart},(p,q_0^{(2)})) \mid \mathsf{Restart} \in \delta_2(q,\alpha)\}\\ &\cup\{((p',q)) \mid p' \in \delta_1(p,\alpha), \delta_2(q,\alpha) = \emptyset\}.\end{aligned}$$

The first set lets M reach the accepting configuration if M_1 would do so (observe that because of our assumption M_2 cannot accept). The second to the fifth sets describe the changing of the states of M_1 and M_2 and in addition the tape operation of M_2 (M_1 never performs tape operations). If M_2 performs a restart, M performs a restart, too, but using the non-forgetting property so that the current state of M_1 is not lost. The last set deals with the situation that no more local computation step of M_2 is possible, because δ_2 is not defined for the current configuration of M_2 or q is a communication state. In the latter case M_2 waits for M_1 to reach the corresponding communication state. However, if δ_1 is not defined for some configuration, then M_1 is stuck and $\mathcal{M}$ cannot accept the input. Thus, it is not necessary to define δ for this situation (so M is stuck and $\mathcal{M}'$ does not accept as well). A communication between M_1 and M_2 can be treated by the following two cases (for all $\alpha \in \mathcal{PC}_M$):

$$\begin{aligned}\delta((\mathsf{req}_d^2,\mathsf{res}_{d',c}),\alpha) &= \{(\mathsf{rec}_{d,c}^2,\mathsf{ack}_{d',c})\},\\ \delta((\mathsf{res}_{d,c}^2,\mathsf{req}_{d'}),\alpha) &= \{(\mathsf{ack}_{d,c}^2,\mathsf{rec}_{d',c})\}.\end{aligned}$$

If M_1 wants to communicate with M_2 and the latter is still not in a communication state (the current state $q \notin \mathsf{COM}(M_2)$), M keeps the communication state of M_1 and simulates the computation of M_2 until M_2 reaches a corresponding communication state. As in the computation of $\mathcal{M}$ the simulation of M_1 is stopped (M_1 waits) until the communication can be completed.

$$\begin{aligned}\delta((\mathsf{req}_d^2,q),\alpha) = &\{((\mathsf{req}_d^2,q'),\mathsf{MVR}) \mid (q',\mathsf{MVR}) \in \delta_2(q,\alpha)\}\\ &\cup\{((\mathsf{req}_d^2,q'),\mathsf{MVL}) \mid (q',\mathsf{MVL}) \in \delta_2(q,\alpha)\}\\ &\cup\{((\mathsf{req}_d^2,q'),\beta) \mid (q',\beta) \in \delta_2(q,\alpha)\}\\ &\cup\{(\mathsf{req}_d^2,q') \mid q' \in \delta_2(q,\alpha)\}\\ &\cup\{(\mathsf{Restart},(\mathsf{req}_d^2,q_0^{(2)})) \mid \mathsf{Restart} \in \delta_2(q,\alpha)\},\\ \delta((\mathsf{res}_{d,c}^2,q),\alpha) = &\{((\mathsf{res}_{d,c}^2,q'),\mathsf{MVR}) \mid (q',\mathsf{MVR}) \in \delta_2(q,\alpha)\}\\ &\cup\{((\mathsf{res}_{d,c}^2,q'),\mathsf{MVL}) \mid (q',\mathsf{MVL}) \in \delta_2(q,\alpha)\}\\ &\cup\{((\mathsf{res}_{d,c}^2,q'),\beta) \mid (q',\beta) \in \delta_2(q,\alpha)\}\\ &\cup\{(\mathsf{res}_{d,c}^2,q') \mid q' \in \delta_2(q,\alpha)\}\\ &\cup\{(\mathsf{Restart},(\mathsf{res}_{d,c}^2,q_0^{(2)})) \mid \mathsf{Restart} \in \delta_2(q,\alpha)\}.\end{aligned}$$

In the case that M_2 never reaches a corresponding communication state the simulation of M_1 is stopped, and thus, M cannot reach the accepting configuration. This is the same behaviour as in the original system $\mathcal{M}$: M_1 is stuck while waiting for a corresponding answer from M_2 that is never sent, hence, $\mathcal{M}$ does not accept the input.

Besides the communication from M_1 with M_2 the former one can communicate with other components (but M_2 cannot, as $\mathcal{M}$ is centralized). For all $2 < i \leq n$, $\alpha \in \mathcal{PC}_M$, and $q \in Q_2$ define:

$$\begin{aligned}
\delta((\mathsf{req}^i_d, q), \alpha) &= \{\mathsf{req}^i_{[d,q]}\},\\
\delta(\mathsf{rec}^i_{[d,q],c}, \alpha) &= \{(p,q) \mid p \in Q_1, p \in \delta_1(\mathsf{rec}^i_{d,c}, \alpha)\},\\
\\
\delta((\mathsf{res}^i_{d,c}, q), \alpha) &= \{\mathsf{res}^i_{[d,q],c}\},\\
\delta(\mathsf{ack}^i_{[d,q],c}, \alpha) &= \{(p,q) \mid p \in Q_1, p \in \delta_1(\mathsf{ack}^i_{d,c}, \alpha)\}.
\end{aligned}$$

While M simulates a communication of M_1, it stores the current state of M_2 within the local index of the communication state. When continuing the computation after the communication from the corresponding receive or acknowledge state, M_2 can go on with the local computation (except in case that M_1 performs another communication with a component that is not M_2).

□

Observe that the constructed component M contains nonforgetting operations as well as SCO transitions in general. By Corollary 3 an equivalent system without SCO transitions exists. The next lemma deals with the nonforgetting property of the master component.

Lemma 3. *Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a centralized PCRA system of degree $n > 1$ such that M_1 is a nonforgetting component. Then there exists a system $\mathcal{M}' = (M_1', M_2', M_3, \ldots, M_n)$ of the same type but without a nonforgetting component such that $L(\mathcal{M}) = L(\mathcal{M}')$ holds.*

Proof. The basic idea for this proof is to modify M_1 and M_2 in such a way that the computation of M_2 is controlled by M_1. For this purpose, M_2 always asks M_1 with a request about how it should continue its computation. Now there are two situations when M_1 instructs M_2 to execute computation or communication steps. First, M_1 wants to make a restart and needs M_2 for storing the restart state. And second, to execute a communication step between both. For the latter situation it is important that M_2 can perform local operations to reach the corresponding communication state. M_1' and M_2' are constructed by modifying M_1 and M_2 in the following way.

The new initial states are req^2_R for M'_1 and $\mathsf{res}_{[q_0^{(2)}],[q_0^{(1)}]}$ for M'_2, where $q_0^{(1)}$ and $q_0^{(2)}$ are the original initial states of M_1 and M_2. This reflects the fact that M'_1 asks M'_2 for the restart state after a restart operation, and that M'_2 gives the information to M'_1 that the current state is $q_0^{(1)}$ and stores its own current state $q_0^{(2)}$ within the local index. Due to Corollary 2 it can be assumed that the initial states of M_1 and M_2 are not communication states. Together with the definition of

$$p \in \delta'_1(\mathsf{rec}^2_{R,[p]}, \alpha) \tag{1}$$

and

$$\mathsf{req}_{[q]} \in \delta'_2(\mathsf{ack}_{[q],[q_0^{(1)}]}, \beta) \tag{2}$$

for all $p \in Q_1$, $q \in Q_2$, $\alpha \in \mathcal{PC}_{M_1}$, and $\beta \in \mathcal{PC}_{M_2}$ every computation of $\mathcal{M}'$ starts with

$$\begin{aligned} & (\mathsf{req}^2_R ¢ w\$, \mathsf{res}_{[q_0^{(2)}],[q_0^{(1)}]} ¢ w\$, q_0^{(3)} ¢ w\$, \ldots, q_0^{(n)} ¢ w\$) \\ \vdash_{\mathcal{M}'} \; & (\mathsf{rec}^2_{R,[q_0^{(1)}]} ¢ w\$, \mathsf{ack}_{[q_0^{(2)}],[q_0^{(1)}]} ¢ w\$, \ldots) \\ \vdash_{\mathcal{M}'} \; & (q_0^{(1)} ¢ w\$, \mathsf{req}_{[q_0^{(2)}]} ¢ w\$, \ldots). \end{aligned}$$

Since $\mathcal{M}$ is centralized, the components $M_3, \ldots, M_n$ work independently from M_2, and thus, they behave exactly in the same way as in the system $\mathcal{M}$. In addition, all communication steps between M_1 and one of the components $M_3, \ldots, M_n$ are realized in the system $\mathcal{M}'$ as in $\mathcal{M}$, because the behaviour of M'_1 differs only from M_1 in the two situations described above (restart, communication with M_2). Thus, the configurations of M_3, …, M_n are disregarded in the next considerations. As long as M'_1 does not perform a restart operation or wants to communicate with M'_2, the second component M'_2 remains within a request state the index of which contains the last state that was reached by its local computation.

When M'_1 wants to make a restart, it sends the restart state to M'_2, performs the restart, and obtains the restart state from M'_2. Therefore, we replace every transition of the form $(\mathsf{Restart}, r) \in \delta_1(p, \alpha)$ as follows:

$$\mathsf{res}^2_{R,[r]} \in \delta'_1(p, \alpha) \tag{3}$$

$$\mathsf{Restart} \in \delta'_1(\mathsf{ack}^2_{R,[r]}, \alpha) \tag{4}$$

$$\mathsf{res}_{[q],[r]} \in \delta'_2(\mathsf{rec}_{[q],[r]}, \beta) \text{ for all } q \in Q_2 \text{ and } \beta \in \mathcal{PC}_{M_2} \tag{5}$$

$$\mathsf{req}_{[q]} \in \delta'_2(\mathsf{ack}_{[q],[r]}, \beta) \text{ for all } q \in Q_2 \text{ and } \beta \in \mathcal{PC}_{M_2} \tag{6}$$

Assume that M'_1 and M'_2 are in the configurations upv and $x\mathsf{req}_{[q]}y$, and M'_1 wants to perform a restart (e.g. $(\mathsf{Restart}, r) \in \delta_1(p, \pi_{M_1}(v))$. Then $\mathcal{M}'$ executes the following computation with the result that M'_1 reaches the restart configuration with the given restart state r and M_2' reaches the same configuration as before the restart of M_1':

$$\begin{aligned}
& (upv, x\mathsf{req}_{[q]}y, \dots) && (7)\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{res}^2_{R,[r]}v, x\mathsf{req}_{[q]}y, \dots) && \text{(because of 3)}\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{ack}^2_{R,[r]}v, x\mathsf{rec}_{[q],[r]}y, \dots) && \text{(comm. step)}\\
\vdash_{\mathcal{M}'} \; & (\mathsf{req}^2_R uv, x\mathsf{res}_{[q],[r]}y, \dots) && \text{(because of 4 and 5)}\\
\vdash_{\mathcal{M}'} \; & (\mathsf{rec}^2_{R,[r]}uv, x\mathsf{ack}_{[q],[r]}y, \dots) && \text{(comm. step)}\\
\vdash_{\mathcal{M}'} \; & (ruv, x\mathsf{req}_{[q]}y, \dots) && \text{(because of 1 and 6)}
\end{aligned}$$

To simulate communications between M_1 and M_2 in $\mathcal{M}$, it is necessary to let M_2' perform the local computation that M_2 does before the communication in the system $\mathcal{M}$. To start the local computation of M_2' and arrange the communication, the transition

$$q \in \delta_2'(\mathsf{rec}_{[q],L}, \beta) \tag{8}$$

is added for all $q \in Q_2$ and $\beta \in \mathcal{PC}_{M_2}$, where '$L$' stands for 'local computation' (see below). When M_2' receives the information 'L' from M_1' it changes into the state q via an SCO transition, where q is the last state reached by the local computation of M_2' and that is stored in the local index of the communication state. All transitions of the form $\mathsf{req}^2_d \in \delta_1(p, \alpha)$ and $\mathsf{res}_{d',c} \in \delta_2(q, \beta)$ are replaced as follows:

$$\begin{aligned}
\mathsf{res}^2_{req(d),L} &\in \delta_1'(p, \alpha) && (9)\\
\mathsf{req}^2_{req(d)} &\in \delta_1'(\mathsf{ack}^2_{req(d),L}, \alpha) && (10)\\
\mathsf{rec}^2_{d,c} &\in \delta_1'(\mathsf{rec}^2_{req(d),res(c)}, \alpha) && (11)\\
\mathsf{res}^2_{req(d),L} &\in \delta_1'(\mathsf{rec}^2_{req(d),[q_0^{(1)}]}, \alpha), && (12)\\
\mathsf{res}_{[ack_{d',c}],res(c)} &\in \delta_2'(q, \beta), && (13)\\
\mathsf{req}_{[ack_{d',c}]} &\in \delta_2'(\mathsf{ack}_{[ack_{d',c}],res(c)}, \beta). && (14)
\end{aligned}$$

With the lines (9), (10), and (11) M_1' asks M_2' for continuing the local computation, awaits the answer from M_2', and changes in the original receive state if M_2' sends a corresponding answer. The original communication state req_d of M_1 is stored in the local index of the used communication states. Line (12) is needed, when M_2' performs a restart during the local computation, thus when it is set into its initial state $\mathsf{res}_{[q_0^{(2)}],[q_0^{(1)}]}$ and sends M_1' the initial state $q_0^{(1)}$. M_1' can detect this situation, since the received information is obviously not an expected communication answer.

Thus, M_1' asks M_2' again to continue the local computation and awaits the answer of the initiated communication. After the restart, M_2' goes on with the local computation from the restart configuration (using line (8)). When M_2' reaches a communication state, it sends this through a response to M_1' (line (13)), stores the acknowledge state corresponding to the currently performed communication in the local index, and again awaits new instructions from M_1' by waiting in a request state.

In the simulated situation above M_1 reaches a request state and M_2 reaches the corresponding response state. The following modifications describe the other way around. Hence, replace $\mathsf{res}^2_{d,c} \in \delta_1(p, \alpha)$ and $\mathsf{req}_{d'} \in \delta_2(q, \beta)$ with:

$$\begin{align}
\mathsf{res}^2_{res(d,c),L} &\in \delta_1'(p, \alpha), \tag{15}\\
\mathsf{req}^2_{res(d,c)} &\in \delta_1'(\mathsf{ack}^2_{res(d,c),L}, \alpha), \tag{16}\\
\mathsf{res}^2_{d,c} &\in \delta_1'(\mathsf{rec}^2_{res(d,c),req}, \alpha), \tag{17}\\
\mathsf{res}^2_{res(d,c),L} &\in \delta_1'(\mathsf{rec}^2_{res(d,c),[q_0^{(1)}]}, \alpha), \tag{18}\\
\mathsf{res}_{rec(d'),req} &\in \delta_2'(q, \beta), \tag{19}\\
\mathsf{req}_{rec(d')} &\in \delta_2'(\mathsf{ack}_{rec(d'),req}, \beta), \tag{20}\\
\mathsf{req}_{[rec_{d',c}]} &\in \delta_2'(\mathsf{rec}_{rec(d'),c}, \beta). \tag{21}
\end{align}$$

One additional aspect in comparison to the former direction of the communication (lines (9) to (14)) is the fact that in this case (lines (15) to (21)) M_1' has to send the information c to M_2'. This is done through the original response state $\mathsf{res}^2_{d,c}$ (line (17)) and the additional request of line (20).

As mentioned above, within the local computation of M_2' a restart is possible (M_2' is set into its restart state), while M_1' is waiting for a communication answer (see lines (10) and (16), respectively). This leads to one of the following two computations, depending on the original communication state of M_1:

$$\begin{align}
& (u\mathsf{req}^2_{req(d)}v, \mathsf{res}_{[q_0^{(2)}],[q_0^{(1)}]}xy, \dots) & \tag{22}\\
\vdash_{\mathcal{M}'}\ & (u\mathsf{rec}^2_{req(d),[q_0^{(1)}]}v, \mathsf{ack}_{[q_0^{(2)}],[q_0^{(1)}]}xy, \dots) & \text{(comm. step)}\\
\vdash_{\mathcal{M}'}\ & (u\mathsf{res}^2_{req(d),L}v, \mathsf{req}_{[q_0^{(2)}]}xy, \dots) & \text{(because of 12 and 2)}\\
\vdash_{\mathcal{M}'}\ & (u\mathsf{ack}^2_{req(d),L}v, \mathsf{rec}_{[q_0^{(2)}],L}xy, \dots) & \text{(comm. step)}\\
\vdash_{\mathcal{M}'}\ & (u\mathsf{req}^2_{req(d)}v, q_0^{(2)}xy, \dots) & \text{(because of 10 and 8)}
\end{align}$$

$$\begin{aligned}
& (u\mathsf{req}^2_{res(d,c)}v, \mathsf{res}_{[q_0^{(2)}],[q_0^{(1)}]}xy, \dots) && (23)\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{rec}^2_{res(d,c),[q_0^{(1)}]}v, \mathsf{ack}_{[q_0^{(2)}],[q_0^{(1)}]}xy, \dots) && \text{(comm. step)}\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{res}^2_{res(d,c),L}v, \mathsf{req}_{[q_0^{(2)}]}xy, \dots) && \text{(because of 18 and 2)}\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{ack}^2_{res(d,c),L}v, \mathsf{rec}_{[q_0^{(2)}],L}xy, \dots) && \text{(comm. step)}\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{req}^2_{res(d,c)}v, q_0^{(2)}xy, \dots) && \text{(because of 16 and 8)}
\end{aligned}$$

Consider a communication between M_1 and M_2 within a computation of $\mathcal{M}$:

$$(u\mathsf{req}^2_d v, x\mathsf{res}_{d',c}y, \dots) \vdash_{\mathcal{M}} (u\mathsf{rec}^2_{d,c}v, x\mathsf{ack}_{d',c}y, \dots).$$

To reach this communication step there have to exist states $p \in Q_1$, $q \in Q_2$ and transitions $\mathsf{req}^2_d \in \delta_1(p, \pi_{M_1}(v))$, $\mathsf{res}_{d',c} \in \delta_2(q, \pi_{M_2}(y))$. Thus, $\mathcal{M}'$ performs the following computation:

$$\begin{aligned}
& (upv, x'\mathsf{req}_{[q']}y', \dots) && (24)\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{res}^2_{req(d),L}v, x'\mathsf{req}_{[q']}y', \dots) && \text{(because of 9)}\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{ack}^2_{req(d),L}v, x'\mathsf{rec}_{[q'],L}y', \dots) && \text{(comm. step)}\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{req}^2_{req(d)}v, x'q'y', \dots) && \text{(because of 10 and 8)}\\
\vdash^*_{\mathcal{M}'} \; & (u\mathsf{req}^2_{req(d)}v, xqy, \dots) && \text{(local comp. of } M_2'\text{)}\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{req}^2_{req(d)}v, x\mathsf{res}_{[\mathsf{ack}_{d',c}],res(c)}y, \dots) && \text{(because of 13)}\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{rec}^2_{req(d),res(c)}v, x\mathsf{ack}_{[\mathsf{ack}_{d',c}],res(c)}y, \dots) && \text{(comm. step)}\\
\vdash_{\mathcal{M}'} \; & (u\mathsf{rec}^2_{d,c}v, x\mathsf{req}_{[\mathsf{ack}_{d',c}]}y, \dots) && \text{(because of 11 and 14)}
\end{aligned}$$

The communication in the opposite direction of the form

$$(u\mathsf{res}^2_{d,c}v, x\mathsf{req}_{d'}y, \dots) \vdash_{\mathcal{M}} (u\mathsf{ack}^2_{d,c}v, x\mathsf{rec}_{d',c}y, \dots)$$

is simulated in $\mathcal{M}'$ in the following way:

$$\begin{array}{lll}
 & (upv, x'\mathsf{req}_{[q']}y', \dots) & (25)\\
\vdash_{\mathcal{M}'} & (u\mathsf{res}^2_{res(d,c),L}v, x'\mathsf{req}_{[q']}y', \dots) & \text{(because of 15)}\\
\vdash_{\mathcal{M}'} & (u\mathsf{ack}^2_{res(d,c),L}v, x'\mathsf{rec}_{[q'],L}y', \dots) & \text{(comm. step)}\\
\vdash_{\mathcal{M}'} & (u\mathsf{req}^2_{res(d,c)}v, x'q'y', \dots) & \text{(because of 16 and 8)}\\
\vdash^*_{\mathcal{M}'} & (u\mathsf{req}^2_{res(d,c)}v, xqy, \dots) & \text{(local comp. of } M_2')\\
\vdash_{\mathcal{M}'} & (u\mathsf{req}^2_{res(d,c)}v, x\mathsf{res}_{rec(d'),req}y, \dots) & \text{(because of 19)}\\
\vdash_{\mathcal{M}'} & (u\mathsf{rec}^2_{res(d,c),req}v, x\mathsf{ack}_{rec(d'),req}y, \dots) & \text{(comm. step)}\\
\vdash_{\mathcal{M}'} & (u\mathsf{res}^2_{d,c}v, x\mathsf{req}_{rec(d')}y, \dots) & \text{(because of 17 and 20)}\\
\vdash_{\mathcal{M}'} & (u\mathsf{ack}^2_{d,c}v, x\mathsf{rec}_{rec(d'),c}y, \dots) & \text{(comm. step)}\\
\vdash_{\mathcal{M}'} & (u'p'v', x\mathsf{req}_{[\mathsf{rec}_{d',c}]}y, \dots) & \text{(because of 21)}
\end{array}$$

Now $L(\mathcal{M}) = L(\mathcal{M}')$ can be shown by induction on the length of the computation. $L(\mathcal{M}) = L(\mathcal{M}')$ holds iff $\kappa \vdash^*_{M_1,\mathcal{M}} \mathsf{Accept} \Leftrightarrow \kappa \vdash^*_{M_1',\mathcal{M}'} \mathsf{Accept}$ for a configuration κ that can be reached from the initial configuration. For zero computation steps $\kappa = \mathsf{Accept}$, and the equivalence holds. For exactly one computation step $\kappa = uqv$ for a state q and a tape content uv, it is

$$\begin{aligned}
uqv \vdash_{M_1,\mathcal{M}} \mathsf{Accept} &\Leftrightarrow \mathsf{Accept} \in \delta_1(q, \pi_{M_1}(v))\\
&\Leftrightarrow \mathsf{Accept} \in \delta_1'(q, \pi_{M_1'}(v))\\
&\Leftrightarrow uqv \vdash_{M_1',\mathcal{M}'} \mathsf{Accept}.
\end{aligned}$$

For more than one computation step, we have the following:

$$\begin{array}{lclll}
uqv \vdash^*_{M_1,\mathcal{M}} \mathsf{Accept} & \Leftrightarrow & \exists\text{configuration } u'q'v' : & uqv \vdash_{M_1,\mathcal{M}} u'q'v' & \\
 & & & \quad \vdash^*_{M_1,\mathcal{M}} \mathsf{Accept} & \\
 & \Leftrightarrow & \exists\text{configuration } u'q'v' : & uqv \vdash^*_{M_1',\mathcal{M}'} u'q'v' & (*)\\
 & & & \quad \vdash^*_{M_1',\mathcal{M}'} \mathsf{Accept} & (**)
\end{array}$$

Line (*) holds for a MVR, MVL, or rewrite operation, since M_1 and M_1' behave in the same way for these cases. The same argument holds for a communication with any component except M_2. For a restart operation the validity was shown in (7). For a communication with M_2 the validity was shown in (24) and (25). Line (**) follows from the induction hypothesis. Since uqv is an arbitrary configuration, this part of the proof also holds for κ as the inital configuration of M_1. Hence, $L(\mathcal{M}) = L(\mathcal{M}')$ can be concluded. □

Observe that independently of whether the original system $\mathcal{M}$ contains SCO transitions or not, new SCO transitions can result within the proof of Lemma 3 through lines (1) and (8). Due to Corollary 3 there exists an equivalent system without SCO transitions.

With the previous preparations we can now conclude the most important result of this section, namely that centralization is not a restriction for PCRA systems, that is, centralized and non-centralized PCRA systems have the same computational power. Formally, we state this in the following corollaries.

Corollary 4. *For every PC-X-system $\mathcal{M}$ ($\mathsf{X} \in \mathcal{T}$), there exists a cPC-X-system $\mathcal{M}'$ with $L(\mathcal{M}) = L(\mathcal{M}')$ such that $\mathcal{M}'$ has the same number of components as $\mathcal{M}$.*

Proof. Systems of degree one are per definition centralized. For every non-centralized PCRA system of a higher degree, there exists an equivalent centralized system with an additional master component due to Theorems 4 and 5. According to Lemma 2 this additional component can be merged with the second component of the system, hence the new system has the same degree as the original non-centralized system, but the nonforgetting property is necessary (in general) for the first component of the constructed system. Using Lemma 3, we can avoid the use of this property. □

Further, the system constructed within the proofs of Theorem 4, Theorem 5, Lemma 2, and Lemma 3 accepts the input if and only if the first component reaches the accepting configuration. Thus, we can conclude the following:

Corollary 5. *For every PCRA system $\mathcal{M}$, there exists a centralized system $\mathcal{M}'$ of the same type and the same degree with $L(\mathcal{M}) = L(\mathcal{M}')$ such that $\mathcal{M}'$ accepts an input if and only if the first component of $\mathcal{M}'$ accepts.*

Moreover, the construction of the centralized system is deterministic, hence the following two corollaries can be concluded.

Corollary 6. *Theorem 4, Theorem 5, Lemma 2, and Lemma 3 also hold for locally deterministic systems. Particularly, for every det-local-PC-X-system $\mathcal{M}$ ($\mathsf{X} \in \mathcal{T}$), a det-local-cPC-X-system $\mathcal{M}'$ exists with $L(\mathcal{M}) = L(\mathcal{M}')$ such that $\mathcal{M}'$ has the same number of components as $\mathcal{M}$.*

Corollary 7. $\mathcal{L}(\mathsf{det}\text{-}\mathsf{global}\text{-}\mathsf{PC}\text{-}\mathsf{X}) = \mathcal{L}(\mathsf{det}\text{-}\mathsf{local}\text{-}\mathsf{PC}\text{-}\mathsf{X})$ *for all* $\mathsf{X} \in \mathcal{T}$.

Proof. Every globally deterministic system is a locally deterministic system as well. In addition, from Corollary 6 we see that, for every locally deterministic system $\mathcal{M}$, there exists a centralized locally deterministic system $\mathcal{M}'$ with $L(\mathcal{M}) = L(\mathcal{M}')$. Moreover, the constructed system (see the proofs of Theorem 4, Theorem 5, Lemma 2, and Lemma 3) is globally deterministic, since the system accepts any input if and only if the first component of the system accepts. Thus, for every locally deterministic PCRA system, there exists an equivalent (centralized) globally deterministic system. □

Due to Corollary 7 it is not longer necessary to distinguish between locally determinstic and globally deterministic PCRA systems. Therefore, in what follows the term *determinstic* will be used for systems with deterministic components - independent of which component is allowed to accept. Deterministic systems are marked with the prefix 'det-'.

Since all constructions used for centralization do not influence the rewrite steps, the centralization results can be carried over to systems with monotone components, and instead of distinguishing between the two types det-local-mon-PC-X and det-global-mon-PC-X, we just use det-mon-PC-X in further considerations.

Using the constructions presented in this section, we can make another useful observation concerning communication deadlocks in centralized systems. Although a component of a (non-centralized) system is stuck due to an unanswered communication, this does not give a disturbance of the communication or computation of the constructed centralized system. Mainly two basic facts are responsible for this: 1) The components are somehow 'communicational complete'. This means that, whenever the local computation of a component cannot be continued because of a nondefined transition or because it is waiting for an answer for a communication, the component sends an according message to the master. 2) The master component is programmed in such a way that it stores the situations of all components, and thus, it is able to handle all communication situations.

We summarize this section by the following corollary.

Corollary 8. *For every PCRA system $\mathcal{M}$, there can be effectively constructed a centralized system $\mathcal{M}'$ with the following properties:*

1. *$\mathcal{M}'$ is of the same type as $\mathcal{M}$,*
2. *$\mathcal{M}'$ has the same number of components as $\mathcal{M}$,*
3. *$\mathcal{M}'$ accepts the input if and only if its first component accepts, and*
4. *$\mathcal{M}'$ cannot reach a communication deadlock.*

5.4 Systems of nonforgetting restarting automata

The computation of a restarting automaton consists of different types of transition steps. One type is the restart operation that puts the finite control into the dedicated initial state and repositions the read/write window on the left end of the working tape. Therefore, the information about the current state and the window's position of the last cycle is lost ('forgotten'). At first view, it seems possible to store the current state and window position by writing it on the tape, but this method is quite restricted according to the type of the automaton (auxiliary symbols are needed), the size of the read/write window (it has to be larger than one), the length-reducing rewriting, and whether the window is allowed to move left in the deterministic case (this is required if the information about the state and window position is needed before it can be read from the tape).

A nonforgetting restarting automaton contains an additional set of restarting states (that is a subset of all states), and a restart operation now allows the automaton to change to a determined restart state (instead of the initial state) - depending on the current configuration. If all states of the automaton are allowed to be used as restart states, then the set of restart states can be omitted. This type of automata was introduced in [MS04] and investigated in [JO07, Mes07, Mes08, MO06, MO11].

It depends on the type of automaton whether the nonforgetting property yields an increase in the expressive power or not. This also depends on other properties like monotonicity, determinism, and shrinking[6]. For deterministic restarting automata Hartmut Messerschmidt shows in [Mes07] by using the copy language that for all automata of type $\mathsf{R(R)(W)(W)}$, the nonforgetting property gives more expressive power. For deterministic two-way restarting automata ($\mathsf{RL(W)(W)}$) it is open whether the nonforgetting property increases the expressive power.

Theorem 6. [Mes07] *$\mathcal{L}(\textsf{det-}X) \subset \mathcal{L}(\textsf{det-nf-}X)$ holds for all $X \in \mathcal{T}_R$.*

The following results show that in the case of monotone restarting automata, the usage of auxiliary symbols compensates for the nonforgetting property (see [MO06, Ott06]). One-way monotone nonforgetting restarting automata without auxiliary symbols that restart immediately after a rewriting are more powerful than those without the nonforgetting property (see [MO06]).

[6] In contrast to usual restarting automata, the rewrite operation of a shrinking restarting automaton does not necessarily have to shorten the length of the tape content. Instead, there exists a weight function that associates a weight to the tape content, and in each rewrite step this weight must decrease.

Theorem 7. [MO06, Ott06]

$$\begin{array}{lll}
\mathcal{L}(\textsf{\textit{mon-R}}) & \subset \mathcal{L}(\textsf{\textit{mon-nf-R}}) & \\
\mathcal{L}(\textsf{\textit{mon-RW}}) & \subset \mathcal{L}(\textsf{\textit{mon-nf-RW}}) & \\
\mathcal{L}(\textsf{\textit{mon-RWW}}) & = \mathcal{L}(\textsf{\textit{mon-nf-RWW}}) & (= \textsf{\textit{CFL}}) \\
\mathcal{L}(\textsf{\textit{mon-RRWW}}) & = \mathcal{L}(\textsf{\textit{mon-nf-RRWW}}) & (= \textsf{\textit{CFL}}) \\
\mathcal{L}(\textsf{\textit{mon-RLWW}}) & = \mathcal{L}(\textsf{\textit{mon-nf-RLWW}}) & (= \textsf{\textit{CFL}})
\end{array}$$

Results for restarting automata that are deterministic and monotone are summarized in the following theorem.

Theorem 8. [JMPV95, Mes07, MO06, MO11]

$$\begin{array}{lll}
(\textsf{\textit{DCFL}} =) & \mathcal{L}(\textsf{\textit{det-mon-R}}) & = \mathcal{L}(\textsf{\textit{det-mon-nf-R}}) \\
(\textsf{\textit{DCFL}} =) & \mathcal{L}(\textsf{\textit{det-mon-RW}}) & = \mathcal{L}(\textsf{\textit{det-mon-nf-RW}}) \\
(\textsf{\textit{DCFL}} =) & \mathcal{L}(\textsf{\textit{det-mon-RWW}}) & = \mathcal{L}(\textsf{\textit{det-mon-nf-RWW}}) \\
(\textsf{\textit{DCFL}} =) & \mathcal{L}(\textsf{\textit{det-mon-RR}}) & \subset \mathcal{L}(\textsf{\textit{det-mon-nf-RR}}) \\
(\textsf{\textit{DCFL}} =) & \mathcal{L}(\textsf{\textit{det-mon-RRW}}) & \subset \mathcal{L}(\textsf{\textit{det-mon-nf-RRW}}) \\
(\textsf{\textit{DCFL}} =) & \mathcal{L}(\textsf{\textit{det-mon-RRWW}}) & \subset \mathcal{L}(\textsf{\textit{det-mon-nf-RRWW}}) \\
(\textsf{\textit{LRR}} =) & \mathcal{L}(\textsf{\textit{det-mon-RL}}) & = \mathcal{L}(\textsf{\textit{det-mon-nf-RL}}) \\
(\textsf{\textit{LRR}} =) & \mathcal{L}(\textsf{\textit{det-mon-RLW}}) & = \mathcal{L}(\textsf{\textit{det-mon-nf-RLW}}) \\
(\textsf{\textit{LRR}} =) & \mathcal{L}(\textsf{\textit{det-mon-RLWW}}) & = \mathcal{L}(\textsf{\textit{det-mon-nf-RLWW}})
\end{array}$$

For shrinking nonforgetting restarting automata the equivalence to the forgetting variants was shown for sh-RRWW, sh-RWW, and det-sh-RLWW automata in [JO07, Mes07, MO11].

Theorem 9. [JO07, Mes07, MO11]

$$\begin{array}{ll}
\mathcal{L}(\textsf{\textit{det-sh-RLWW}}) & = \mathcal{L}(\textsf{\textit{det-nf-sh-RLWW}}) \\
\mathcal{L}(\textsf{\textit{sh-RRWW}}) & = \mathcal{L}(\textsf{\textit{nf-sh-RRWW}}) \\
\mathcal{L}(\textsf{\textit{sh-RWW}}) & = \mathcal{L}(\textsf{\textit{nf-sh-RWW}})
\end{array}$$

For general nondeterministic restarting automata there are no explicit results concerning the difference between forgetting and nonforgetting automata. Nevertheless the language

$$L = \{a^n b^n \mid n \geq 0\} \cup \{a^n b^m \mid m > 2n \geq 0\}$$

is an easy example to show that also in the (general) nondeterministic case, the expressive power can increase due to the nonforgetting property. This language cannot be accepted by any restarting automaton without auxiliary symbols (see

[Ott06]), but it can be accepted by the nf-R-automaton $M := (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b\}, ¢, \$, q_0, 5, \delta, \{q_0, q_1, q_2\})$ described by the following meta-instructions:

$$\begin{array}{ll}
(q_0, ¢\$, \mathsf{Accept}), & \\
(q_0, ¢ab\$, \mathsf{Accept}), & (q_0, ¢abb \cdot b^+ \cdot \$, \mathsf{Accept}), \\
(q_0, ¢a^*, aabb \to ab, q_1), & (q_0, ¢a^*, aabbb \to ab, q_2), \\
(q_1, ¢a^*, aabb \to ab, q_1), & (q_2, ¢a^*, aabbb \to ab, q_2), \\
(q_1, ¢ab\$, \mathsf{Accept}), & (q_2, ¢abb \cdot b^+ \cdot \$, \mathsf{Accept}).
\end{array}$$

Corollary 9. *$\mathcal{L}(X) \subset \mathcal{L}(\mathit{nf}\text{-}X)$ holds for all $X \in \{R, RW, RR, RRW, RL, RLW\}$.*

Now we will prove that a nonforgetting restarting automaton of an arbitrary type can be simulated by a PCRA system with two forgetting components of the same type. Trivially, a nonforgetting automaton cannot be simulated in general by a PCRA system of only one forgetting component of the same type. This holds particularly for types X with $\mathcal{L}(\mathsf{X}) \subset \mathcal{L}(\mathsf{nf\text{-}X})$.

Theorem 10. *For every type $X \in \mathcal{T}$, $\mathcal{L}(\mathit{nf}\text{-}X) \subseteq \mathcal{L}(\mathit{PC}\text{-}X(2))$.*

Proof. The proof idea is to construct, for a given nonforgetting automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta, Q_R)$, an equivalent system $\mathcal{M} = (M_1, M_2)$, where the first component works mainly like M. Only when M performs a restart, M_1 sends the corresponding restart state to the second component M_2, makes the restart (without restart state, but changing into the initial state), then requests the restart state from M_2, and changes into the received restart state immediately. M_1 and M_2 are defined formally as follows:

$$\begin{aligned}
M_1 &= (Q_1, \Sigma, \Gamma, ¢, \$, \mathsf{req}, k, \delta_1), \\
M_2 &= (Q_2, \Sigma, \Sigma, ¢, \$, \mathsf{res}_{q_0}, 1, \delta_2) \text{ with}
\end{aligned}$$

(1) $Q_1 = Q \cup \{\mathsf{req}\} \cup \{\mathsf{res}_q, \mathsf{rec}_q, \mathsf{ack}_q \mid q \in Q_R\}$,

(2) $Q_2 = \{\mathsf{req}\} \cup \{\mathsf{res}_q, \mathsf{rec}_q, \mathsf{ack}_q \mid q \in Q_R\}$,

(3) for all $p, q \in Q$ and $\alpha, \beta \in \mathcal{PC}^{(k)}$ define:

$$\begin{array}{lcl}
(p, \beta) \in \delta_1(q, \alpha) & \Leftrightarrow & (p, \beta) \in \delta(q, \alpha), \\
(p, \mathsf{MVR}) \in \delta_1(q, \alpha) & \Leftrightarrow & (p, \mathsf{MVR}) \in \delta(q, \alpha), \\
(p, \mathsf{MVL}) \in \delta_1(q, \alpha) & \Leftrightarrow & (p, \mathsf{MVL}) \in \delta(q, \alpha), \\
\mathsf{Accept} \in \delta_1(q, \alpha) & \Leftrightarrow & \mathsf{Accept} \in \delta(q, \alpha),
\end{array}$$

(4) for all transitions $(\mathsf{Restart}, q) \in \delta(p, \alpha)$ define:

$$\mathsf{res}_q \in \delta_1(p, \alpha),$$
$$\delta_1(\mathsf{ack}_q, \alpha) = \{\mathsf{Restart}\},$$

(5) for all $q \in Q_R$ and $\alpha \in \{¢\} \cdot (\Gamma^{k-1} \cup (\Gamma^{\leq k-2} \cdot \{\$\}))$ define:

$$\delta_1(\mathsf{rec}_q, \alpha) = \{q\},$$

(6) for all $q \in Q_R$ define:

$$\delta_2(\mathsf{rec}_q, ¢) = \{\mathsf{res}_q\},$$
$$\delta_2(\mathsf{ack}_q, ¢) = \{\mathsf{req}\}.$$

It remains to show that $L(M) = L(\mathcal{M})$. Therefore, proceeding by induction on the length of the computation of M, we prove that for all configurations κ not containing a communication state, the following holds:

$$\forall w \in \Sigma^* : q_0 ¢ w \$ \vdash_M^* \kappa \quad \Leftrightarrow \quad \mathsf{req} ¢ w \$ \vdash_{M_1,\mathcal{M}}^* \kappa.$$

Induction basis (one computation step): Let $w \in \Sigma^*$ be an arbitrary input word. Then:

$q_0 ¢ w \$ \vdash_M \kappa$

$\Leftrightarrow$ 1) $\kappa = \mathsf{Accept}$ and $\mathsf{Accept} \in \delta(q_0, \pi_k(¢ w \$))$, or
2) $\kappa = ¢ q w \$$ for a $q \in Q$ (MVR step) and $(q, \mathsf{MVR}) \in \delta(q_0, \pi_k(¢ w \$))$, or
3) $\kappa = ¢ u q v \$$ for a $q \in Q$ (rewrite step) and $(q, ¢ u) \in \delta(q_0, \pi_k(¢ w \$))$
(A MVL step as well as a restart are not possible in the first computation step, as the read/write window stands on the left border of the tape, and there was no rewrite operation before, respectively.)

$\Leftrightarrow$ 1) $\kappa = \mathsf{Accept}$ and $\mathsf{Accept} \in \delta_1(q_0, \pi_k(¢ w \$))$, or
2) $\kappa = ¢ q w \$$ for a $q \in Q$ and $(q, \mathsf{MVR}) \in \delta_1(q_0, \pi_k(¢ w \$))$, or
3) $\kappa = ¢ u q v \$$ for a $q \in Q$ and $(q, ¢ u) \in \delta_1(q_0, \pi_k(¢ w \$))$

$\Leftrightarrow$ $(\mathsf{req} ¢ w \$, \mathsf{res}_{q_0} ¢ w \$) \vdash_{\mathcal{M}} (\mathsf{rec}_{q_0} ¢ w \$, \mathsf{ack}_{q_0} ¢ w \$) \vdash_{\mathcal{M}} (q_0 ¢ w \$, \mathsf{req} ¢ w \$)$ and
1) $\kappa = \mathsf{Accept}$ and $q_0 ¢ w \$ \vdash_{M_1,\mathcal{M}} \mathsf{Accept}$, or
2) $\kappa = ¢ q w \$$ for a $q \in Q$ and $q_0 ¢ w \$ \vdash_{M_1,\mathcal{M}} ¢ q w \$$, or
3) $\kappa = ¢ u q v \$$ for a $q \in Q$ and $q_0 ¢ w \$ \vdash_{M_1,\mathcal{M}} ¢ u q v \$$

$\Leftrightarrow$ $\mathsf{req} ¢ w \$ \vdash_{M_1,\mathcal{M}}^3 \kappa$

Induction step: Let $w \in \Sigma^*$ be an arbitrary input word. Then:

$q_0 ¢ w \$ \vdash_M^r \kappa$ $(r > 1)$

$\Leftrightarrow$ There exists a configuration κ' with: $q_0 ¢ w \$ \vdash_M^{r-1} \kappa' \vdash_M \kappa$.
The configuration κ' does not contain a communication state, since it only contains a state from Q. If the last computation step is a MVR, MVL, accept, or rewrite step, then

$$\kappa' \vdash_M \kappa \;\Leftrightarrow\; \kappa' \vdash_{M_1,\mathcal{M}}^* \kappa$$

follows directly from line (3) of the construction. If it is a restart, then there exist states $q \in Q_R$, $p \in Q$ and words x, y, v with $v \in \Gamma^*$ and $xy = ¢ v \$$ so that $\kappa = q ¢ v \$$, $\kappa' = xpy$, and $(\mathsf{Restart}, q) \in \delta(p, \pi_k(y))$.

$\Leftrightarrow$ $\mathsf{req} ¢ w \$ \vdash_{M_1,\mathcal{M}}^* \kappa'$ (induction hypothesis) and

$$\begin{aligned} \delta_1(p, \pi_k(y)) &\ni \mathsf{res}_q, \\ \delta_1(\mathsf{ack}_q, \pi_k(y)) &= \{\mathsf{Restart}\}, \\ \delta_1(\mathsf{rec}_q, \pi_k(xy)) &= \{q\}, \end{aligned} \qquad \begin{aligned} \delta_2(\mathsf{rec}_q, ¢) &= \{\mathsf{res}_q\}, \\ \delta_2(\mathsf{ack}_q, ¢) &= \{\mathsf{req}\} \end{aligned}$$

$\Leftrightarrow$ $\mathsf{req} ¢ w \$ \vdash_{M_1,\mathcal{M}}^* \kappa'$ and

$$\begin{aligned} (\kappa', \mathsf{req} ¢ w \$) = (xpy, \mathsf{req} ¢ w \$) &\vdash_{\mathcal{M}} (x\mathsf{res}_q y, \mathsf{req} ¢ w \$) \\ &\vdash_{\mathcal{M}} (x\mathsf{ack}_q y, \mathsf{rec}_q ¢ w \$) \vdash_{\mathcal{M}} (\mathsf{req} xy, \mathsf{res}_q ¢ w \$) \vdash_{\mathcal{M}} (\mathsf{rec}_q xy, \mathsf{ack}_q ¢ w \$) \\ &\vdash_{\mathcal{M}} (qxy, \mathsf{req} ¢ w \$) = (q ¢ v \$, \mathsf{req} ¢ w \$) = (\kappa, \mathsf{req} ¢ w \$). \end{aligned}$$

Observe that the current state of M_2 is always req except at the very beginning of the computation and during a restart of M_1. It is reached at the beginning of every computation of $\mathcal{M}$ as well as after every restart of M_1. Hence $\kappa' \vdash_{M_1,\mathcal{M}}^* \kappa$.

$\Leftrightarrow$ $\mathsf{req} ¢ w \$ \vdash_{M_1,\mathcal{M}}^* \kappa$

This holds particularly for $\kappa = \mathsf{Accept}$. Furthermore, since M_2 cannot reach the accepting configuration, $\mathcal{M}$ accepts if and only if M_1 accepts. Thus, $L(M) = L(\mathcal{M})$. If M can perform restarts, then M_1 uses SCO transitions. With Corollary 1 and Theorem 2 there exists an automaton M_1' of the same type but without SCO transitions so that $M_1' \equiv_c M_1$ and $L(\mathcal{M}) = L(\mathcal{M}')$ with $\mathcal{M}' = (M_1', M_2)$. □

At the beginning of this section we stated that for many types of restarting automata the usage of the nonforgetting property causes a proper increase in expressive power. Can this be carried over to systems of restarting automata? In [HOV11] it was proved that in the case of PCRA systems with two (monotone) deterministic RRWW-components, the nonforgetting property is of no advantage for the expressive power (independently of monotonicity). Within the next proof we carry this result over the general case, that is, to arbitrary many (at least two) components and for any type of restarting automata.

Theorem 11. *$\mathcal{L}(\textsf{nf-PC-}X(n)) = \mathcal{L}(\textsf{PC-}X(n))$ for all $X \in \mathcal{T}$ and $n \geq 2$.*

Proof. In this proof we combine the constructions of the proofs of Theorems 4, 5, 10, Lemmata 2 and 3, and Corollary 3 as follows: In the proofs of Theorems 4 and 5, an equivalent centralized system was constructed for an arbitrary given (non-centralized) PCRA system. This system contains an additional master component that can also be used to store the restart states of the components like in the proof of Theorem 10. Then, Lemmata 2 and 3 can be used to obtain an equivalent system of the same degree, and finally Corollary 3 is used to avoid SCO transitions.

The way in which the restart state is stored by the master component of the centralized system is described now in more detail. Let $\mathcal{M} = (M, M_1, M_2, \ldots, M_n)$ be the centralized system that is constructed in the proofs of Theorems 4 and 5, where $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ is the master component. With q_i, δ_i, Γ_i, and Q_{Ri} we denote the initial state, the transition relation, the tape alphabet, and the set of restart states of M_i, $1 \leq i \leq n$, respectively. We modify $\mathcal{M}$ in the following way:

1. Before executing a restart operation, a component communicates this fact and the corresponding restart state to the master. For this purpose, all transitions of the form $(\mathsf{Restart}, q) \in \delta_i(p, \alpha)$ are replaced by the transitions $\mathsf{res}_{[restart,q]} \in \delta_i(p, \alpha)$ and $\delta_i(\mathsf{ack}_{[restart,q]}, \alpha) = \{\mathsf{Restart}\}$.

2. After performing a restart step, a component requests its restart state from the master. Therefore, a request state is used as the new initial state. The original initial state q_i is replaced by the new initial state $\mathsf{req}_{restart}$, and transitions $\delta_i(\mathsf{rec}_{restart,[restart,q]}, \alpha) = \{q\}$ are added for all $\alpha \in \{¢\} \cdot (\Gamma_i^{k-1} \cup (\Gamma_i^{\leq k-2} \cdot \{\$\}))$.

3. At the beginning of each computation of $\mathcal{M}$, the master has to tell each component its original initial state and then takes up its own original initial state. Therefore, the new initial state of the master is $\mathsf{res}^1_{[restart,q_1]}$, and the following transitions are added to the master:

$$\delta(\mathsf{ack}^i_{[restart,q_i]}, ¢) = \{\mathsf{res}^{i+1}_{[restart,q_{i+1}]}\} \text{ for all } 1 \leq i < n,$$
$$\delta(\mathsf{ack}^n_{[restart,q_n]}, ¢) = \{\mathsf{req}^1_{\langle *,\ldots,* \rangle}\}.$$

Observe that q_1, q_2, $\ldots$, q_n, and $\mathsf{req}^1_{\langle *,\ldots,* \rangle}$ are the original initial states of M_1, M_2, $\ldots$, M_n, and the master component M. Furthermore, all response and acknowledge states used in these transitions are new states (they are not contained within the original set of states of M).

4. Receiving a restart information (which contains the fact that the component wants to restart as well as the corresponding restart state) from a component, the master has to store this information and to send it back to the component immediately. Hence, we add the following transitions to the master (for all $1 \leq m \leq n$ and for all $q \in Q_{Rm}$):

$$\begin{aligned}\delta(\mathsf{rec}^m_{\langle t_1,\ldots,t_n\rangle,[restart,q]}, ¢) &= \{\mathsf{res}^m_{\langle t_1,\ldots,t_n\rangle,[restart,q]}\},\\ \delta(\mathsf{ack}^m_{\langle t_1,\ldots,t_n\rangle,[restart,q]}, ¢) &= \{\mathsf{req}^m_{\langle t_1,\ldots,t_n\rangle}\}.\end{aligned}$$

Here m is the index of the currently asked component, and $t_1, t_2, \ldots, t_n$ are the situations of $M_1, M_2, \ldots, M_n$ (see the proof of Theorem 4 for further details).

Let $\mathcal{M}' = (M', M_1', M_2', \ldots, M_n')$ be the system that we obtain by applying the modifications described above to $\mathcal{M}$. The initial configuration of $\mathcal{M}'$ for an input word w is

$$(\mathsf{res}^1_{[restart,q_1]}¢w\$, \mathsf{req}_{restart}¢w\$, \ldots, \mathsf{req}_{restart}¢w\$),$$

and each computation of $\mathcal{M}'$ starts with

$$\begin{aligned}
& (\mathsf{res}^1_{[restart,q_1]}¢w\$, \mathsf{req}_{restart}¢w\$, \ldots, \mathsf{req}_{restart}¢w\$)\\
\vdash_{\mathcal{M}'} \; & (\mathsf{ack}^1_{[restart,q_1]}¢w\$, \mathsf{rec}_{restart,[restart,q_1]}¢w\$, \mathsf{req}_{restart}¢w\$, \ldots, \mathsf{req}_{restart}¢w\$)\\
\vdash_{\mathcal{M}'} \; & (\mathsf{res}^2_{[restart,q_2]}¢w\$, q_1¢w\$, \mathsf{req}_{restart}¢w\$, \ldots, \mathsf{req}_{restart}¢w\$)\\
\vdash_{\mathcal{M}'} \; & (\mathsf{ack}^2_{[restart,q_2]}¢w\$, *, \mathsf{rec}_{restart,[restart,q_2]}¢w\$, \mathsf{req}_{restart}¢w\$, \ldots, \mathsf{req}_{restart}¢w\$)\\
\vdash_{\mathcal{M}'} \; & (\mathsf{res}^3_{[restart,q_3]}¢w\$, *, q_2¢w\$, \mathsf{req}_{restart}¢w\$, \ldots, \mathsf{req}_{restart}¢w\$)\\
& \qquad\vdots\\
\vdash_{\mathcal{M}'} \; & (\mathsf{res}^n_{[restart,q_n]}¢w\$, *, \ldots, *, \mathsf{req}_{restart}¢w\$)\\
\vdash_{\mathcal{M}'} \; & (\mathsf{ack}^n_{[restart,q_n]}¢w\$, *, \ldots, *, \mathsf{rec}_{restart,[restart,q_n]}¢w\$)\\
\vdash_{\mathcal{M}'} \; & (\mathsf{req}^1_{\langle *,\ldots,*\rangle}¢w\$, *, \ldots, *, q_n¢w\$).
\end{aligned}$$

After reaching the original initial configuration $q_i¢w\$$, a component M_i starts its local computation without waiting for the other components. In this case $*$ is used to denote an arbitrary local successor configuration. Observe that the computation $\mathsf{req}_{restart}¢w\$ \vdash_{M_i',\mathcal{M}'} q_i¢w\$$ is deterministic for all components M_i', so that it is assured that each component reaches its original initial configuration. Now, we can prove

$$q_i¢w\$ \vdash_{M_i,\mathcal{M}} \kappa \quad \Leftrightarrow \quad q_i¢w\$ \vdash^*_{M_i',\mathcal{M}'} \kappa$$

for all $1 \leq i \leq n$, $w \in \Sigma^*$, and configurations κ not containing a communication state by induction on the number of restarts in the computation. If the computation does not contain a restart, the equivalence holds, as the modifications above only influence the application of the restart operation but not any other operation. Moreover, the following equivalences hold:

Let $q_i\text{¢}w\$ \vdash^*_{M_i,\mathcal{M}} \kappa$ contain $n+1$ restarts ($n \geq 0$).

$\Leftrightarrow$ $\exists p \in Q_i, q \in Q_{Ri}$, words u, v with $uv \in \{\text{¢}\} \cdot \Gamma_i^* \cdot \{\$\}$:

$$q_i\text{¢}w\$ \vdash^*_{M_i,\mathcal{M}} upv \vdash_{M_i,\mathcal{M}} quv \vdash^*_{M_i,\mathcal{M}} \kappa,$$

where $upv \vdash_{M_i,\mathcal{M}} quv$ is a restart (w.l.o.g. the last restart of the computation), $q_i\text{¢}w\$ \vdash^*_{M_i,\mathcal{M}} upv$ contains n restarts, and $quv \vdash^*_{M_i,\mathcal{M}} \kappa$ contains no restart.

$\Leftrightarrow$ $\exists p \in Q_i, q \in Q_{Ri}$, words u, v with $uv \in \{\text{¢}\} \cdot \Gamma_i^* \cdot \{\$\}$:
$q_i\text{¢}w\$ \vdash^*_{M'_i,\mathcal{M}'} upv, quv \vdash^*_{M'_i,\mathcal{M}'} \kappa$ (because of the induction basis and hypothesis), and $(\mathsf{Restart}, q) \in \delta_i(p, \pi_k(v))$.

$\Leftrightarrow$ $\exists p \in Q_i, q \in Q_{Ri}$, words u, v with $uv \in \{\text{¢}\} \cdot \Gamma_i^* \cdot \{\$\}$:
$q_i\text{¢}w\$ \vdash^*_{M'_i,\mathcal{M}'} upv, quv \vdash^*_{M'_i,\mathcal{M}'} \kappa$, $\mathsf{res}_{[restart,q]} \in \delta'_i(p, \pi_k(v))$,
$\delta'_i(\mathsf{ack}_{[restart,q]}, \pi_k(v)) = \{\mathsf{Restart}\}$,
$\delta'(\mathsf{rec}^i_{\langle t_1,\ldots,t_n\rangle,[restart,q]}, \text{¢}) = \{\mathsf{res}^i_{\langle t_1,\ldots,t_n\rangle,[restart,q]}\}$,
and $\delta'(\mathsf{ack}^i_{\langle t_1,\ldots,t_n\rangle,[restart,q]}, \text{¢}) = \{\mathsf{req}^i_{\langle t_1,\ldots,t_n\rangle}\}$.

$\Leftrightarrow$ $\exists p \in Q_i, q \in Q_{Ri}$, words u, v with $uv \in \{\text{¢}\} \cdot \Gamma_i^* \cdot \{\$\}$: (here only the interaction between M' and M'_i is considered)

$$\begin{aligned}
& (\mathsf{res}^1_{[restart,q]}, \ldots, \mathsf{req}_{restart}, \ldots) \\
\vdash^*_{M'_i,\mathcal{M}'} \quad & (\mathsf{req}^i_{\langle t_1,\ldots,t_n\rangle}, \ldots, u\mathsf{res}_{[restart,q]}v, \ldots) \\
\vdash_{M'_i,\mathcal{M}'} \quad & (\mathsf{rec}^i_{\langle t_1,\ldots,t_n\rangle,[restart,q]}, \ldots, u\mathsf{ack}_{[restart,q]}v, \ldots) \\
\vdash_{M'_i,\mathcal{M}'} \quad & (\mathsf{res}^i_{\langle t_1,\ldots,t_n\rangle,[restart,q]}, \ldots, \mathsf{req}_{restart}uv, \ldots) \\
\vdash_{M'_i,\mathcal{M}'} \quad & (\mathsf{ack}^i_{\langle t_1,\ldots,t_n\rangle,[restart,q]}, \ldots, \mathsf{rec}_{restart,[restart,q]}uv, \ldots) \\
\vdash_{M'_i,\mathcal{M}'} \quad & (\mathsf{req}^i_{\langle t_1,\ldots,t_n\rangle}, \ldots, quv, \ldots)
\end{aligned}$$

and $quv \vdash^*_{M'_i,\mathcal{M}'} \kappa$.

$\Leftrightarrow$ $\exists p \in Q_i, q \in Q_{Ri}$, words u, v with $uv \in \{\text{¢}\} \cdot \Gamma_i^* \cdot \{\$\}$:

$$\begin{aligned}
& q_i\text{¢}w\$ \vdash^*_{M'_i,\mathcal{M}'} upv \vdash_{M'_i,\mathcal{M}'} u\mathsf{res}_{[restart,q]}v \vdash_{M'_i,\mathcal{M}'} u\mathsf{ack}_{[restart,q]}v \\
& \quad \vdash_{M'_i,\mathcal{M}'} \mathsf{req}_{restart}uv \vdash_{M'_i,\mathcal{M}'} \mathsf{rec}_{restart,[restart,q]}uv \vdash_{M'_i,\mathcal{M}'} quv \vdash^*_{M'_i,\mathcal{M}'} \kappa.
\end{aligned}$$

Since this holds particularly for $\kappa = \mathsf{Accept}$, it follows that $L(\mathcal{M}) = L(\mathcal{M}')$. Observe that due to step two of the construction, the modified components can contain SCO transitions. By Lemmata 2 and 3 and Corollary 3, there exists an equivalent system of the same type, the same number of components, and without

SCO transitions. □

Remark. *Theorems 10 and 11 hold for deterministic and/or monotone variants as well, since the modifications are strictly deterministic and influence only the communicational behaviour.*

Corollary 10. *For all $\mathsf{X} \in \mathcal{T}$, if $\mathcal{L}(\mathsf{X}) \subset \mathcal{L}(\mathsf{nf}\text{-}\mathsf{X})$, then $\mathcal{L}(\mathsf{X}) \subset \mathcal{L}(\mathsf{PC}\text{-}\mathsf{X})$.*

Above we saw that every nonforgetting automaton can be simulated by a PC system of (forgetting) restarting automata and, moreover, the property of being nonforgetting is of no advantage for PCRA systems. Now we take a look at the opposite direction, that is, we ask in how far a nonforgetting automaton can simulate a PCRA system? Compared to nonforgetting automata, PCRA systems have important advantages:

1. more working space,
2. parallel reading and modification of the whole input at different positions, and
3. arbitrary much communication.

The next theorem shows how the behaviour of a PCRA system can be simulated by a nonforgetting automaton. For this, every component is only allowed to execute at most a constant number of communication steps within one cycle (independently of the length of the input). To represent the different tapes of the components, the input of the nonforgetting automaton is encoded as

$$¢_1 w\$_1 ¢_2 w\$_2 \ldots ¢_n w\$_n,$$

where w is the input of (each component of) the system, and n is the degree of the system. The parallel computation of the system will be serialized by the nonforgetting automaton in such a way that one cycle of a distinguished component corresponds to one cycle of the nonforgetting automaton.

Theorem 12. *Let $\mathcal{M}$ be a PCRA system of degree n in which every component is of type $\mathsf{X} \in \mathcal{T}$ and executes at most $m \in \mathbb{N}$ communication steps per cycle. Then there exists an nf-X-automaton M that behaves as follows: whenever the input is $¢_1 w\$_1 ¢_2 w\$_2 \ldots ¢_n w\$_n$ for a word $w \in \Sigma^*$, then M accepts if and only if $\mathcal{M}$ accepts w. The symbols $¢_i$ and $\$_i$, $1 \leq i \leq n$, are the mutually different sentinels of the i-th component of $\mathcal{M}$ and are not included in Σ.*

Proof. Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PCRA system of type X, $\mathsf{X} \in \mathcal{T}$, and degree n with $M_i = (Q_i, \Sigma, \Gamma_i, ¢_i, \$_i, q_0^{(i)}, k, \delta_i)$, where every component executes at most m communication steps per cycle. We construct an nf-X-automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_{init}, k, \delta, Q_R)$ as follows. The states of M are of the form (q, T), where

$$T = \begin{array}{r|l} & \text{Sequences} \\ \hline 1 & (com_1^{(1)}, \ldots, com_{m_1}^{(1)}[, x^{(1)}]) \\ 2 & (com_1^{(2)}, \ldots, com_{m_2}^{(2)}[, x^{(2)}]) \\ \vdots & \vdots \\ n & (com_1^{(n)}, \ldots, com_{m_n}^{(n)}[, x^{(n)}]) \end{array}$$

is a finite table that contains a sequence of at most m receive and acknowledge states and possibly Accept or $\perp$ at the end for each component ($0 \leq m_1, m_2, \ldots, m_n \leq m, com_j^{(i)} \in \mathsf{REC}(M_i) \cup \mathsf{ACK}(M_i)$ for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m_i\}$, $x^{(i)} \in \{\mathsf{Accept}, \perp\}$ for all $i \in \{1, \ldots, n\}$). Below we write

$$T = ((com_1^{(1)}, \ldots, com_{m_1}^{(1)}[, x^{(1)}]), \ldots, (com_1^{(n)}, \ldots, com_{m_n}^{(n)}[, x^{(n)}])) = (S_1, S_2, \ldots, S_n)$$

denoting the i-th sequence with S_i, and $S = \emptyset$ for an empty sequence. The sequences can be seen as queues, where an element can only be removed from the left and added at the right. The complete set of states of M is given indirectly through the description of the transition relation below. The initial state of M is $q_{init} = (q_0, (\emptyset, \ldots, \emptyset))$. The tape alphabet is

$$\Gamma = \bigcup_{1 \leq i \leq n} (\Gamma_i \cup \{¢_i, \$_i\}).$$

Now, we describe the way in which M simulates a computation of $\mathcal{M}$. A cycle of M is divided into four phases:

1. Resolve communications.
2. Check whether any component reached the accepting configuration. If so, then accept the input.
3. Choose the next component to simulate.
4. Simulate the chosen component.

An important fact is that M is not able to store the current window positions of the simulated components in its finite control. Hence, M may not change the simulated component during a simulation of a cycle. Otherwise, the window position of the previously simulated component is lost, and its simulation cannot be continued at the same tape position. Thus, M always has to simulate a complete cycle of a component, which is finished by a restart operation. Then, in the next cycle, the window position of the component simulated is the left end of the tape section of this component. Thus, a communication step cannot be resolved immediately, at least if the corresponding communication state has not yet been reached by the communication partner. Therefore, M guesses a possible answer nondeterministically, stores the chosen communication state in the corresponding sequence, and continues with the local computation. Since each component only applies at most m communications per cycle, all sequences have a constant maximal length. Having finished a cycle, the component and therewith M restarts. Since M is a nonforgetting automaton, the sequences do not get lost during the restart operation.

At the beginning of each cycle, M can resolve communications. This happens through modifying the table within the current state using an SCO transition, where two corresponding states, which are the first elements of two different sequences, are removed. In this phase M verifies whether the guessed communication steps of the different components correspond to each other. Thereafter, M can accept the input if and only if there is a sequence $S_i = (\mathsf{Accept})$. This means that there is a component M_i that has reached the accepting configuration, and all of its communications have been resolved.

Until here, the window of M remains in the leftmost position of the tape. If M does not accept, it can now simulate another component. For this, only a component with an empty sequence is chosen. A non-empty sequence S_i means that there are still some communications of M_i that must be resolved before the local computation of M_i can continue.

Formally, the behaviour of M is defined by the following transitions:

1. Move right and choose a component:

$$\begin{aligned} ((q_0,T),\mathsf{MVR}) &\in \delta((q_0,T),\alpha) \text{ for all } T \text{ and } \alpha \in \mathcal{PC}_M, \\ (q_0^{(i)},T) &\in \delta((q_0,T),\text{¢}_i\alpha) \text{ for all } T \text{ s.t. } S_i=\emptyset, \text{ and all } \text{¢}_i\alpha \in \mathcal{PC}_M. \end{aligned}$$

   A component is chosen nondeterministically and only if its sequence is empty. A non-empty sequence means that there is at least one communication left to be resolved or that this component is stuck (when $\perp$ occurs in the sequence). In both cases there is no need to go on with simulating this component.

2. Simulate a component:
   First we define subsets of possible window contents of M depending on the simulated component and its window content:

$$U_{\alpha,i} = \begin{cases} \{\alpha\}, & \text{if } \alpha \text{ neither contains } ¢ \text{ nor } \$, \\ \{¢_i\beta \mid \alpha = ¢\beta\}, & \text{if } \alpha \text{ contains } ¢ \text{ but not } \$, \\ \{\beta\$_i\gamma \mid \alpha = \beta\$ \text{ and } \beta\$_i\gamma \in \mathcal{PC}_M\}, & \text{if } \alpha \text{ contains not } ¢ \text{ but } \$, \\ \{¢_i\beta\$_i\gamma \mid \alpha = ¢\beta\$ \text{ and } ¢_i\beta\$_i\gamma \in \mathcal{PC}_M\}, & \text{if } \alpha \text{ contains } ¢ \text{ and } \$. \end{cases}$$

(a) For all transitions of the form $(p, \mathsf{MVR}) \in \delta_i(q, \alpha)$,

$$((p^{(i)}, T), \mathsf{MVR}) \in \delta((q^{(i)}, T), \alpha')$$

for all possible T and all $\alpha' \in U_{\alpha,i}$. The MVL and rewrite steps are treated similarly although in a rewrite step it must be ensured that only symbols of the currently simulated component are rewritten. If the window of M contains symbols of the beginning of the tape section of the next component, these symbols are not allowed to be replaced or removed.

(b) For all transitions of the form $\mathsf{Restart} \in \delta_i(q, \alpha)$,

$$(\mathsf{Restart}, (q_0, T)) \in \delta((q^{(i)}, T), \alpha')$$

for all possible T and all $\alpha' \in U_{\alpha,i}$.

(c) For all transitions of the form $\mathsf{Accept} \in \delta_i(q, \alpha)$,

i. $((q_r, T'), \alpha'') \in \delta((q^{(i)}, T), \alpha')$ for all T and all $\alpha' \in U_{\alpha,i}$. The new table T' results from T by adding Accept to S_i, and α'' results from α' by deleting the first symbol. Since it is not allowed for M_i to move the window over the left and the right border of the tape, i.e. to the left of ¢ and to the right of \$, the first symbol of M's window is contained within the tape section that belongs to M_i during the whole simulation. Which symbol of M_i's part of the tape is deleted does not matter (even if it is $¢_i$ or $\$_i$), because the local computation of M_i has finished at this point. If a rewrite step already took place in the current cycle, then no additional rewrite step is necessary to apply the following restart operation. Whether a rewrite operation was already executed in the current cycle or not can be stored within the states.

ii. $(\mathsf{Restart}, (q_0, T)) \in \delta((q_r, T), \alpha')$ for all T and all $\alpha' \in \mathcal{PC}_M$.

The system may not accept although the component simulated has reached the accepting configuration. There could be some communications of the same cycle that have still not been resolved. Thus, the information about the acceptance of M_i is stored in S_i and can be used after all remaining communications of M_i have been resolved.

(d) For all transitions of the form $\mathsf{req}_d^j \in \delta_i(q, \alpha)$,

$$((\mathsf{rec}_{d,c}^j)^{(i)}, T') \in \delta((q^{(i)}, T), \alpha')$$

for all tables T, all $\alpha' \in U_{\alpha,i}$, all c with $\mathsf{res}_{d',c}^i \in \mathsf{RES}(M_j)$ and $\mathsf{rec}_{d,c}^j \in \mathsf{REC}(M_i)$, and S_i' in T' results from S_i in T by adding $\mathsf{rec}_{d,c}^j$.

(e) For all transitions of the form $\mathsf{res}_{d,c}^j \in \delta_i(q, \alpha)$,

$$((\mathsf{ack}_{d,c}^j)^{(i)}, T') \in \delta((q^{(i)}, T), \alpha')$$

for all tables T and all $\alpha' \in U_{\alpha,i}$, where S_i' in T' results from S_i in T by adding $\mathsf{ack}_{d,c}^j$.

(f) For all undefined transitions $\delta_i(q, \alpha) = \emptyset$,

i. $((q_r, T'), \alpha'') \in \delta((q^{(i)}, T), \alpha')$ for all T and all $\alpha' \in U_{\alpha,i}$. T' results from T by adding $\perp$ to S_i, and α'' results from α by deleting the first symbol as described in item 2(c)i.

ii. $(\mathsf{Restart}, (q_0, T)) \in \delta((q_r, T), \alpha')$ for all T and all $\alpha' \in \mathcal{PC}_M$ (if this is not already given by (c)ii).

Now, S_i cannot become empty again, since $\perp$ cannot be removed from a sequence. Thus, M never tries to simulate M_i again in this computation. Nevertheless, all (unresolved) communications of M_i (of the last cycle) are stored in S_i, and therefore, they are available within the simulated computations of the other components.

3. Resolve communications:
Define for all $\alpha \in \mathcal{PC}_M$

$$(q_0, T') \in \delta((q_0, T), \alpha)$$

if and only if there exist two sequences $S_i = (com_1^{(i)}, com_2^{(i)}, \ldots, com_{m_i}^{(i)}[, x^{(i)}])$ and $S_j = (com_1^{(j)}, com_2^{(j)}, \ldots, com_{m_j}^{(j)}[, x^{(j)}])$ in T such that $com_1^{(i)}$ and $com_1^{(j)}$ are corresponding communication states, and T' results from T by replacing S_i and S_j with $S_i' = (com_2^{(i)}, \ldots, com_{m_i}^{(i)}[, x^{(i)}])$ and $S_j' = (com_2^{(j)}, \ldots, com_{m_j}^{(j)}[, x^{(j)}])$.

4. Accept the input:
Define

$$\mathsf{Accept} \in \delta((q_0, T), \alpha)$$

for all $\alpha \in \mathcal{PC}_M$ and for all T that contain a sequence $S_i = (\mathsf{Accept})$. If T contains such a sequence, then M_i was simulated, it reached the accepting configuration, and moreover, all communications within the computation of M_i were resolved.

Now, each computation of $\mathcal{M}$ is simulated by M in a sequential manner. The currently simulated component is chosen nondeterministically. The different computation phases are not necessarily executed in the given order (resolve communications, check for acceptance, simulate component). This is possible, as they are independent of each other. If there exists an accepting computation of $\mathcal{M}$, then there exists an order of simulating the components such that M adds Accept to the according sequence, all communications of that component are resolved, and M accepts. If $\mathcal{M}$ does not accept the input, then M cannot reach the accepting configuration either. Moreover, if the components of $\mathcal{M}$ are of type X, then M is of type nf-X.

□

We can assume that the restriction of the constant number of communication steps per cycle in Theorem 12 is indeed essential, and that a system with more than constantly many communications per cycle cannot be simulated by any nonforgetting automaton. As mentioned above there are situations where a communication cannot be resolved immediately (e.g. the communication partner has not reached the corresponding communication state yet). On the other hand, if the nonforgetting automaton tried to simulate the communication partner just for resolving the communication, the information about the current window position of the first simulated component would get lost. So there are two options: storing all communications of a cycle in the finite control or storing the current window position on the tape for each communication step. The first approach only works for constantly many communications per cycle as in Theorem 12. The second approach does not work, as for storing the current window position, the tape has to be shortened for each communication step. Thus, only at most as many communication steps as the length of the working tape content of the simulated component can be simulated. Moreover, since the tape content should not be erased, linearly many auxiliary symbols would be needed. This is obviously a contradiction to the definition of nonforgetting restarting automata.

5.5 Closure properties

In this section some typical closure properties of the language classes characterized by PCRA systems are investigated.

Theorem 13. *The language classes $\mathcal{L}(\textsf{(det-)PC-X})$ are closed under union for all $\textsf{X} \in \mathcal{T}$.*

Proof. Let $L_1, L_2 \in \mathcal{L}(\textsf{(det-)PC-X})$ for any $\textsf{X} \in \mathcal{T}$. Then there exist two PCRA systems $\mathcal{M} = (M_1, M_2, \ldots, M_m)$ and $\mathcal{N} = (N_1, N_2, \ldots, N_n)$ of type $\textsf{(det-)PC-X}$ with $L(\mathcal{M}) = L_1$ and $L(\mathcal{N}) = L_2$. Now, the system $\mathcal{A} = (M_1, M_2, \ldots, M_m, N_1, N_2, \ldots, N_n)$ accepts exactly the language $L(\mathcal{M}) \cup L(\mathcal{N}) = L_1 \cup L_2$, as it accepts an input if and only if at least one component of $\mathcal{M}$ or $\mathcal{N}$ reaches the accepting configuration (that is, if and only if $\mathcal{M}$ or $\mathcal{N}$ accepts). Moreover, $\mathcal{A}$ is of the same type as $\mathcal{M}$ and $\mathcal{N}$, since $\mathcal{A}$ uses only the original components of $\mathcal{M}$ and $\mathcal{N}$. Thus, $L_1 \cup L_2 \in \mathcal{L}(\textsf{(det-)PC-X})$. □

Theorem 14. *The language classes $\mathcal{L}(\textsf{(det-)PC-X})$ are closed under intersection for all $\textsf{X} \in \mathcal{T}$.*

Proof. Let L_1 and L_2 be two languages over an alphabet Σ with $L_1, L_2 \in \mathcal{L}(\textsf{(det-)PC-X})$ for any $\textsf{X} \in \mathcal{T}$. Then there exist two PCRA systems $\mathcal{M} = (M_1, M_2, \ldots, M_m)$ and $\mathcal{N} = (N_1, N_2, \ldots, N_n)$ of type $\textsf{(det-)PC-X}$ with $L(\mathcal{M}) = L_1$ and $L(\mathcal{N}) = L_2$. Due to Corollary 4 and Corollary 5, there exist centralized systems $\mathcal{M}' = (M'_1, M'_2, \ldots, M'_m)$ and $\mathcal{N}' = (N'_1, N'_2, \ldots, N'_n)$ with $L(\mathcal{M}) = L(\mathcal{M}')$ and $L(\mathcal{N}) = L(\mathcal{N}')$ such that $\mathcal{M}'$ accepts iff M'_1 accepts, and $\mathcal{N}'$ accepts iff N'_1 accepts. Now, construct a system $\mathcal{A}$ as follows:

$$\mathcal{A} = (A, M'_1, M'_2, \ldots, M'_m, N'_1, N'_2, \ldots, N'_n),$$

where $A = (\{\textsf{req}^{M'_1}, \textsf{req}^{N'_1}, \textsf{rec}^{M'_1}, \textsf{rec}^{N'_1}\}, \Sigma, \Sigma, ¢, \$, \textsf{req}^{M'_1}, 1, \delta_A)$ with

$$\begin{aligned}\delta_A(\textsf{rec}^{M'_1}, ¢) &= \{\textsf{req}^{N'_1}\},\\ \delta_A(\textsf{rec}^{N'_1}, ¢) &= \{\textsf{Accept}\}.\end{aligned}$$

Transitions of M'_1 and N'_1 of the form $\textsf{Accept} \in \delta_{M'_1}(q, \alpha)$ or $\textsf{Accept} \in \delta_{N'_1}(q, \alpha)$ are replaced by $\textsf{res}^A \in \delta_{M'_1}(q, \alpha)$ or $\textsf{res}^A \in \delta_{N'_1}(q, \alpha)$, respectively, for any state q and possible window content α. The components M'_1 and N'_1 now answer to the communication of A instead of accepting by themselves. A (and therefore $\mathcal{A}$) only accepts if both M'_1 and N'_1 answer the communication. On the other hand, if both components answer, A reaches the accepting configuration and $\mathcal{A}$ accepts. Thus, $L(\mathcal{A}) = L(\mathcal{M}) \cap L(\mathcal{N}) = L_1 \cap L_2$. In addition, the resulting system $\mathcal{A}$ has the same type as $\mathcal{M}$ and $\mathcal{N}$. Hence, $L_1 \cap L_2 \in \mathcal{L}(\textsf{(det-)PC-X})$. □

Since we know from [JMPV97] that even restarting automata of the weakest type ($\mathsf{det\text{-}mon\text{-}R}$) accept all deterministic context free languages, and therewith, all regular languages, we immediately obtain:

Corollary 11. *The language classes $\mathcal{L}((\mathsf{det\text{-}})\mathsf{PC\text{-}X})$ are closed under intersection with regular languages for all $\mathsf{X} \in \mathcal{T}$.*

Before we deal with the operation of complementation, we point out a problem that may occur in connection with communication. Consider the following situation: a system is in a particular configuration, and after executing communication steps of some components, the system reaches the same configuration again. We will call such a situation a *communication loop.* If a deterministic system reaches such a communication loop, then the computation will not finish. In this case, the input word is not accepted. In contrast, a nondeterministic system could leave such a loop, and moreover, such loops have no effect on whether the input is accepted or not. Thus, if the input is accepted by a nondeterministic system, then there exists a corresponding accepting computation without a communication loop.

In Theorem 28 of Section 5.7 the decidability of the membership problem for deterministic systems is proved. Moreover, it is shown in detail that there exists a constant positive integer r such that a deterministic system can perform at most r computation steps in a row before it must reach a loop and thus an infinite computation. This fact is used in the proof of the following theorem.

Theorem 15. *The language classes $\mathcal{L}(\mathsf{det\text{-}PC\text{-}X})$ are closed under complement for all $\mathsf{X} \in \mathcal{T}$.*

Proof. Let $L \in \mathcal{L}(\mathsf{det\text{-}PC\text{-}X})$ for any $\mathsf{X} \in \mathcal{T}$ and let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a $\mathsf{det\text{-}PC\text{-}X}$-system with $L(\mathcal{M}) = L$ and $M_i = (Q_i, \Sigma, \Gamma_i, ¢, \$, q_i, k, \delta_i)$ for all $1 \leq i \leq n$. We can assume that $\mathcal{M}$ is centralized and that it accepts an input word if and only if the first component M_1 accepts it (see Corollary 5). For a given input, $\mathcal{M}$ can behave in the following three ways:

1. $\mathcal{M}$ halts and accepts the input,
2. $\mathcal{M}$ halts, but does not accept the input (because all components are stuck), or
3. $\mathcal{M}$ reaches an infinite computation, and hence, does not accept the input.

Observe that M_1, and therewith $\mathcal{M}$, cannot get into a communication deadlock, due to the construction of a centralized system presented in Section 5.3. The last of the three cases above requires to consider infinite computations that are caused either by an according combination of MVL steps and MVR steps, or by

SCO transitions, or by a communication loop. In Theorem 28 of Section 5.7 it is proved that the number of computation steps of each accepting computation of a det-PC-RLWW-system $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ has the upper bound

$$r = n \cdot \left(\max_{1 \leq i \leq n} |Q_i|\right) \frac{l^2 + 5l + 4}{2} \cdot \prod_{i=1}^{n} |\,\mathsf{COM}(M_i)|,$$

where Q_i is the set of states of component M_i, $\mathsf{COM}(M_i)$ is the set of communication states of M_i, and l is the length of the input word. Due to this upper bound r we know the following: whenever M_1 accepts an input, then it must do so within the first r steps of $\mathcal{M}$'s computation.

Now, we construct a system $\mathcal{M}' = (M_1', M_2, \ldots, M_n, M_{n+1})$ that accepts the language $\overline{L}$, which is the complement of L, with the modified master component M_1', the (unchanged) components $M_2, \ldots, M_n$ of the system $\mathcal{M}$, and an additional component M_{n+1}. Basically, $\mathcal{M}'$ simulates $\mathcal{M}$. If M_1 reaches the accepting configuration, then M_1' halts and does not accept. Since no other component of $\mathcal{M}$ is allowed to accept, $\mathcal{M}'$ does not accept. In the case that M_1 gets stuck because of an undefined transition and thus does not accept, M_1' and therewith $\mathcal{M}'$ accept the input. Further, $\mathcal{M}$ can reach an infinitely long computation. This can happen either because M_1 reaches an infinite computation itself, or M_1 halts without accepting and at least one of the other components is in a computation loop. In the latter case, M_1' reaches the accepting configuration as described above. In the former case, the new component M_{n+1} is used to count the steps of M_1'. For this purpose, M_1' communicates with M_{n+1} after each of its computation steps, and M_{n+1} is constructed in such a way that it communicates finitely often and more than r times with M_1' and then accepts the input. Thus, M_{n+1} accepts if and only if M_1 reaches a computation loop. In summery, $\mathcal{M}'$ accepts if and only if $\mathcal{M}$ does not accept.

The components M_1' and M_{n+1} are formally defined as follows:

$$M_1' = (Q_1', \Sigma, \Gamma_1, ¢, \$, q_1, k, \delta_1'),$$

where

$$Q_1' = Q_1 \cup \{\mathsf{res}_{[p,\alpha]}^{n+1}, \mathsf{ack}_{[p,\alpha]}^{n+1} \mid \delta_1(p, \alpha) \neq \emptyset\}$$

and

- for all $\delta_1(p, \alpha) = A$ with $A \neq \mathsf{Accept}$, $\delta_1'(p, \alpha) = \mathsf{res}_{[p,\alpha]}^{n+1}$ and $\delta_1'(\mathsf{ack}_{[p,\alpha]}^{n+1}, \alpha) = A$,
- for all $\delta_1(p, \alpha) = \mathsf{Accept}$, $\delta_1'(p, \alpha) = \emptyset$, and

- for all $\delta_1(p,\alpha)=\emptyset$, $\delta_1'(p,\alpha)=\mathsf{Accept}$.

Consider that the subscripts $[p,\alpha]$ of the response and acknowledge states are interpreted as local information and no explicit message is sent. Further, let

$$t = n \cdot \left(\max_{1\le i\le n} |Q_i| \right)^{n+1}$$

such that

$$r = n \cdot \left(\max_{1\le i\le n} |Q_i| \right) \frac{l^2+5l+4}{2} \cdot \prod_{i=1}^{n} |\,\mathsf{COM}(M_i)| \le t \cdot \frac{l^2+5l+4}{2}.$$

Then the automaton M_{n+1} is given by

$$M_{n+1} = (Q_{n+1}, \Sigma, \Sigma, ¢, \$, \mathsf{req}_1^1, 2, \delta_{n+1}),$$

where

$$Q_{n+1} = \{\mathsf{req}_i^1, \mathsf{rec}_i^1 \mid 1 \le i \le 2t\} \cup \{q, \mathsf{req}_{accept}^1, \mathsf{rec}_{accept}^1\},$$

and

$$\begin{array}{lll}
\delta_{n+1}(\mathsf{rec}_i^1, \alpha) & = \mathsf{req}_{i+1}^1 & \text{for all } 1 \le i < 2t \text{ and } \alpha \in (\{¢\} \cup \Sigma) \cdot (\Sigma \cup \{\$\}), \\
\delta_{n+1}(\mathsf{rec}_{2t}^1, \alpha) & = (q, \mathsf{MVR}) & \text{for all } \alpha \in (\{¢\} \cdot \Sigma) \cup \Sigma^2, \\
\delta_{n+1}(q, \alpha) & = \mathsf{req}_1^1 & \text{for all } \alpha \in \Sigma^2 \cup (\Sigma \cdot \{\$\}), \\
\delta_{n+1}(\mathsf{rec}_{2t}^1, a\$) & = (q, \$) & \text{for all } a \in \Sigma, \\
\delta_{n+1}(q, \$) & = \mathsf{Restart}, & \\
\delta_{n+1}(\mathsf{rec}_{2t}^1, ¢\$) & = \mathsf{req}_{accept}^1, & \\
\delta_{n+1}(\mathsf{rec}_{accept}^1, ¢\$) & = \mathsf{Accept}. &
\end{array}$$

As long as M_1' communicates with M_{n+1}, a cycle of M_{n+1} consists of the following computation steps for an input word $w = w_1w_2\ldots w_l$ and $l > 0$:

$$\begin{array}{rl|rl}
 & \mathsf{req}_1^1 ¢ w \$ & \vdash_{M_{n+1},\mathcal{M}'} & ¢\, \mathsf{rec}_{2t}^1 w \$ \\
\vdash_{M_{n+1},\mathcal{M}'} & \mathsf{rec}_1^1 ¢ w \$ & \vdash_{M_{n+1},\mathcal{M}'} & ¢\, w_1 q w_2 \ldots w_l \$ \\
\vdash_{M_{n+1},\mathcal{M}'} & \mathsf{req}_2^1 ¢ w \$ & \vdash_{M_{n+1},\mathcal{M}'}^{(l-2)(4t+1)} & ¢\, w_1 \ldots w_{l-1} q w_l \$ \\
\vdash_{M_{n+1},\mathcal{M}'} & \mathsf{rec}_2^1 ¢ w \$ & \vdash_{M_{n+1},\mathcal{M}'} & ¢\, w_1 \ldots w_{l-1} \mathsf{req}_1^1 w_l \$ \\
\vdash_{M_{n+1},\mathcal{M}'}^{4t-5} & \mathsf{req}_{2t}^1 ¢ w \$ & \vdash_{M_{n+1},\mathcal{M}'}^{4t-2} & ¢\, w_1 \ldots w_{l-1} \mathsf{req}_{2t}^1 w_l \$ \\
\vdash_{M_{n+1},\mathcal{M}'} & \mathsf{rec}_{2t}^1 ¢ w \$ & \vdash_{M_{n+1},\mathcal{M}'} & ¢\, w_1 \ldots w_{l-1} \mathsf{rec}_{2t}^1 w_l \$ \\
\vdash_{M_{n+1},\mathcal{M}'} & ¢\, q w \$ & \vdash_{M_{n+1},\mathcal{M}'} & ¢\, w_1 \ldots w_{l-1} q \$ \\
\vdash_{M_{n+1},\mathcal{M}'} & ¢\, \mathsf{req}_1^1 w \$ & \vdash_{M_{n+1},\mathcal{M}'} & \mathsf{req}_1^1 ¢ w_1 \ldots w_{l-1} \$. \\
\vdash_{M_{n+1},\mathcal{M}'}^{4t-2} & ¢\, \mathsf{req}_{2t}^1 w \$ & &
\end{array}$$

The number of communications between M_1' and M_{n+1} in a cycle that can be obtained from the computation above is $2t(l+1)$. For an input ε, i.e. $l=0$, M_{n+1}

performs the following computation:

$$\begin{aligned}
& \mathsf{req}_1^1¢\$ \\
\vdash_{M_{n+1},\mathcal{M}'} \ & \mathsf{rec}_1^1¢\$ \\
\vdash^{4t-3}_{M_{n+1},\mathcal{M}'} \ & \mathsf{req}_{2t}^1¢\$ \\
\vdash_{M_{n+1},\mathcal{M}'} \ & \mathsf{rec}_{2t}^1¢\$ \\
\vdash_{M_{n+1},\mathcal{M}'} \ & \mathsf{req}_{accept}^1¢\$ \\
\vdash_{M_{n+1},\mathcal{M}'} \ & \mathsf{rec}_{accept}^1¢\$ \\
\vdash_{M_{n+1},\mathcal{M}'} \ & \mathsf{Accept}.
\end{aligned}$$

The number of communications for $l = 0$ is $2t+1$. For an input of length l, M_{n+1} performs l cycles and a tail computation. Thus, it accepts after exactly

$$\left(\sum_{j=1}^{l} 2t(j+1)\right) + 2t + 1 = 2t\frac{l^2+3l}{2} + 2t + 1 = t\frac{2l^2+6l+4}{2} + 1 > r$$

communications. Whenever M_1' halts (accepting or not), then M_{n+1} will get stuck, because of a missing answer. On the other hand, if M_1' is in a computation loop, then M_{n+1} reaches the accepting configuration. All in all, $\mathcal{M}'$ accepts exactly $\overline{L}$. Since M_{n+1} is a det-R-component, $\mathcal{M}'$ is of the same type as $\mathcal{M}$ and thus, $\overline{L} \in \mathcal{L}(\mathsf{det\text{-}PC\text{-}X})$. This completes the proof that the classes of languages that are accepted by any type of deterministic PCRA systems are closed under complement. □

Theorem 16. *The language classes $\mathcal{L}((\mathsf{det\text{-}})\mathsf{PC\text{-}X})$ are closed under marked product for any $\mathsf{X} \in \mathcal{T}$.*

Proof. Let $L_1, L_2 \in \mathcal{L}((\mathsf{det\text{-}})\mathsf{PC\text{-}X})$ for any $\mathsf{X} \in \mathcal{T}$. Then there exist two centralized systems $\mathcal{M} = (M_1, M_2, \ldots, M_m)$ and $\mathcal{N} = (N_1, N_2, \ldots, N_n)$ with $L(\mathcal{M}) = L_1$ and $L(\mathcal{N}) = L_2$. Further, let

$$M_i = (Q_{M_i}, \Sigma_M, \Gamma_{M_i}, ¢, \$, q_0^{(M_i)}, k_M, \delta_{M_i})$$

for all $1 \leq i \leq m$ and

$$N_j = (Q_{N_j}, \Sigma_N, \Gamma_{N_j}, ¢, \$, q_0^{(N_j)}, k_N, \delta_{N_j})$$

for all $1 \leq j \leq n$. W.l.o.g. we may assume that transitions of the form $\mathsf{MVL} \in \delta(p, ¢\alpha)$ are not allowed. We construct a system $\mathcal{C} = (M_1', M_2', \ldots, M_m', N_1', N_2', \ldots, N_n')$ with $L(\mathcal{C}) = \{w_1\#w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$, where # is the middle marker that is not contained in the tape alphabets of the components of $\mathcal{M}$ and $\mathcal{N}$. The main idea of this proof is to modify the transition relations of the components of $\mathcal{M}$ and $\mathcal{N}$ such that $M_1', \ldots, M_m'$ ignore the part of the tape to the right of the #-symbol, and $N'_1, \ldots, N'_n$ ignore the part of the tape to the left of #. Thus, the #-symbol is interpreted as \$, ¢ respectively. Formally, $M'_i := (Q_{M_i}, \Sigma, \Gamma_{M'_i}, ¢, \$, q_0^{(M_i)}, k_M, \delta_{M'_i})$, $1 \leq i \leq m$, and $N'_j := (Q_{N_j} \cup \{q_0^{(N'_j)}\}, \Sigma, \Gamma_{N'_j}, ¢, \$, q_0^{(N'_j)}, k_N, \delta_{N'_j})$, $1 \leq j \leq n$, are defined as follows:

- $\Sigma := \Sigma_M \cup \{\#\} \cup \Sigma_N$, $\Gamma_{M'_i} := \Gamma_{M_i} \cup \{\#\} \cup \Sigma_N$, $\Gamma_{N'_j} := \Sigma_M \cup \{\#\} \cup \Gamma_{N_j}$.
- If $A \in \delta_{M_i}(q, \alpha)$ and α does not end with \$, then $A \in \delta_{M'_i}(q, \alpha)$.

  If $A \in \delta_{M_i}(q, \alpha\$)$ and A is not a rewrite operation, then $A \in \delta_{M'_i}(q, \alpha\#\gamma)$ for all $\gamma \in \Sigma_N^{k_M-|\alpha|-1} \cup (\Sigma_N^{\leq(k_M-|\alpha|-2)} \cdot \{\$\})$.

  If $(p, \beta\$) \in \delta_{M_i}(q, \alpha\$)$, then $(p, \beta\#\gamma) \in \delta_{M'_i}(q, \alpha\#\gamma)$ for all $\gamma \in \Sigma_N^{k_M-|\alpha|-1} \cup \Sigma_N^{\leq(k_M-|\alpha|-2)} \cdot \{\$\}$.
- $\delta_{N'_j}(q_0^{(N'_j)}, a\alpha) = \{(q_0^{(N'_j)}, \mathsf{MVR})\}$ for all $a \in \{¢\} \cup \Sigma_M$ and $\alpha \in \Sigma_M^* \cdot ((\{\#\} \cdot \Gamma_{N_j}^*) \cup \{\varepsilon\})$ with $|\alpha| = k_N - 1$ or $\alpha \in \Sigma_M^* \cdot \{\#\} \cdot \Gamma_{N_j}^* \cdot \{\$\}$ with $|\alpha| \leq k_N - 1$.

  $\delta_{N'_j}(q_0^{(N'_j)}, \#\alpha) = \{q_0^{(N_j)}\}$ for all $\alpha \in \Gamma_{N_j}^{k_N-1} \cup (\Gamma_{N_j}^{\leq(k_N-2)} \cdot \{\$\})$.

  If $A \in \delta_{N_j}(q, \alpha)$ and α does not start with ¢, then $A \in \delta_{N'_j}(q, \alpha)$.

  If $A \in \delta_{N_j}(q, ¢\alpha)$ and A is not a rewrite operation, then $A \in \delta_{N'_j}(q, \#\alpha)$.

  If $(p, ¢\beta) \in \delta_{N_j}(q, ¢\alpha)$, then $(p, \#\beta) \in \delta_{N'_j}(q, \#\alpha)$.

With this modifications we obtain that, for all $1 \leq i \leq m$ and $1 \leq j \leq n$,

$$q_0^{(M_i)}¢v\$ \vdash_{M_i,\mathcal{M}}^* v_1qv_2 \quad \Leftrightarrow \quad q_0^{(M_i)}¢v\#w\$ \vdash_{M'_i,\mathcal{C}}^* v_1qv'_2w\$$$

and

$$q_0^{(N_j)}¢w\$ \vdash_{N_j,\mathcal{N}}^* w_1qw_2 \quad \Leftrightarrow \quad q_0^{(N'_j)}¢v\#w\$ \vdash_{N'_j,\mathcal{C}}^* ¢vq_0^{(N_j)}\#w\$ \vdash_{N'_j,\mathcal{C}}^* ¢vw'_1qw'_2,$$

where v'_2 arises from v_2 by replacing \$ with # and w'_1 and w'_2 arise from w_1, w_2 respectively, by replacing ¢ with # (w_2 contains ¢ if $w_1 = \varepsilon$). Since the system $\mathcal{C}$ should only accept, if $\mathcal{M}$ and $\mathcal{N}$ accept, two more modifications are necessary concerning the master components M_1 and N_1:

- Replace all transitions of the form $\mathsf{Accept} \in \delta_{M'_1}(q, \alpha)$ by the transitions $\mathsf{req}_{Accept}^{N'_1} \in \delta_{M'_1}(q, \alpha)$ and $\mathsf{Accept} \in \delta_{M'_1}(\mathsf{rec}_{Accept,Accept}^{N'_1}, \alpha)$ in M'_1. Add the new communication states $\mathsf{req}_{Accept}^{N'_1}$ and $\mathsf{rec}_{Accept,Accept}^{N'_1}$ to the set of states of M'_1.
- Replace all transitions of the form $\mathsf{Accept} \in \delta_{N'_1}(q, \alpha)$ by the transition $\mathsf{res}_{Accept}^{M'_1} \in \delta_{N'_1}(q, \alpha)$ in N'_1 and add the new communication states $\mathsf{res}_{Accept}^{M'_1}$

and $\mathsf{ack}^{M_1'}_{Accept}$ to the set of states of N_1', where the subscript *Accept* denotes the message that is sent from N_1' to M_1'.

Thus, the System $\mathcal{C}$ accepts an input $v\#w$ if and only if $\mathcal{M}$ accepts the input v and $\mathcal{N}$ accepts the input w. Moreover, the construction is strictly deterministic. If $\mathcal{M}$ and $\mathcal{N}$ are deterministic, then so is $\mathcal{C}$. Thus, $L(\mathcal{C}) = L_1{\cdot}\{\#\}{\cdot}L_2 \in \mathcal{L}((\mathsf{det\text{-}})\mathsf{PC\text{-}X})$.□

From [JLNO04] we know, that $\mathcal{L}(\mathsf{RWW})$ and $\mathcal{L}(\mathsf{RRWW})$ are closed under product. Now we extend this result to PCRA systems of these types and to PC-RLWW-systems.

Theorem 17. *The language classes $\mathcal{L}$(PC-RWW), $\mathcal{L}$(PC-RRWW), and $\mathcal{L}$(PC-RLWW) are closed under product.*

Proof. Let $L_1, L_2 \in \mathcal{L}(\mathsf{PC\text{-}X})$ for $\mathsf{X} \in \{\mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$, and let $\mathcal{M} = (M_1, M_2, \ldots, M_m)$ and $\mathcal{N} = (N_1, N_2, \ldots, N_n)$ be PC-X-systems with $L(\mathcal{M}) = L_1$ and $L(\mathcal{N}) = L_2$. We can assume that $\mathcal{M}$ and $\mathcal{N}$ are centralized systems that accept an input if and only if their first components M_1 and N_1 reach the accepting configuration (due to Corollary 4). Now, we construct a system $\mathcal{C} = (M_1', M_2', \ldots, M_m', N_1', N_2', \ldots, N_n')$ that consists of modified components of $\mathcal{M}$ and $\mathcal{N}$ and that accepts the language $L = L_1 \cdot L_2$. The main idea of this proof is to choose nondeterministically a factorization of the input $w = uv$, store this factorization by writing a marker on the tape, and then check whether u is contained in L_1 and v is contained in L_2. The latter can be done quite similarly to the proof of the closure under the marked product (see Theorem 16 for further details).

The behaviour of $\mathcal{C}$ can be described as follows. In the inital configuration the input w is on the tapes of all components. Now, $\mathcal{C}$ works in two phases: (1) guess the factorization and write the marker # (that is a new tape symbol for all components) on all tapes, and (2) verify the chosen factorization.

A basic requirement is that all components work on the same factorization. Therefore, a dedicated component determines the factorization and instructs the other components by communication. Let M_1' be this component. First, it chooses nondeterministically one of the following cases, where $w = uv$ is the factorization that is to be computed.

Case 1 ($|u| = 0$). If the first factor is ε, then all M_i', $1 \leq i \leq m$, empty their tapes, and then they change into the simulation phase. All N_j', $1 \leq j \leq n$, keep their current tape contents.

Case 2 ($|u| = 1$). If the first factor has length one, then all M_i', $1 \leq i \leq m$, remove all symbols of the tape except the first symbol, i.e. the symbol directly to the right of the ¢-symbol. The components N_j', $1 \leq j \leq n$, remove only their first symbols.

Case 3 ($|u| \geq 2$ and $|v| = 0$). In this case, the components M_i', $1 \leq i \leq m$, keep

their current tape contents, and all N'_j, $1 \leq j \leq n$, empty their tapes.
Case 4 ($|u| \geq 2$ and $|v| = 1$). In this case, the components M'_i, $1 \leq i \leq m$, remove the last tape symbol, i.e. the symbol directly to the left of the \$-symbol, and all N'_j, $1 \leq j \leq n$, remove all tape symbols except the last one.
Case 5 ($|u| \geq 2$ and $|v| \geq 2$). If M'_1 decides that both factors have a length of at least two, then the marker # is written on all tapes as follows. All M'_i, $1 \leq i \leq m$, move their windows three steps to the right, and all N'_j, $1 \leq j \leq n$, move their windows one step to the right. Then, the windows of all components are moved simultaneously to the right. At some point in time, all components write the marker # on the tape and change into the simulation phase. If the windows are moved too many steps to the right, i.e. \$ appears in the window, and the marker cannot be written, then the system gets stuck. Otherwise, for an input w and a factorization $w = uv = u_1 \dots u_{l_1} v_1 \dots v_{l_2}$, the tape inscriptions of all M'_i are $\text{¢}u\#v_3 \dots v_{l_2}\$$, and those of all N'_j are $\text{¢}u_1 \dots u_{l_1-2}\#v\$$. This means that all M'_i use the first two symbols of the second factor to write the marker, and all N'_j use the last two symbols of the first factor. Thus, the first factor is completely on the tapes of all M'_i, and the second factor is completely on the tapes of all N'_j.

For writing the marker and preparing the tapes in the first phase, a window size of two is necessary for all components. After determining the factorization, all components change into dedicated states to begin the simulation phase. If the factorization corresponds to one of the cases 1 to 4, the components M'_i and N'_j behave exactly like the original components M_i and N_j in the simulation. In case 5, the components verify the corresponding factors as shown in the proof of Theorem 16. Since the input $w = uv \in L_1 \cdot L_2$ if $u \in L_1$ and $v \in L_2$, the system $\mathcal{C}$ must accept w if both systems $\mathcal{M}$ and $\mathcal{N}$ accept the corresponding factors. As we have assumed that $\mathcal{M}$ and $\mathcal{N}$ accept with their first components, so $\mathcal{C}$ has to accept w if M_1 accepts u and N_1 accepts v. For this, if M_1 would accept u, then M'_1 communicates with N'_1 and if N_1 would accept v, then N'_1 answers to M'_1 by sending a corresponding message, and afterwards, M'_1 accepts. Hence, $\mathcal{C}$ accepts an input w if there exists a factorization $w = uv$ such that $\mathcal{M}$ accepts u and $\mathcal{N}$ accepts v.

On the other hand, if such a factorization does not exist, then M_1 or N_1 (or both) do not accept the corresponding factor, and thus, M'_1 does not accept w for any possible factorization. Observe that no component other than M'_1 can reach the accepting configuration. Therefore, it follows that $L = L(\mathcal{C}) = L(\mathcal{M}) \cdot L(\mathcal{N}) = L_1 \cdot L_2$. Moreover, since each component restarts immediately after removing tape symbols or writing the marker in the first phase, we can state that, if $\mathcal{M}$ and $\mathcal{N}$ are of type PC-X, then $\mathcal{C}$ is of the same type. Hence, $L \in \mathcal{L}(\text{PC-X})$. □

Theorem 18. *The language classes $\mathcal{L}$(PC-RWW), $\mathcal{L}$(PC-RRWW), and $\mathcal{L}$(PC-RLWW) are closed under Kleene closure.*

Proof. Let $L \in \mathcal{L}(\textsf{PC-RLWW})$, and let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PC-RLWW-system with $L(\mathcal{M}) = L$. For the language classes $\mathcal{L}$(PC-RWW) and $\mathcal{L}$(PC-RRWW) the proof is similar. We can assume that $\mathcal{M}$ is a centralized system that accepts if and only if M_1 accepts (due to Corollary 4). We now construct a system $\mathcal{M}' = (M_1', M_2', \ldots, M_n')$ that accepts the language L^*. The main idea is to divide an input w nondeterministically into a factorization $w = w_1 w_2 \ldots w_r$ (w_1, w_2, $\ldots$, w_r are words over the input alphabet) and check whether each factor is contained in L. If such a factorization exists, then $w \in L^*$, otherwise $w \notin L^*$. For obtaining the factorization, sequentially one factor of a nondeterministically chosen length is separated from the left-hand side of the tape content and is then checked by simulating $\mathcal{M}$ on this factor by the modified components $M_1', M_2', \ldots, M_n'$.

The behaviour of the system $\mathcal{M}'$ can be described as follows. At the beginning of a computation, an input word w is on the tapes of all components. If the input is ε, then $\mathcal{M}'$ accepts immediately, since ε is in the Kleene closure of every language. Otherwise, the computation consists of three phases that can be repeated several times in the given order: 1) write a marker on the tape to separate a new factor, 2) check whether the chosen factor is contained in L, and 3) remove the remaining symbols of the last verified factor. The number of iterations of the three phases depends on the nondeterministically chosen number of factors of w. The computation and the progress of the phases are shown in Figure 5.3 for an arbitrary component M_i', where we assume a factorization $w = w_1 w_2 \ldots w_r$ and the form $w_j = w_{j,1} w_{j,2} \ldots w_{j,s_j}$ for all factors w_j, $1 \leq j \leq r$. Observe that the factorization is not fixed at the beginning of the computation (although Figure 5.3 may seem to imply this), but it is determined successively by repeating the three phases above. The words $w_1', w_2', \ldots, w_r'$ in Figure 5.3 are used to denote the remaining symbols after the simulation in the second phase on the words $w_1, w_2, \ldots, w_r$ was performed. Each of the symbols $w_{1,1}', w_{1,s_1}', w_{2,1}', w_{2,s_2}', \ldots, w_{r,1}', w_{r,s_r}'$ is either $w_{1,1}, w_{1,s_1}, w_{2,1}, w_{2,s_2}, \ldots, w_{r,1}, w_{r,s_r}$, if the corresponding stored symbol was not removed during the simulation in phase two, or ε if it was removed. This will be illustrated in more detail in the following explanations of the different phases.

Phase 1. In this phase M_1' first decides nondeterministically whether the whole (remaining) tape content is interpreted as the last factor of the factorization, or whether a new factor is separated. In the former case M_1' informs all the other components by sending a corresponding message, and the system continues with the second phase. Otherwise, all components move their windows to the right simultaneously. This can be controlled by M_1' through a communication between

$$\begin{array}{llll}
 & ¢w_{1,1}w_{1,2}...w_{1,s_1}w_{2,1}w_{2,2}...w_{2,s_2}w_3w_4...w_r\$ & [-,-,-] & (1)\\
\text{Phase 1} & \downarrow & & \\
 & ¢w_{1,1}w_{1,2}...w_{1,s_1-1}\#w_{2,2}...w_{2,s_2}w_3w_4...w_r\$ & [-,w_{1,s_1},w_{2,1}] & (2)\\
\text{Phase 2} & \downarrow & & \\
 & ¢w'_1\#w_{2,2}...w_{2,s_2}w_3w_4...w_r\$ & [-,w'_{1,s_1},w_{2,1}] & (3)\\
\text{Phase 3} & \downarrow & & \\
 & ¢\#w_{2,2}...w_{2,s_2}w_3w_4...w_r\$ & [w_{2,1},-,-] & (4)\\
\text{Phase 1} & \downarrow & & \\
 & ¢\#w_{2,2}...w_{2,s_2-1}\#w_{3,2}...w_{3,s_3}w_4...w_r\$ & [w_{2,1},w_{2,s_2},w_{3,1}] & (5)\\
\text{Phase 2} & \downarrow & & \\
 & ¢\#w'_2\#w_{3,2}...w_{3,s_3}w_4...w_r\$ & [w'_{2,1},w'_{2,s_2},w_{3,1}] & (6)\\
\text{Phase 3} & \downarrow & & \\
 & ¢\#w_{3,2}...w_{3,s_3}w_4...w_r\$ & [w_{3,1},-,-] & (7)\\
\text{Phase 1} & \downarrow & & \\
 & ¢\#w_{3,2}...w_{3,s_3-1}\#w_{4,2}...w_{4,s_4}w_5...w_r\$ & [w_{3,1},w_{3,s_3},w_{4,1}] & (8)\\
 & \downarrow & & \\
 & ... & & (9)\\
 & \downarrow & & \\
 & ¢\#w_{r-1,2}...w_{r-1,s_{r-1}-1}\#w_{r,2}...w_{r,s_r}\$ & [w_{r-1,1},w_{r-1,s_{r-1}},w_{r,1}] & (10)\\
\text{Phase 2} & \downarrow & & \\
 & ¢\#w'_{r-1}\#w_{r,2}...w_{r,s_r}\$ & [w'_{r-1,1},w'_{r-1,s_{r-1}},w_{r,1}] & (11)\\
\text{Phase 3} & \downarrow & & \\
 & ¢\#w_{r,2}...w_{r,s_r}\$ & [w_{r,1},-,-] & (12)\\
\text{Phase 1} & \downarrow & & \\
 & ¢\#w_{r,2}...w_{r,s_r}\$ & [w_{r,1},-,-] & (13)\\
\text{Phase 2} & \downarrow & & \\
 & ¢\#w'_r\$ & [w'_{r,1},-,-] & (14)\\
\text{Phase 3} & \downarrow & & \\
 & ¢\$ & [-,-,-] & (15)
\end{array}$$

Figure 5.3: The progress of the phases for an arbitrary component M'_i.

M'_1 and all other components before each local computation step. When M'_1 nondeterministically decides that the last symbol of the new factor is reached, it sends an appropriate message to the other components, all components write the marker # (that is a new tape symbol for all components) at the current position, and thereafter they perform a restart immediately (see lines 2, 5, 8, and 10 of Figure 5.3). For writing the marker the two tape symbols at the current position have to be replaced. In order to not forget the information about these symbols, they are stored and carried within the states during the whole computation (the nonforgetting property and the local information of the communication states can be used for this). Except for the first run or when checking a factor of length one (see below), two markers are on the tape within the second phase: the marker of the previous factor and the currently written one. This fact is important, as the markers determine the positions of the replaced tape symbols, and these symbols are needed for the simulation of the second phase. To store the rewritten symbols, annotations to the states are used that have the form $[a, b, c]$, where a is the first symbol of the currently considered factor combined with the first marker, b is the last symbol of the currently considered factor, and c is the first symbol of the following factor. The symbols b and c have been replaced by writing the second marker. When instead of a, b, or c, a minus is used within the triple, then the corresponding marker was not written. For example, at the beginning of phase 1, b and c are always '$-$', because the second marker has still to be written. Using '$-$' is different from using ε, which means that the corresponding marker exists, but the corresponding symbol was removed during the simulation within the second phase.

Look for example at line 5 of Figure 5.3. This situation occurs between finishing phase 1 and the beginning of phase 2. Two markers are currently on the tape, the first one directly next to the ¢-symbol and the second one between the symbols w_{2,s_2-1} and $w_{3,2}$. The first symbol of w_2, which is $w_{2,1}$, was replaced by the first marker in line 2 and is still stored as the first entry of the triple in line 5. In contrast, the last symbol of w_2, which is w_{2,s_2}, and the first symbol of w_3, which is $w_{3,1}$, were replaced by the second marker in the previous first phase and thus, they are stored as the second and the third entry in the triple of line 5.

Since for each factor one marker is set, whereby two symbols are replaced, factors of length one have to be treated in a special way. Thus, if M_1' nondeterministically chooses to separate a factor of length one, then two cases have to be distinguished. First, if no marker is on the tape (this situation can be detected through the triple $[-,-,-]$ as in line 1), then a marker is written directly next to the ¢-symbol and the replaced symbols are stored in the second and the third position of the triple (the one and only symbol of the current factor as second entry and the first letter of the following factor as the third entry). Second, if there is already a marker on the tape, then it must already be positioned directly next to the ¢-symbol (see line 4), so this situation can be detected immediately (the annotation has the form $[a,-,-]$), and no other marker will be written. Both situations are demonstrated in Figure 5.4. For factors of length one, M_1' sends a corresponding message to all other components, thus the situation being in state $q[-,a,b]$ (first situation) or $q[a,-,-]$ (second situation) with $¢\#\ldots\$$ on the tape is then treated in the second phase as having $¢a\$$ on the tape.

Table 5.1 summarizes all situations that can appear at the end of phase 1 and the beginning of phase 2.

Phase 2. After setting a marker or deciding whether the remaining tape content is the last factor or whether a factor of length one has to be checked, $\mathcal{M}'$ checks whether the factor is contained in L. For this purpose it simulates $\mathcal{M}$, and the components interpret the stored symbols and the marker as described above. Moreover, the windows of all components may never be moved across the (second)

$$\text{Situation 1:}\quad \begin{array}{c} ¢abc\ldots\$ \\ [-,-,-] \end{array} \xrightarrow{\text{Phase 1}} \begin{array}{c} ¢\#c\ldots\$ \\ [-,a,b] \end{array} \xrightarrow{\text{Phase 2}} \begin{array}{c} ¢\#c\ldots\$ \\ [-,a',b] \end{array} \xrightarrow{\text{Phase 3}} \begin{array}{c} ¢\#c\ldots\$ \\ [b,-,-] \end{array}$$

$$\text{Situation 2:}\quad \begin{array}{c} ¢\#\ldots\$ \\ [a,-,-] \end{array} \xrightarrow{\text{Phase 1}} \begin{array}{c} ¢\#\ldots\$ \\ [a,-,-] \end{array} \xrightarrow{\text{Phase 2}} \begin{array}{c} ¢\#\ldots\$ \\ [a',-,-] \end{array} \xrightarrow{\text{Phase 3}} \begin{array}{c} ¢\ldots\$ \\ [-,-,-] \end{array}$$

Figure 5.4: Choosing a factor of length one.

Situation (tape content and state annotation)	interpretation of the first #-symbol	interpretation of the second #-symbol
1) $¢\#c\ldots\$$, $[-,a,b]$	(no first marker)	$a\$$
2) $¢\#\ldots\$$, $[a,-,-]$	$a\$$	(no second marker)
3) $¢w_{i,1}w_{i,2}\ldots w_{i,s_i-1}\#w_{i+1,2}\ldots\$$, $[-,w_{i,s_i},w_{i+1,1}]$	(no first marker)	$w_{i,s_i}\$$
4) $¢\#w_{i,2}w_{i,3}\ldots w_{i,s_i-1}\#w_{i+1,2}\ldots\$$, $[w_{i,1},w_{i,s_i},w_{i+1,1}]$	$w_{i,1}$	$w_{i,s_i}\$$

Table 5.1: Situations after the first phase.

#-symbol. More precisely, the second #-symbol is not allowed to appear as the first symbol of the windows (only at the second or higher position of the windows). This is necessary in order to enable a rewrite step and a restart operation without deleting the # or a symbol to the right of it. But this is no problem due to the look ahead of at least one tape cell (with a window size of at least two).

Two critical situations can appear during the simulation of $\mathcal{M}$. First, a component M_i, and thus M_i', $(2 \leq i \leq n)$ reaches an infinite computation that is caused by a combination of MVL and MVR steps. Second, a component M_i, and thus M_i', $(2 \leq i \leq n)$ gets stuck in an unanswered communication state. Consider a scenario where $w_1, w_2 \in L$, $u \in L^*$ with $u = w_1 w_2$, and there is no other factorization of u such that all factors are containted in L. Moreover, assume that there is exactly one accepting computation of $\mathcal{M}$ on input w_1 and w_2, respectively, and that there exists a component M_i that reaches one of the two above described critical situations during the computation of $\mathcal{M}$ on w_1 and that further executes a communication with M_1 within the accepting computation of $\mathcal{M}$ on w_2. Although $\mathcal{M}$ is correctly simulated by $\mathcal{M}'$ on the first factor, it cannot verify the factor w_2, since M_i' is in a critical situation and thus not available anymore. Hence, $\mathcal{M}'$

cannot accept u, although $u \in L^*$.

To avoid these situations a communication protocol is used that allows M_1' to control the whole computation of $\mathcal{M}'$ such that each communication is strictly predetermined and, in addition, no arbitrary long local computations can be executed. For that we use a similar but simplified version of the construction from the proof of Theorem 4. In the following the main idea is described, where the annotations in brackets refer to the construction below.

To control the computation of $\mathcal{M}'$, the master component M_1' not only simulates M_1, but communicates cyclically with all components that are still involved in the simulation of $\mathcal{M}$ before each simulated local computation step. In these communications, M_1' asks the components about the transitions they will apply in the next simulation step (A1.1). Each M_i' ($2 \leq i \leq n$) communicates with M_1' before each simulation step and tells it whether the next simulation step is a local step (B1), a communication (B2 and B3), or that no transition can be applied (B5) (observe that M_1' needs to expect only finitely many possible answers of M_i' depending on the transition mapping of M_i). In the case of a communication, it sends information about the corresponding communication state. Thereafter, the component requests further instructions from M_1'. Now, if the answer of M_i' is 'local step', then M_1' just allows M_i' to continue its computation (A1.2), and M_i' performs the corresponding simulation step. In the case that M_i' answers with information about a communication step (this can only be a communication with M_1, since $\mathcal{M}$ is centralized) or an undefined situation, then M_1' stores this information using t_i in the subscript of its communication states and asks the next available component M_m' with $t_m = *$ (A1.3). After all components have been asked and the answer of the last component was computed (A1.4), M_1' simulates the stored computation step (A1.5). Whenever M_1' wants to simulate a communication step with a component M_i' (A is a communication state), then three situations are possible: First, M_i' has not reached any communication state to simulate ($t_i = *$). In this case, M_1' just remembers A and repeats the communication cycle. Second, M_i' has already reached the corresponding communication state ($t_i = req$ or $t_i = res(c)$). Then, M_1' informs M_i' that the communication can be successfully simulated. Third, M_i' is stuck ($t_i =\perp$) or it has reached a communication state that does not correspond to that of M_1' (e.g. both communication states are request states or response states). In the third case, M_1' gets stuck as well as M_1 would in this situation, and thus, the current factor is not successfully verified.

If M_1 would reach the accepting configuration, then M_1' sends the message 'stop' to all components to finish the simulation in the second phase and to start with the third phase immediately. In the construction below this is indicated by reaching q_{phase3} as the initial state of the third phase.

Now, the modifications for the components are defined that are necessary for implementing the described communication protocol. Here, $\hat{\delta}_i$ denotes the transition mapping of component $\hat{M}_i$ that results from M_i by applying the modifications of the first phase and interpreting the marker and the stored tape symbols as described above. Moreover, we assume that M_1' starts the second phase always in state $q_{\langle *,\ldots,*\rangle}$, where q is the initial state of M_1. The construction steps indexed by A refer to the master component $\hat{M}_1$ and those indexed by B refer to all the other components.

A1) For all $A \in \hat{\delta}_1(q,\alpha)$ with $A \neq \mathsf{Accept}$ and all situations $\langle t_2,\ldots,t_n\rangle$ with $t_j \in \{*,\bot,req\} \cup \{res(c) \mid c \text{ is a message of any } q \in \mathsf{COM}(M_j)\}$, $2 \leq j \leq n$, define:

A1.1) Ask the first available component:

$$\mathsf{req}^m_{A\langle t_2,\ldots,t_n\rangle} \in \delta_1'(q_{\langle t_2,\ldots,t_n\rangle},\alpha) \text{ such that } t_m = * \text{ and for all } j \in \{2,\ldots,m-1\} : t_j \neq *.$$

A1.2) Answer in the case of 'local step':

$$\delta_1'(\mathsf{rec}^i_{A\langle t_2,\ldots,t_n\rangle,\mathsf{localstep}},\alpha) = \{\mathsf{res}^i_{A\langle t_2,\ldots,t_n\rangle,\mathsf{continue}}\} \text{ for all } 2 \leq i \leq n.$$

A1.3) Store the information in the cases of communication and undefined situation and ask the next available component:

$$\begin{aligned}\delta_1'(\mathsf{ack}^i_{A\langle t_2,\ldots,t_n\rangle,\mathsf{continue}},\alpha) &= \{\mathsf{req}^m_{A\langle t_2,\ldots,t_n\rangle}\},\\ \delta_1'(\mathsf{rec}^i_{A\langle t_2,\ldots,t_n\rangle,answer},\alpha) &= \{\mathsf{req}^m_{A\langle t_2,\ldots,t_{i-1},answer,t_{i+1},\ldots,t_n\rangle}\},\end{aligned}$$

with $answer \in \{\bot,req\}\cup\{res(c) \mid c \text{ is a message of any } q \in \mathsf{COM}(M_i)\}$, for all $2 \leq i \leq n$, and with $i < m$, $t_m = *$, and for all $j \in \{i+1,\ldots,m-1\}$: $t_j \neq *$.

A1.4) Get the answer from the last available component:

$$\begin{aligned}\delta_1'(\mathsf{ack}^m_{A\langle t_2,\ldots,t_n\rangle,\mathsf{continue}},\alpha) &= \{\mathsf{sim}_{A\langle t_2,\ldots,t_n\rangle}\},\\ \delta_1'(\mathsf{rec}^m_{A\langle t_2,\ldots,t_n\rangle,answer},\alpha) &= \{\mathsf{sim}_{A\langle t_2,\ldots,t_{i-1},answer,t_{i+1},\ldots,t_n\rangle}\},\end{aligned}$$

with $answer \in \{\bot,req\}\cup\{res(c) \mid c \text{ is a message of any } q \in \mathsf{COM}(M_m)\}$ and for all $j \in \{m+1,\ldots,n\}$: $t_j \neq *$.

A1.5) Simulate the transition of M_1:

$$\delta'_1(\mathsf{sim}_{A\langle t_2,\ldots,t_n\rangle},\alpha)=\begin{cases}\{(p_{\langle t_2,\ldots,t_n\rangle},\mathsf{MVR})\}, & \text{if } A=(p,\mathsf{MVR}),\\ \{(p_{\langle t_2,\ldots,t_n\rangle},\mathsf{MVL})\}, & \text{if } A=(p,\mathsf{MVL}),\\ \{(p_{\langle t_2,\ldots,t_n\rangle},\beta)\}, & \text{if } A=(p,\beta),\\ \{\mathsf{req}^m_{A\langle t_2,\ldots,t_n\rangle}\}, & \text{if } A=\mathsf{req}^i_d \text{ or } A=\mathsf{res}^i_{d,c}\\ & \text{and } t_i=*,\\ \{\mathsf{res}^i_{A\langle t_2,\ldots,t_n\rangle,c}\}, & \text{if } A=\mathsf{req}^i_d \text{ and } t_i=res(c)\\ & \text{or } A=\mathsf{res}^i_{d,c} \text{ and } t_i=req,\end{cases}$$

s.t. for all $j\in\{1,\ldots,m-1\}$: $t_j\neq *$.

In the last case the communication between M_1 and M_i is simulated, a new transition of M_1 is chosen, and the first available component is asked again:

$$\delta'_1(\mathsf{ack}^i_{A\langle t_2,\ldots,t_n\rangle,c},\alpha)=\begin{cases}\{\mathsf{req}^m_{A'\langle t_2,\ldots,t_{i-1},*,t_{i+1},\ldots,t_n\rangle}\mid A'\in\delta_1(\mathsf{rec}^i_{d,c},\alpha)\},\\ \quad\text{if } \mathsf{Accept}\notin\delta_1(\mathsf{rec}^i_{d,c},\alpha), A=\mathsf{req}^i_d, \text{and } t_i=res(c),\\ \{\mathsf{req}^m_{A'\langle t_2,\ldots,t_{i-1},*,t_{i+1},\ldots,t_n\rangle}\mid A'\in\delta_1(\mathsf{ack}^i_{d,c},\alpha)\},\\ \quad\text{if } \mathsf{Accept}\notin\delta_1(\mathsf{ack}^i_{d,c},\alpha), A=\mathsf{res}^i_{d,c}, \text{and } t_i=req,\\ \{\mathsf{res}^2_{-,\mathsf{stop}}\},\\ \quad\text{if } \mathsf{Accept}\in\delta_1(\mathsf{rec}^i_{d,c},\alpha), A=\mathsf{req}^i_d, \text{and } t_i=res(c),\\ \{\mathsf{res}^2_{-,\mathsf{stop}}\},\\ \quad\text{if } \mathsf{Accept}\in\delta_1(\mathsf{ack}^i_{d,c},\alpha), A=\mathsf{res}^i_{d,c}, \text{and } t_i=req,\end{cases}$$

s.t. for all $j\in\{1,\ldots,m-1\}$: $t_j\neq *$.

A2) For all $\mathsf{Accept}\in\hat{\delta}_1(q,\alpha)$ and all possible situations $\langle t_2,\ldots,t_n\rangle$ define:

$$\begin{aligned}\delta'_1(q_{\langle t_2,\ldots,t_n\rangle},\alpha)&=\{\mathsf{res}^2_{-,\mathsf{stop}}\},\\ \delta'_1(\mathsf{ack}^i_{-,\mathsf{stop}},\alpha)&=\{\mathsf{res}^{i+1}_{-,\mathsf{stop}}\}\text{ for all } 2\le i\le n-1,\\ \delta'_1(\mathsf{ack}^n_{-,\mathsf{stop}},\alpha)&=\{q_{phase3}\}.\end{aligned}$$

B1) For all $A\in\hat{\delta}_i(q,\alpha)$ and $A\notin\mathsf{COM}(M_i)$ define:

$$\begin{aligned}\mathsf{res}^1_{A,\mathsf{localstep}}&\in\delta'_i(q,\alpha),\\ \delta'_i(\mathsf{ack}^1_{A,\mathsf{localstep}},\alpha)&=\{\mathsf{req}^1_A\},\\ \delta'_i(\mathsf{rec}^1_{A,\mathsf{continue}},\alpha)&=\{A\},\\ \delta'_i(\mathsf{rec}^1_{A,\mathsf{stop}},\alpha)&=\{q_{phase3}\}.\end{aligned}$$

B2) For all $\mathsf{req}^1_d \in \hat{\delta}_i(q,\alpha)$ and $q \notin \mathsf{COM}(M_i)$ define:

$$\begin{array}{rcl}
\mathsf{res}^1_{req(d),req} & \in & \delta'_i(q,\alpha), \\
\delta'_i(\mathsf{ack}^1_{req(d),req},\alpha) & = & \{\mathsf{req}^1_{req(d)}\}, \\
\delta'_i(\mathsf{rec}^1_{req(d),c},\alpha) & = & \delta_i(\mathsf{rec}^1_{d,c},\alpha) \setminus \mathsf{COM}(M_i) \\
 & & \cup\, \{\mathsf{res}^1_{req(d'),req} \mid \mathsf{req}^1_{d'} \in \delta_i(\mathsf{rec}^1_{d,c},\alpha)\} \\
 & & \cup\, \{\mathsf{res}^1_{res(d',c'),res(c')} \mid \mathsf{res}^1_{d',c'} \in \delta_i(\mathsf{rec}^1_{d,c},\alpha)\}, \\
\delta'_i(\mathsf{rec}^1_{req(d),\mathsf{stop}},\alpha) & = & \{q_{phase3}\}.
\end{array}$$

B3) For all $\mathsf{res}^1_{d,c} \in \hat{\delta}_i(q,\alpha)$ and $q \notin \mathsf{COM}(M_i)$ define:

$$\begin{array}{rcl}
\mathsf{res}^1_{res(d,c),res(c)} & \in & \delta'_i(q,\alpha), \\
\delta'_i(\mathsf{ack}^1_{res(d,c),res(c)},\alpha) & = & \{\mathsf{req}^1_{res(d,c)}\}, \\
\delta'_i(\mathsf{rec}^1_{res(d,c),c},\alpha) & = & \delta_i(\mathsf{ack}^1_{d,c},\alpha) \setminus \mathsf{COM}(M_i) \\
 & & \cup\, \{\mathsf{res}^1_{req(d'),req} \mid \mathsf{req}^1_{d'} \in \delta_i(\mathsf{ack}^1_{d,c},\alpha)\} \\
 & & \cup\, \{\mathsf{res}^1_{res(d',c'),res(c')} \mid \mathsf{res}^1_{d',c'} \in \delta_i(\mathsf{ack}^1_{d,c},\alpha)\}, \\
\delta'_i(\mathsf{rec}^1_{res(d,c),\mathsf{stop}},\alpha) & = & \{q_{phase3}\}.
\end{array}$$

B4) Transitions of the form $A \in \hat{\delta}_i(q,\alpha)$ with $A, q \in \mathsf{COM}(M_i)$ are already treated implicitly in B2 and B3.

B5) For all $\hat{\delta}_i(q,\alpha) = \emptyset$ define:

$$\begin{array}{rcl}
\delta'_i(q,\alpha) & = & \{\mathsf{res}^1_{\perp,\perp}\}, \\
\delta'_i(\mathsf{ack}^1_{\perp,\perp},\alpha) & = & \{\mathsf{req}^1_{\perp}\}. \\
\delta'_i(\mathsf{rec}^1_{\perp,\mathsf{stop}},\alpha) & = & \{q_{phase3}\}.
\end{array}$$

If $\mathcal{M}$ accepts the considered factor, that is, all components reach the state q_{phase3}, then M'_1 removes one of the remaining symbols of the processed factor at the current window position, performs a restart into a dedicated restart state, and begins with the third phase. The fact whether or not the (RRWW-, RLWW-) component has already made a rewrite step in the current cycle (the last cycle of the simulation in phase 2) can also be encoded within the states. If it has already performed a rewrite step, then only a restart without removing a symbol will be executed. If the factor was completely deleted during the simulation in phase 2, so that no more symbol can be deleted, then a change of the state (SCO transition) into the restart state of the third phase will be applied (the window is then already positioned at the leftmost border of the tape).

Additionally, for simplification of the notation we used SCO transitions in the construction. These can easily be eliminated.

Phase 3. In this phase all components remove the remaining symbols of the factor they worked on in phase 2 as well as the first marker (see lines 6 and 7 of Figure 5.3). Removing the first marker includes setting '$-$' at the first position of the triple. When the tape content starts with $\cent\#$ and the first entry of the triple is a minus or the triple is $[-,-,-]$ (no marker on the tape), then the factor is completely removed, the triples are changed from $[-,b,c]$ to $[c,-,-]$ (even if b and c are minus; see lines 3 and 4), and $\mathcal{M}'$ continues with the first phase again.

If the tape is empty after phase 3, then $\mathcal{M}'$ has guessed a factorization, all factors of which were accepted by $\mathcal{M}$, and thus $\mathcal{M}'$ accepts the input word. If $\mathcal{M}$ chooses a factorization such that one factor is not accepted by $\mathcal{M}$, then M_1' does not change from phase 2 to phase 3 for this factor, since M_1 would not reach the accepting configuration. Hence, M_1' gets stuck during the corresponding second phase. If there exists a valid factorization, then $\mathcal{M}'$ can guess it and accept the input word. If the input does not have a valid factorization, then $\mathcal{M}'$ fails on at least one factor of every factorization. Thus, $\mathcal{M}'$ accepts w if and only if $w \in L^*$. Moreover, the modifications of the components do not change their type, so if $L \in \mathcal{L}(\mathsf{PC\text{-}RLWW})$ ($L \in \mathcal{L}(\mathsf{PC\text{-}RRWW})$, $L \in \mathcal{L}(\mathsf{PC\text{-}RWW})$), then $L^* \in \mathcal{L}(\mathsf{PC\text{-}RLWW})$ ($L^* \in \mathcal{L}(\mathsf{PC\text{-}RRWW})$, $L \in \mathcal{L}(\mathsf{PC\text{-}RWW})$), too. □

The positive closure L^+ of a language L is quite the same as the closure L^* with the only difference that $\varepsilon \in L^+$ holds if and only if $\varepsilon \in L$ (see [HU79]). From Theorem 18 we can also conclude that the language classes $\mathcal{L}(\mathsf{PC\text{-}RWW})$, $\mathcal{L}(\mathsf{PC\text{-}RRWW})$, and $\mathcal{L}(\mathsf{PC\text{-}RLWW})$ are closed under positive closure. The proof only has to be modified in the following way: If the input is ε, then $\mathcal{M}'$ has to simulate $\mathcal{M}$ on ε and check whether $\mathcal{M}$ accepts ε. In the affirmative, $\mathcal{M}'$ accepts ε, otherwise it does not.

Corollary 12. *$\mathcal{L}(\mathsf{PC\text{-}RWW})$, $\mathcal{L}(\mathsf{PC\text{-}RRWW})$, and $\mathcal{L}(\mathsf{PC\text{-}RLWW})$ are closed under positive closure.*

The next closure properties we want to deal with are the closure under homomorphisms and inverse homomorphisms. Due to the strict limit of workspace and the length reducing property for the rewrite operations, we previously establish some more technical features. So we first show, that it is no advantage if we allow a restart operation without applying a rewrite step in the same cycle, additional to the usual restart operation that can only be applied after exactly one rewrite step was executed in the considered cycle. Then we show how to use a subsystem in order to simulate a single restarting automaton with a bigger tape. That is, the tapes of the subsystem's components are put together to one tape, and the communication is used to simulate a single automaton with one (imagined) window operating on the whole common tape.

Lemma 4. *Let M be a restarting automaton of type* (det-)X, $\mathsf{X} \in \mathcal{T}$, *with the following property: M is allowed to perform a restart operation even if no rewrite step was performed in the same cycle. Further, let M' be obtained from M by disallowing these particular restart steps. Then $M \equiv_c M'$.*

Proof. Let

$$q_0 ¢ w \$ = w_0 \vdash_M w_1 \vdash_M w_2 \vdash_M \ldots \vdash_M w_r$$

be an arbitrary computation of M for an input word w. Consider a part of such a computation

$$q_0 ¢ u \$ \vdash_M^* u_1 q u_2 \vdash_M^{Restart} q_0 ¢ u \$,$$

where $q_0 ¢ u\$$ is a restart configuration and either $u_1 = \varepsilon$ and $u_2 = ¢ u \$$ or $u_1 = ¢ u_1'$, $u_2 = u_2' \$$, and $u_1' u_2' = u$. The tape content has not been changed, as no rewrite step took place, and so, the same restart configuration is reached again. If a computation of M, beginning from the initial configuration and reaching a configuration κ, includes such a part, that is, there exists a cycle with a restart but no rewrite

$$q_0 ¢ w \$ \vdash_M^{r_1} q_0 ¢ u \$ \vdash_M^{r_2} u_1 q u_2 \vdash_M^{Restart} q_0 ¢ u \$ \vdash_M^{r_3} \kappa,$$

then M could also perform the computation without this cycle:

$$q_0 ¢ w \$ \vdash_M^{r_1} q_0 ¢ u \$ \vdash_M^{r_3} \kappa.$$

Thus, there exists a computation $q_0 ¢ w \$ \vdash_M^* \kappa$ without cycles of this form. Since M' is defined exactly as M, it follows that $q_0 ¢ w \$ \vdash_{M'}^* \kappa$. In particular, this holds for $\kappa = \mathsf{Accept}$ and for configurations κ containing request and response states. Hence, it follows that $L(M) = L(M')$ and, moreover, $M \equiv_c M'$. □

For deterministic automata a cycle without a rewrite step leads immediately to an infinite computation, since a restarting configuration is reached twice, and the resulting computation loops. Now, we use the result from the previous lemma and show how a PCRA (sub)system of degree n can simulate a single restarting automaton with a tape that is n times the size of the input word. In particular, this result shows that the available workspace within a PCRA system can be increased by an arbitrary constant factor by increasing the number of components.

Lemma 5. *A* (det-)PC-X*-system,* $\mathsf{X} \in \mathcal{T}$, *of degree n can be constructed that behaves on input w like an individual* (det-)X*-automaton on input w^n.*

Proof. Let $\mathcal{S} = (M_1, M_2, \ldots, M_n)$ be a (det-)PC-X-system with $\mathsf{X} \in \mathcal{T}$. Let further $M_{\mathcal{S}} = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_{\mathcal{S}})$ be the automaton that is to be simulated by $\mathcal{S}$.

Basically, the finite control of $M_{\mathcal{S}}$ is simulated by the finite control of M_1, and the tape of $M_{\mathcal{S}}$ is obtained by concatenating the tapes of all components of $\mathcal{S}$ as illustrated in Figure 5.5 (w_1, w_2, and w_n are used in the picture to signalize different tape contents, although all components start their computations with the same input word). Sometimes the tape of $M_{\mathcal{S}}$ is also called the common tape of $\mathcal{S}$. The read/write window of $M_{\mathcal{S}}$ is realized in $\mathcal{S}$ by cooperation of the windows of $M_1, \ldots, M_n$ through communication.

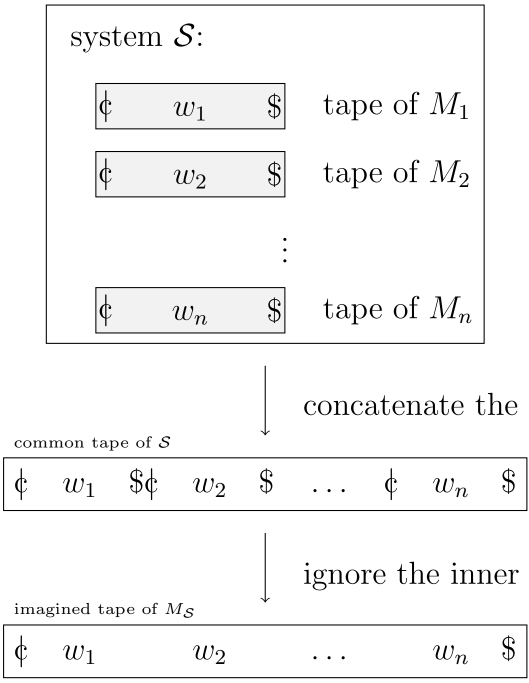
system $\mathcal{S}$:
¢ w_1 \$
tape of M_1
¢ w_2 \$
tape of M_2
¢ w_n \$
tape of M_n
concatenate the tapes of $\mathcal{S}$
common tape of $\mathcal{S}$
¢ w_1 \$¢ w_2 \$ ... ¢ w_n \$
ignore the inner ¢ and \$
imagined tape of $M_{\mathcal{S}}$
¢ w_1 w_2 ... w_n \$

Figure 5.5: The tapes of the system $\mathcal{S}$ are used to simulate the tape of automaton $M_{\mathcal{S}}$.

At the very beginning and after each restart operation, all components except the first one move their windows one position to the right such that the windows are placed directly to the right of the ¢-symbol. Then $\mathcal{S}$ can execute anyone of the following operations of $M_{\mathcal{S}}$: 1) MVR step, 2) MVL step, 3) rewrite step, 4) restart step, 5) accept step, and 6) communication. The last item is necessary if $M_{\mathcal{S}}$ is a component within a system (then, $\mathcal{S}$ can be seen as a subsystem). Before one of the operations can be performed, $\mathcal{S}$ has to determine the current window content. Now, it is explained in detail how $\mathcal{S}$ simulates the different operations.

Reading the current content of the common window. The common window can be positioned on an arbitrary cell of any of the component's tapes. To determine the current content of the common window, $\mathcal{S}$ proceeds as follows. If the window of M_1 does not contain the \$-symbol, then the content of the common window is the content of M_1's window. If the \$-symbol appears in the window of M_1, then M_1 reads some $\alpha_1\$$ with $|\alpha_1| = l < k$. In this case M_1 knows the first l symbols of the common window. But since $l < k$ and the inner ¢- and \$-symbols are ignored, at least one symbol of the common window's content is still missing . Therefore, M_1 tries to obtain the remaining symbols from M_2. It sends a message to M_2 and asks for its window content. Then, M_2 sends its current window content α_2 (excluding ¢ and \$). If $|\alpha_1| + |\alpha_2| < k$, then some symbols of the common window content are still missing. Thus, M_1 asks M_3 for its window content and so forth. This proceeds until either M_1 has obtained altogether k symbols or the obtained string ends by \$, that is, the common window has reached the right hand end of the tape, and M_n sent a string to M_1 that ends by \$.

After this procedure M_1 knows the current content of the common window and can now simulate one of the above mentioned transitions.

Moving the common window. Whenever $M_{\mathcal{S}}$ performs a MVL step or a MVR step ($\mathsf{MVL} \in \delta_{\mathcal{S}}(q, \alpha)$, $\mathsf{MVR} \in \delta_{\mathcal{S}}(q, \alpha)$), then $\mathcal{S}$ has to move the common window one position to the left or to the right, respectively. This is controlled by M_1 as follows. If the content of M_1's window is not equal to \$ (the common window is placed on M_1's tape), then M_1 just performs a MVL step or a MVR step, respectively. If the window is already placed on the leftmost end of the tape, then no MVL operation is possible, and M_1 continous its computation without moving its window. If the window content of M_1 is equal to \$, then the common window is currently positioned on some other tape. Thus, M_1 asks M_2 to move the common window and expects one of three possible answers from M_2: 'yes, I moved the common window', 'no, the common window is not placed on my tape', or 'yes, I moved the common window, and I reached the ¢-symbol' (only for the MVL step). In the former case, the operation is finished, and M_1 continous with reading the current content of the common window again. In the second case, M_1 asks M_3. If M_3 gives the second answer, too, then M_1 asks M_4 and so forth until either one component sends the first answer or M_n sends the second answer, that is, the common window only contains the \$-symbol, and therefore, the window cannot be moved to the right. The third answer is sent by a component if it moves its window to the left and thereby reaches the ¢-symbol. In this situation the common window exceeds the border between the answering component and its predecessor component (for a component M_i the predecessor component is M_{i-1}). Thus, the window of the predecessor component that is currently placed on the \$-symbol has to perform a MVL step, too. If the predecessor component also reaches the ¢-symbol (because its tape contains only ¢\$), then the predecessor of the predecessor has to perform a MVL step and so on. This procedure is controlled by M_1 through a determined communication protocol.

Rewrite step. If $M_{\mathcal{S}}$ performs a rewrite operation $(q, \beta) \in \delta_{\mathcal{S}}(p, \alpha)$, then $\mathcal{S}$ simulates it by the same rewriting at the the same position of the common tape.

Since there are various different options for the position of the common window, we explain now, how $\mathcal{S}$ works for these different cases.

Case 1: The whole common window is placed on the tape of exactly one component M_i ($1 \leq i \leq n$). Then M_i just replaces α by β, and afterwards its window has the correct position, that is, to the right of the currently written β.

Case 2: The common window ranges over the tapes of several components, let us say over the m tapes of components $M_i, M_{i+1}, \ldots, M_{i+m-1}$. Let $\alpha = \alpha_0\alpha_1 \ldots \alpha_{m-1}$, where α_j is the part of α that is placed on the tape of M_{i+j}. That means, α_0 is a non-empty suffix of the tape content of M_i (without \$), $\alpha_1, \ldots, \alpha_{m-2}$ are the possibly empty complete tape contents of $M_{i+1}, \ldots, M_{i+m-2}$ (excluding ¢ and \$), and α_{m-1} is a non-empty prefix of the tape content of M_{i+m-1} (without ¢). This is illustrated in Figure 5.6. Further, let $t = |\{\alpha_j \mid 1 \leq j \leq m-2, \alpha_j = \varepsilon\}|$ be the number of empty tapes. Then, we can distinguish between two more cases: (a) $|\beta| \leq |\alpha| - m + t$ and (b) $|\alpha| - m + t < |\beta| (\leq |\alpha| - 1)$.

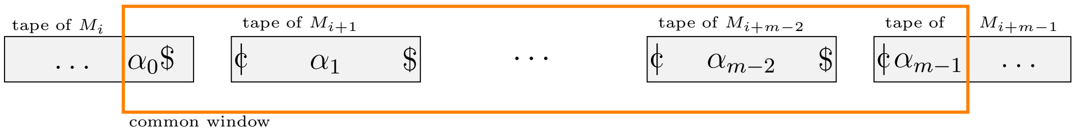

Figure 5.6: The common window of $\mathcal{S}$ ranges over several tapes.

Case 2a: If $|\beta| \leq |\alpha| - m + t$, then β is short enough that each of $M_i, M_{i+1}, \ldots, M_{i+m-1}$ can perform a length reducing rewrite operation. Then there exists a unique factorization of β into $\beta = \beta_0\beta_1 \ldots \beta_{m-1}$ such that:

$$\begin{aligned}\exists r \in \{0, \ldots, m-1\} : \quad & \forall j < r : |\beta_j| = |\alpha_j| - 1, \text{ if } \alpha_j \neq \varepsilon, \text{ and } \beta_j = \varepsilon, \text{ otherwise;} \\ & \wedge |\beta_r| < |\alpha_r|; \\ & \wedge \forall j > r : \beta_j = \varepsilon.\end{aligned}$$

Observe, the factorization of β can be obtained deterministically. All components M_{i+j}, $0 \leq j \leq m-1$, replace α_j by β_j.

Case 2b: In the case that $|\alpha| - m + t < |\beta|$, there exists at least one component of $M_i, M_{i+1}, \ldots, M_{i+m-1}$ that cannot apply a length reducing rewrite. If $M_{\mathcal{S}}$ is an R-, RR-, or RL-automaton, then it can only delete some symbols of α and the corresponding components of $\mathcal{S}$ delete the relevant symbols from their tapes. In the general case ($M_{\mathcal{S}}$ is of type RW, RWW, RRW, RRWW, RLW, or RLWW) α could be rewritten by a β that consists of completely different symbols than α, so that all symbols of α are replaced. Therefore we have to find a solution for those components that cannot make a length reducing rewrite step. They behave as follows. Let M_{i+j}, $0 \leq j < m$, be such a component. It deletes α_j from its tape and stores β_j in the finite control. This is possible because $|\beta_j| < k$, so the length of the stored string is bounded by the constant $k - 1$.

Then, when a component M_i, $1 \leq i \leq n-1$ reads the local window content, moves the local window, or applies further local rewrite steps, the right sentinel \$ is interpreted as the string $\gamma\$$ from the finite control, where γ is the string that is currently stored within the finite control. For M_n the ¢-symbol is interpreted as ¢γ. Later replacings of (a part of) γ are realized by changing it within the finite control.

In order to not forget the stored γ within the finite control, the nonforgetting property is needed. As described in Section 5.4 we can eliminate this property by using some extra components.

Restart step. When $M_{\mathcal{S}}$ performs a restart operation ($\mathsf{Restart} \in \delta_{\mathcal{S}}(q, \alpha)$), then $\mathcal{S}$ behaves as follows. M_1 sends the message 'restart' to each component and performs a restart operation on its own. Whenever one of the other components obtains the restart message, then it performs a restart operation immediately, moves its window one cell to the right, and changes into a request state to await further instructions from M_1. Consider that there exist situations where a component did not perform a rewrite step in the current cycle before it obtains the restart message (e.g. if the common window was only moved over its tape without performing any other operations). For such cases we use Lemma 4, and the component can nevertheless perform the restart operation. Figure 5.7 demonstrates the situation after $\mathcal{S}$ has executed a restart operation.

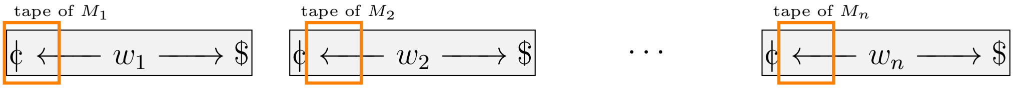

Figure 5.7: Situation of $\mathcal{S}$ after a restart operation.

Accept step. Whenever $M_{\mathcal{S}}$ can perform an accept step ($\mathsf{Accept} \in \delta_{\mathcal{S}}(q, \alpha)$), then M_1 and thus $\mathcal{S}$ can also accept reading α with the common window.

Communication. Consider $M_{\mathcal{S}}$ as a component within a system $\mathcal{C}$. Then, $M_{\mathcal{S}}$ is allowed to communicate with other components of $\mathcal{C}$ by reaching communication states. Our system $\mathcal{S}$ that simulates $M_{\mathcal{S}}$ can then be interpreted as a subsystem within the system $\mathcal{C}$ and has to behave exactly like $M_{\mathcal{S}}$ according to the communication with other components. We call M_1 the 'representative' of $\mathcal{S}$, that is, only M_1 handles all communication between $\mathcal{S}$ and the other components of $\mathcal{C}$. For the simulation, this means that, whenever $M_{\mathcal{S}}$ reaches a request or response state, then M_1 reaches the same communication state, too. Moreover, any transition of $M_{\mathcal{S}}$ that is based on a receive or acknowledge state is handled like any other transition of $M_{\mathcal{S}}$.

Summarizing, we have constructed a system $\mathcal{S}$ that simulates the behaviour of an automaton $M_{\mathcal{S}}$. Thus, in the further work it suffices to describe $M_{\mathcal{S}}$ that is allowed to use a tape of length $l \cdot |w|$, where l is any positive integer and w is the input word (the degree of $\mathcal{S}$ depends on l). □

Using the technique of Lemma 5, we can now show the closure under ε-free homomorphisms and inverse homomorphisms.

Theorem 19. *The language classes $\mathcal{L}$(PC-RWW), $\mathcal{L}$(PC-RRWW), and $\mathcal{L}$(PC-RLWW) are closed under ε-free homomorphisms.*

Proof. Let $L \in \mathcal{L}(\mathsf{X})$ with $\mathsf{X} \in \{\mathsf{PC\text{-}RWW}, \mathsf{PC\text{-}RRWW}, \mathsf{PC\text{-}RLWW}\}$ be a language over the alphabet Σ and $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PCRA system of type X with $L(\mathcal{M}) = L$. Further, let $h : \Sigma^* \to \Delta^*$ be an ε-free homomorphism, i.e. $|h(a)| \geq 1$ for all $a \in \Sigma$, and let $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$ be a set of copies of input symbols such that $\overline{\Sigma} \cap \Delta = \emptyset$. We construct a system $\mathcal{M}' = (M_0, M_{1,1}, M_{1,2}, M_{2,1}, M_{2,2}, \ldots, M_{n,1}, M_{n,2})$ that accepts exactly the language $h(L) \subseteq \Delta^*$.

Basically, each pair $M_{i,1}$ and $M_{i,2}$, $1 \leq i \leq n$, represents a subsystem $\mathcal{M}_i$ of $\mathcal{M}'$ that is used to simulate component M_i from the original system $\mathcal{M}$ as described in Lemma 5.

Now, $\mathcal{M}'$ works in two phases: (1) translation and (2) simulation. In the translation phase M_0 moves its window one position to the right, i.e. directly to the right of ¢, and tries to interpret the window content (or a prefix of it) as an image $h(a)$ for any $a \in \Sigma$. For this purpose, the window size of M_0 is $max\{|h(a)| \mid a \in \Sigma\}$. If the tape content of M_0 does not begin with an image of h, then M_0 gets stuck. Otherwise, M_0 choses nondeterministically a corresponding preimage, sends it to $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$, removes the read image from the tape, executes a restart operation, and repeats these steps. Initially, all $\mathcal{M}_i$, $1 \leq i \leq n$, have placed their windows at the left-hand side of the tapes and wait in a request state for instructions from M_0. If $\mathcal{M}_i$ gets a message from M_0 with a preimage $a \in \Sigma$, then it moves its window to the right over all symbols of $\overline{\Sigma}$ and replaces the first two found input symbols of the tape, i.e. symbols contained in Δ, with the copy of the preimage $\overline{a}$. Thereafter, it performs a restart operation immediately and expects further instructions from M_0 again, waiting in an according request state.

If, after M_0 performed the MVR step, its window contains only \$, then the whole input word was translated into its preimage, and M_0 sends messages to $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ that signalize the end of translation and the beginning of the simulation of $\mathcal{M}$. For an input word $w = h(v)$ for any $v \in \Sigma^*$, the contents of the tapes of $\mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}_n$ now have the form $\cent\overline{v}z\$$, where $z \in \Delta^*$ consists of the remaining input symbols. These symbols are removed from all tapes between the translation phase and the simulation phase. Observe that each of the subsystems $\mathcal{M}_1, \dots, \mathcal{M}_n$ consists of two components and thus has a common tape size of $2 \cdot |w|$. That is indeed enough space for the translation, since at most $|w|$ images have to be translated, and in each translation step exactly two symbols are rewritten. In the remaining computation of $\mathcal{M}'$, M_0 does not perform a computation step anymore.

In the simulation phase, the subsystems $\mathcal{M}_i$, $1 \leq i \leq n$, behave on input $\overline{v}$ mainly like the original components M_i on input v just as it is described in Lemma 5. For this, all symbols of Σ within transitions of M_i are replaced with their according copy from $\overline{\Sigma}$, and whenever M_i performs a restart step, then $\mathcal{M}_i$ restarts in the original initial state of M_i instead of its own initial state. For doing this, the nonforgetting property is used that can be eliminated as described in Theorem 11.

For any input w, if w is an image of some v, then v can be guessed during the translation phase, $\mathcal{M}$ is simulated on v by $\mathcal{M}'$, and $\mathcal{M}'$ accepts if and only if $\mathcal{M}$ accepts. Otherwise, i.e. the input is no valid image, M_0 gets stuck in the translation phase, $\mathcal{M}'$ does not reach the simulation phase, and thus cannot accept the input. Hence, $L(\mathcal{M}') = h(L(\mathcal{M}))$. Moreover, $\mathcal{M}'$ is of the same type as $\mathcal{M}$, so $h(L) \in \mathcal{L}(\mathsf{X})$, which completes the proof. □

Theorem 20. *The language classes $\mathcal{L}$(det-PC-X) and $\mathcal{L}$(PC-X) are closed under inverse homomorphisms for all X ∈ {RWW, RRWW, RLWW}.*

Proof. Let $L \in \mathcal{L}$((det-)PC-X) for X ∈ {RWW, RRWW, RLWW} be a language over the alphabet Δ. Then there exists a (det-)PC-X-system $\mathcal{M} = (M_1, M_2, \dots, M_n)$ with $L(\mathcal{M}) = L$. Let further $h : \Sigma^* \to \Delta^*$ be a homomorphism and $r = max\{|h(a)| \mid a \in \Sigma\}$ is the size of the longest image of h. We build a system $\mathcal{M}'$ with $L(\mathcal{M}') = h^{-1}(L)$, whose construction is nearly the same as that of the system constructed in the proof of Theorem 19. The only difference is the opposite direction of the translation, i.e. an input $w \in \Sigma^*$ of $\mathcal{M}'$ is interpreted as a preimage of h that is translated into the uniquely determined image $h(w)$. For this purpose, $\mathcal{M}'$ consists of $1 + n \cdot (r + 1)$ components:

$$\mathcal{M}' = (M_0, M_{1,1}, M_{1,2}, \dots, M_{1,r+1}, M_{2,1}, M_{2,2}, \dots, M_{2,r+1}, \\ \dots, M_{n,1}, M_{n,2}, \dots, M_{n,r+1})$$

and, for each $1 \leq i \leq n$, the components $M_{i,1}, M_{i,2}, \dots, M_{i,r+1}$ are combined to a subsystem $\mathcal{M}_i$, so that for each component M_i of $\mathcal{M}$ there is a subsystem $\mathcal{M}_i$ in $\mathcal{M}'$.

For input w, the common tapes of $\mathcal{M}_1, \ldots, \mathcal{M}_n$ have a size of $(r+1) \cdot |w|$. In each translation step $r+1$ symbols of the common tapes are replaced by $\overline{h(a)}$ for a symbol a of the input word. After $\mathcal{M}'$ executed $|w|$ translation steps, the common tape of $\mathcal{M}_1, \ldots, \mathcal{M}_n$ contains the inscription $¢\overline{h(w)}\$$, and $\mathcal{M}$ can be simulated, where each of the original components M_i is simulated by the subsystem $\mathcal{M}_i$.

Summerizing, each input w of $\mathcal{M}'$ is translated into the unique image $h(w)$, and then $\mathcal{M}$ is simulated on $h(w)$. Thus, $\mathcal{M}'$ accepts on input w if and only if $\mathcal{M}$ accepts on input $h(w)$. Hence, $L(\mathcal{M}') = h^{-1}(L(\mathcal{M}))$. Moreover, $\mathcal{M}'$ is of the same type as $\mathcal{M}$, and thus, $h^{-1}(L) \in \mathcal{L}(\mathsf{(det\text{-})PC\text{-}X})$. □

The question of whether the language classes accepted by PCRA systems are also closed under arbitrary homomorphisms, i.e. including homomorphisms that can erase symbols, must be answered with no. We prove this fact by using the known fact that each recursively enumerable language L can be written as $h(L_1 \cap L_2)$, where h is a homomorphism and L_1, L_2 are deterministic context-free languages [Har78].

Theorem 21. *The language classes $\mathcal{L}(\mathsf{(det\text{-})PC\text{-}X})$ are not closed under arbitrary homomorphisms for each $\mathsf{X} \in \mathcal{T}$.*

Proof. We prove this by contradiction and assume that $\mathcal{L}(\mathsf{(det\text{-})PC\text{-}X})$ is closed under applying arbitrary homomorphisms for any $\mathsf{X} \in \mathcal{T}$. Let L be an arbitrary recursively enumerable language with the representation $L = h(L_1 \cap L_2)$, where h is a homomorphism and L_1, L_2 are deterministic context-free languages [Har78]. Since $\mathcal{L}(\mathsf{(det\text{-})PC\text{-}X}) \supseteq \mathsf{DCFL}$, it follows that $L_1, L_2 \in \mathcal{L}(\mathsf{(det\text{-})PC\text{-}X})$. From Theorem 14 we know that $\mathcal{L}(\mathsf{(det\text{-})PC\text{-}X})$ is closed under intersection, hence $(L_1 \cap L_2) \in \mathcal{L}(\mathsf{(det\text{-})PC\text{-}X})$. Because of our assumption we have $h(L_1 \cap L_2) \in \mathcal{L}(\mathsf{(det\text{-})PC\text{-}X})$. But then $\mathsf{RE} \subseteq \mathcal{L}(\mathsf{(det\text{-})PC\text{-}X})$ would follow, which contradicts the fact that $\mathcal{L}(\mathsf{(det\text{-})PC\text{-}X}) \subseteq \mathsf{CSL} \subset \mathsf{RE}$ (see Corollary 21). Thus, the assumption is false, that is, $\mathcal{L}(\mathsf{(det\text{-})PC\text{-}X})$ is not closed under arbitrary homomorphisms for any $\mathsf{X} \in \mathcal{T}$. □

Since in the proof of Theorem 20 Δ can include symbols that are not contained in Σ, the translation of the input into its image according to the homomorphism needs auxiliary symbols in general. Therefore, the result is restricted to the language classes of PCRA systems that are allowed to use auxiliary symbols. The following table summarizes the closure properties that we have established within this section. The first column contains the language classes that are mainly divided into deterministic and nondeterministic classes and classes with auxiliary symbols and those without. The meaning of the operations are from left to right: union, intersection, intersection with a regular language, complementation, marked product, product, Kleene closure, positive closure, arbitrary homomorphism, ε-free homomorphism, and inverse homomorphism.

	$\cup$	$\cap$	$\cap_{REG}$	^{c}	$\cdot_{\#}$	$\cdot$	$*$	$+$	h	h_{ε}	h^{-1}
$\mathcal{L}$(det-PC-R(R)(W))	+	+	+	+	+	?	?	?	−	?	?
$\mathcal{L}$(det-PC-RL(W))	+	+	+	+	+	?	?	?	−	?	?
$\mathcal{L}$(det-PC-R(R)WW)	+	+	+	+	+	?	?	?	−	?	+
$\mathcal{L}$(det-PC-RLWW)	+	+	+	+	+	?	?	?	−	?	+
$\mathcal{L}$(PC-R(R)(W))	+	+	+	?	+	?	?	?	−	?	?
$\mathcal{L}$(PC-RL(W))	+	+	+	?	+	?	?	?	−	?	?
$\mathcal{L}$(PC-R(R)WW)	+	+	+	?	+	+	+	+	−	+	+
$\mathcal{L}$(PC-RLWW)	+	+	+	?	+	+	+	+	−	+	+

The next corollary gives a concluding result of this section, namely that the language classes $\mathcal{L}$(PC-RWW), $\mathcal{L}$(PC-RRWW), and $\mathcal{L}$(PC-RLWW) are so-called AFLs (abstract families of languages).

Corollary 13. *The language classes $\mathcal{L}$(PC-RWW), $\mathcal{L}$(PC-RRWW), and $\mathcal{L}$(PC-RLWW) are AFLs (Abstract Families of Languages).*

Proof. This follows immediately from the closure under ε-free morphisms, inverse morphisms, intersection with regular languages, union, concatenation, and positive closure [RS97]. □

Corollary 14. *The language classes $\mathcal{L}$(PC-RWW), $\mathcal{L}$(PC-RRWW), and $\mathcal{L}$(PC-RLWW) are not full AFLs.*

Proof. As we have seen in Theorem 21, $\mathcal{L}$(PC-RWW), $\mathcal{L}$(PC-RRWW), and $\mathcal{L}$(PC-RLWW) are not closed under applying arbitrary homomorphisms. Thus, they are not full AFLs. □

5.6 Computational power

In this section we investigate the computational power of PCRA systems and try to find some relations to languages classes of single restarting automata and other well-known complexity classes. The first lemma is quite obvious, as a system is at least as powerful as a system of the same type with less components.

Lemma 6. *Let $\mathsf{X} \in \mathcal{T}$, $\mathsf{p} \in \{\varepsilon, \mathsf{det}, \mathsf{mon}, \mathsf{det\text{-}mon}\}$, and $n \geq 1$. Then*

$$\mathcal{L}(\mathsf{p\text{-}PC\text{-}X}(n)) \subseteq \mathcal{L}(\mathsf{p\text{-}PC\text{-}X}(n+1)).$$

In particular, a PCRA system is at least as powerful as a single restarting automaton of the same type. The question of whether there are cases in which the number of the components yields a strict hierarchy, is still open.

The next two lemmata result directly from the definition of acceptance for the PCRA systems. Here, the notation $L(M)$ for a component M describes the language that is accepted by M without performing any communication, that is, if we interprete M as a single restarting automaton outside the system (where reaching a communication state just means to reject the input).

Lemma 7. *Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PCRA system. Then the following result holds:*

$$\forall i \in \{1, \ldots, n\} : L(M_i) \subseteq L(\mathcal{M}).$$

Proof. A PCRA system accepts an input word if and only if at least one component reaches the accepting configuration. Thus, each input word that is accepted by any component is also accepted by the system. □

Lemma 8. *Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a PCRA system and let $\mathcal{M}' = (M_1, M_2, \ldots, M_n, M_{n+1})$, that is, $\mathcal{M}'$ includes $\mathcal{M}$ and an additional component M_{n+1}. Then $L(\mathcal{M}) \cup L(M_{n+1}) \subseteq L(\mathcal{M}')$.*

Proof. If an input word is accepted by $\mathcal{M}$ or M_{n+1}, then it is accepted by at least one component of $\mathcal{M}$ or by M_{n+1}. Thus, the input is accepted by at least one component of $\mathcal{M}'$ and hence by $\mathcal{M}'$. In the case that M_{n+1} does not communicate with any component of $\mathcal{M}$, then $L(\mathcal{M}) \cup L(M_{n+1})$ coincides with $L(\mathcal{M}')$. On the other hand, if M_{n+1} accepts some words only due to communications between M_{n+1} and components of $\mathcal{M}$, then $L(\mathcal{M}) \cup L(M_{n+1}) \subset L(\mathcal{M}')$ holds. □

For our next consideration we will use the language of the following example.

Example 17. The language

$$L_{a^nb^{(2)n(+r)}} = \{a^nb^n \mid n \geq 0\} \cup \{a^nb^m \mid m > 2n \geq 0\}$$

is accepted by the following det-mon-PC-R(2)-system $\mathcal{M}_{a^nb^{(2)n(+r)}} = (M_1, M_2)$. Since no communication is needed, we can describe the components by their meta-instructions:

$$\begin{array}{llll} M_1: & (¢\$, \mathsf{Accept}), & M_2: & (¢ \cdot a^*, aabbb \to ab), \\ & (¢ \cdot a^*, aabb \to ab), & & (¢, abbb \to b), \\ & (¢ab\$, \mathsf{Accept}), & & (¢, bb \to b), \\ & & & (¢b\$, \mathsf{Accept}). \end{array}$$

The first component just accepts the words of the first part of the language, and the second component accepts the second part. Due to the definition, the system accepts an input word if and only if either M_1 or M_2 accepts it. □

Example 17 shows that $L_{a^nb^{(2)n(+r)}} \in \mathcal{L}(\mathsf{det\text{-}mon\text{-}PC\text{-}R}(2))$, and thus, it is included in all the language classes $\mathcal{L}((\mathsf{det\text{-}})(\mathsf{mon\text{-}})\mathsf{PC\text{-}R(R)(W)}(2))$. Furthermore, we know from [Ott06] that $L_{a^nb^{(2)n(+r)}}$ can in fact be accepted by a det-RWW-automaton, but it cannot be accepted by any restarting automaton without auxiliary symbols. Together with Lemma 6 we can conclude the following:

Corollary 15. *For all* $X \in \{R, RR, RW, RRW, RL, RLW\}$,

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

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.

Theorem 22.

$$\begin{array}{llll} \text{(a)} & \mathcal{L}(RLWW) \subseteq \mathcal{L}(PC\text{-}RLW(2)), & \text{(d)} & \mathcal{L}(det\text{-}RLWW) \subseteq \mathcal{L}(det\text{-}PC\text{-}RLW(2)), \\ \text{(b)} & \mathcal{L}(RRWW) \subseteq \mathcal{L}(PC\text{-}RRW(2)), & \text{(e)} & \mathcal{L}(det\text{-}RRWW) \subseteq \mathcal{L}(det\text{-}PC\text{-}RRW(2)), \\ \text{(c)} & \mathcal{L}(RWW) \subseteq \mathcal{L}(PC\text{-}RW(2)), & \text{(f)} & \mathcal{L}(det\text{-}RWW) \subseteq \mathcal{L}(det\text{-}PC\text{-}RW(2)). \end{array}$$

Proof. Assume that the language L is accepted by an R(R)WW-automaton or det-R(R)WW-automaton, respectively. In [NO03] it was proved by Niemann and Otto that a language L is accepted by a (deterministic) R(R)WW-automaton if and only if there exist a (deterministic) R(R)W-automaton M and a regular language R such that $L = L(M) \cap R$. This proof can also be applied to RLWW-automata [Ott06]. Moreover, REG is a subset of all the language classes $\mathcal{L}(\mathsf{det\text{-}RW})$, $\mathcal{L}(\mathsf{det\text{-}RRW})$,

$\mathcal{L}$(det-RLW), $\mathcal{L}$(RW), $\mathcal{L}$(RRW), and $\mathcal{L}$(RLW) [Mrá01, Ott06]. Since all language classes obtained from PCRA systems are closed under intersection, there exists a PCRA system of type (det-)PC-RRW, (det-)PC-RW, or (det-)PC-RLW, respectively, that accepts L. □

Theorem 22 can be generalized as follows:

Corollary 16. *For each $n \geq 2$ holds:*

$$\begin{array}{lrcl}
\text{(a)} & \mathcal{L}(\textit{PC-RLWW}(n)) & \subseteq & \mathcal{L}(\textit{PC-RLW}(n+1)), \\
\text{(b)} & \mathcal{L}(\textit{PC-RRWW}(n)) & \subseteq & \mathcal{L}(\textit{PC-RRW}(n+1)), \\
\text{(c)} & \mathcal{L}(\textit{PC-RWW}(n)) & \subseteq & \mathcal{L}(\textit{PC-RW}(n+1)), \\
\text{(d)} & \mathcal{L}(\textit{det-PC-RLWW}(n)) & \subseteq & \mathcal{L}(\textit{det-PC-RLW}(n+1)), \\
\text{(e)} & \mathcal{L}(\textit{det-PC-RRWW}(n)) & \subseteq & \mathcal{L}(\textit{det-PC-RRW}(n+1)), \\
\text{(f)} & \mathcal{L}(\textit{det-PC-RWW}) & \subseteq & \mathcal{L}(\textit{det-PC-RW}(n+1)).
\end{array}$$

Proof. Here, we apply the proof of Theorem 3.1 of [NO03] to the components of a PCRA system. Let $\mathcal{M} = (M_1, \ldots, M_n)$ be a centralized system that accepts if and only if its first component accepts, and the components of $\mathcal{M}$ are allowed to use auxiliary symbols. Further, let $M_i = (Q_i, \Sigma, \Gamma_i, ¢, \$, q_i, k, \delta_i)$ for all $1 \leq i \leq n$. We construct a system $\mathcal{M}' = (M_1', \ldots, M_n', M_{n+1})$ with $M_i' = (Q_i, \Gamma_i, \Gamma_i, ¢, \$, q_i, k, \delta_i)$ for all $1 \leq i \leq n$. Observe that each M_i' is communicational equivalent to M_i. Then, M_1' is modified such that instead of accepting the input it initiates a communication with M_{n+1}. Finally, M_{n+1} works like a finite automaton that accepts Σ^* and communicates with M_1' if and only if the input is contained in Σ^*. If this communication step is successful, then and only then M_1' accepts. Thus, $L(\mathcal{M}') = L(\mathcal{M})$. □

Theorem 23. *DCFL* $\subset \mathcal{L}$(*det-mon-PC-R*(2)).

Proof. The inclusion follows immediately from DCFL $= \mathcal{L}$(det-mon-R) [JMPV95] and Corollary 15. □

Based on the definition of the acceptance criterion for PCRA systems, we can extend this result to finite unions and intersections of deterministic context-free languages.

Corollary 17. *Every finite union of deterministic context-free languages can be accepted by a det-mon-PC-R-system without using communication steps.*

Corollary 18. *Every finite intersection of deterministic context-free languages can be accepted by a det-mon-PC-R-system using a constant number of communication steps. Moreover, the used number of communication steps is at most the degree of the system minus one.*

Proof. For each deterministic context-free language that takes part in the intersection, one component is used to check membership. Then, if the input is an element of the first language, the first component sends a message to all the other components. Each of the other components for which the input is an element of the corresponding language replies to the first component. If all components reply to the first one, then the first one accepts the input, and hence, the system accepts the input. □

Particularly, we used this for the systems $\mathcal{M}_{a^nb^{(2)n(+r)}}$, $\mathcal{M}_{Gladkij}$, and $\mathcal{M}_{a^nb^nc^nd^n}$ accepting the languages

$$\begin{aligned}
L_{a^nb^{(2)n(+r)}} &= \{a^nb^n \mid n \geq 0\} \cup \{a^nb^m \mid m > 2n \geq 0\},\\
L_{Gladkij} &= \{w\#w^R\#w \mid w \in \{a,b\}^*\}\\
&= \{w\#w^R\#u \mid w,u \in \{a,b\}^*\} \cap \{u\#w^R\#w \mid w,u \in \{a,b\}^*\}, \text{ and}\\
L_{a^nb^nc^nd^n} &= \{a^nb^nc^nd^n \mid n \geq 1\}\\
&= \{a^nb^nc^id^j \mid n,i,j \geq 1\} \cap \{a^ib^nc^nd^j \mid n,i,j \geq 1\} \cap\\
&\qquad\qquad\qquad\qquad \{a^ib^jc^nd^n \mid n,i,j \geq 1\}.
\end{aligned}$$

Corollary 19. *The class of all finite intersections of deterministic context-free languages is a proper subset of $\mathcal{L}$(det-mon-PC-R).*

Proof. The copy language $L_{w\#w}$ cannot be written as a finite intersection of (deterministic) context-free languages [Wot73]. On the other hand, we have seen in Example 9 that this language is accepted by a det-mon-PC-R(2)-system. □

Let $\mathcal{L}$ be a language class. The boolean closure of $\mathcal{L}$ is the smallest family of languages that contains $\mathcal{L}$ and that is closed under (finite) union and complementation [Sal73]. Thus, we have established a first lower bound for PCRA systems, which is the boolean closure of the deterministic context-free languages. In the next part of this section we will improve this lower bound by comparing PCRA systems to multi-head automata and systems of parallel communicating finite automata.

5.6.1 Comparison with multi-head automata and PC systems of finite automata

It is already known that some particular types of restarting automata with window size one characterize exactly the set of the regular languages: R(1), RW(1), RWW(1) (see [Mrá01]), and det-RR(1) (see [Rei07]). We want to use this fact to compare the PC systems of finite automata with PC systems of such a simple type of restarting automata, and we will see that, although the components have the same computational power (accepting the regular languages), the various approaches of communication and cooperation lead to systems with different computational power.

The connections between the language classes characterized by multi-head automata and PCFA systems are given in Section 3. In the next theorem we put the language classes of PCRA systems in this context. For doing so, we show by a straightforward construction that every (deterministic) n-head automaton can be simulated by a (deterministic) PCRA system with n components and window size one. Moreover, there are languages accepted by $\mathsf{det\text{-}PC\text{-}R}(2,1)$-systems, for which there does not exist a (nondeterministic) one-way multi-head automaton accepting the same language.

Theorem 24.

$$\begin{array}{ll} \text{(a)} & \mathcal{L}(\mathit{1\text{-}NFA}(1)) = \mathcal{L}(\mathit{PCFA}(1)) = \mathcal{L}(\mathit{PC\text{-}R}(1,1)) \\ & \qquad = \mathcal{L}(\mathit{1\text{-}DFA}(1)) = \mathcal{L}(\mathit{DPCFA}(1)) = \mathcal{L}(\mathit{det\text{-}PC\text{-}R}(1,1)) = \mathit{REG}. \end{array}$$

For all $n \geq 2$,

$$\begin{array}{lccccc} \text{(b)} & & & \mathcal{L}(\mathit{2\text{-}NFA}(n)) & \subseteq & \mathcal{L}(\mathit{PC\text{-}RL}(n,1)), \\ \text{(c)} & & & \mathcal{L}(\mathit{2\text{-}DFA}(n)) & \subseteq & \mathcal{L}(\mathit{det\text{-}PC\text{-}RL}(n,1)), \\ \text{(d)} & \mathcal{L}(\mathit{PCFA}(n)) & = & \mathcal{L}(\mathit{1\text{-}NFA}(n)) & \subset & \mathcal{L}(\mathit{PC\text{-}R}(n,1)),\ \textit{and} \\ \text{(e)} & \mathcal{L}(\mathit{DPCFA}(n)) & = & \mathcal{L}(\mathit{1\text{-}DFA}(n)) & \subset & \mathcal{L}(\mathit{det\text{-}PC\text{-}R}(n,1)). \end{array}$$

Proof. Line (a) follows immediately from the equivalence of $\mathcal{L}(\mathsf{R}(1))$, $\mathcal{L}(\mathsf{det\text{-}R}(1))$, and the class of regular languages [Mrá01, Rei07]. Statement (b) is proved by a straightforward construction of a $\mathsf{PC\text{-}RL}(n,1)$-system from a two-way n-head automaton. Let $A = (Q_A, \Sigma, n, \delta_A, ¢, \$, q_0, F)$ be a nondeterministic two-way n-head finite automaton. A PCRA system $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ of type $\mathsf{PC\text{-}RL}(n,1)$ 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 various components of $\mathcal{M}$ are defined as follows. For each $2 \leq i \leq n$, $M_i = (Q_i, \Sigma, \Sigma, ¢, \$, q_0^{(i)}, 1, \delta_i)$, where $Q_i = \{q_0^{(i)}, \mathsf{req}^1, \mathsf{rec}^1_{-1}, \mathsf{rec}^1_1, \mathsf{rec}^1_0\} \cup \{\, \mathsf{res}^1_a, \mathsf{ack}^1_a \mid a \in \Sigma \,\}$ and

$$\begin{array}{lcllcl} \delta_i(q_0^{(i)}, a) & = & \{\mathsf{res}^1_a\}, & \delta_i(\mathsf{rec}^1_{-1}, a) & = & \{(q_0^{(i)}, \mathsf{MVL})\}, \\ \delta_i(\mathsf{ack}^1_a, a) & = & \{\mathsf{req}^1\}, & \delta_i(\mathsf{rec}^1_1, a) & = & \{(q_0^{(i)}, \mathsf{MVR})\}, \\ & & & \delta_i(\mathsf{rec}^1_0, a) & = & \{\mathsf{res}^1_a\}. \end{array}$$

for all $a \in \Sigma \cup \{¢, \$\}$. The component $M_1 = (Q_1, \Sigma, \Sigma, ¢, \$, \mathsf{req}^2_{\langle q_0 \rangle}, 1, \delta_1)$ is defined as follows, where $q \in Q_A$ and $a \in \Sigma \cup \{¢, \$\}$, and Q_1 is given implicitly through the description of δ_1:

$$\begin{aligned}
\delta_1(q,a) &= \{\mathsf{req}^2_{\langle q\rangle}\},\\
\delta_1(\mathsf{rec}^2_{\langle q\rangle,c_2},a) &= \{\mathsf{req}^3_{\langle q,c_2\rangle}\},\\
\delta_1(\mathsf{rec}^3_{\langle q,c_2\rangle,c_3},a) &= \{\mathsf{req}^4_{\langle q,c_2,c_3\rangle}\},\\
&\ \ \vdots\\
\delta_1(\mathsf{rec}^{n-1}_{\langle q,c_2,c_3,\ldots,c_{n-2}\rangle,c_{n-1}},a) &= \{\mathsf{req}^n_{\langle q,c_2,\ldots,c_{n-1}\rangle}\},\\
\delta_1(\mathsf{rec}^n_{\langle q,c_2,c_3,\ldots,c_{n-1}\rangle,c_n},a) &=\\
&\{\,\mathsf{res}^n_{\langle p,d_1,d_2,\ldots,d_{n-1}\rangle,d_n} \mid (p,(d_1,d_2,d_3,\ldots,d_n)) \in \delta_A(q,(a,c_2,c_3,\ldots,c_n))\,\}\\
&\cup\{\,\mathsf{Accept} \mid \delta_A(q,(a,c_2,c_3,\ldots,c_n)) = \emptyset \text{ and } q \in F\,\},\\
\delta_1(\mathsf{ack}^n_{\langle p,d_1,d_2,\ldots,d_{n-1}\rangle,d_n},a) &= \{\mathsf{res}^{n-1}_{\langle p,d_1,d_2,\ldots,d_{n-2}\rangle,d_{n-1}}\},
\end{aligned}$$

$$\begin{aligned}
\delta_1(\mathsf{ack}^{n-1}_{\langle p,d_1,d_2,\ldots,d_{n-2}\rangle,d_{n-1}},a) &= \{\mathsf{res}^{n-2}_{\langle p,d_1,d_2,\ldots,d_{n-3}\rangle,d_{n-2}}\},\\
&\ \ \vdots\\
\delta_1(\mathsf{ack}^3_{\langle p,d_1,d_2\rangle,d_3},a) &= \{\mathsf{res}^2_{\langle p,d_1\rangle,d_2}\},\\
\delta_1(\mathsf{ack}^2_{\langle p,d_1\rangle,d_2},a) &= \begin{cases}\{(p,\mathsf{MVL})\}, & \text{if } d_1=-1,\\ \{\mathsf{req}^2_{\langle p\rangle}\}, & \text{if } d_1=0,\\ \{(p,\mathsf{MVR})\}, & \text{if } d_1=1.\end{cases}
\end{aligned}$$

For an input word $w \in \Sigma^*$, the initial configuration of A is $(q_0¢w\$,\ldots,q_0¢w\$)$, and the simulation of A by $\mathcal{M}$ starts with

$$(q_0^{(1)}¢w\$, q_0^{(2)}¢w\$,\ldots,q_0^{(n)}¢w\$) \vdash_{\mathcal{M}} (\mathsf{req}^2_{\langle q_0\rangle}¢w\$, \mathsf{res}^1_{¢}¢w\$,\ldots,\mathsf{res}^1_{¢}¢w\$).$$

A computation step of A of the form

$$(u_1qa_1v_1, u_2qa_2v_2,\ldots,u_nqa_nv_n) \vdash_A (x_1pb_1y_1, x_2pb_2y_2,\ldots,x_npb_ny_n)$$

is then simulated by the following computation of $\mathcal{M}$:

$$\begin{array}{lllll}
 & (u_1\mathsf{req}^2_{\langle q\rangle}a_1v_1, & u_2\mathsf{res}^1_{a_2}a_2v_2, & u_3\mathsf{res}^1_{a_3}a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{res}^1_{a_{n-1}}a_{n-1}v_{n-1}, & u_n\mathsf{res}^1_{a_n}a_nv_n) \\
\vdash_{\mathcal{M}} & (u_1\mathsf{rec}^2_{\langle q\rangle,a_2}a_1v_1, & u_2\mathsf{ack}^1_{a_2}a_2v_2, & u_3\mathsf{res}^1_{a_3}a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{res}^1_{a_{n-1}}a_{n-1}v_{n-1}, & u_n\mathsf{res}^1_{a_n}a_nv_n) \\
\vdash_{\mathcal{M}} & (u_1\mathsf{req}^3_{\langle q,a_2\rangle}a_1v_1, & u_2\mathsf{req}^1a_2v_2, & u_3\mathsf{res}^1_{a_3}a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{res}^1_{a_{n-1}}a_{n-1}v_{n-1}, & u_n\mathsf{res}^1_{a_n}a_nv_n) \\
\vdash_{\mathcal{M}} & (u_1\mathsf{rec}^3_{\langle q,a_2\rangle,a_3}a_1v_1, & u_2\mathsf{req}^1a_2v_2, & u_3\mathsf{ack}^1_{a_3}a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{res}^1_{a_{n-1}}a_{n-1}v_{n-1}, & u_n\mathsf{res}^1_{a_n}a_nv_n) \\
 & & & \vdots & \\
\vdash_{\mathcal{M}} & (u_1\mathsf{req}^n_{\langle q,a_2,\dots,a_{n-1}\rangle}a_1v_1, & u_2\mathsf{req}^1a_2v_2, & u_3\mathsf{req}^1a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{req}^1a_{n-1}v_{n-1}, & u_n\mathsf{res}^1_{a_n}a_nv_n) \\
\vdash_{\mathcal{M}} & (u_1\mathsf{rec}^n_{\langle q,a_2,\dots,a_{n-1}\rangle,a_n}a_1v_1, & u_2\mathsf{req}^1a_2v_2, & u_3\mathsf{req}^1a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{req}^1a_{n-1}v_{n-1}, & u_n\mathsf{ack}^1_{a_n}a_nv_n) \\
\vdash_{\mathcal{M}} & (u_1\mathsf{res}^n_{\langle p,d_2,\dots,d_{n-1}\rangle,d_n}a_1v_1, & u_2\mathsf{req}^1a_2v_2, & u_3\mathsf{req}^1a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{req}^1a_{n-1}v_{n-1}, & u_n\mathsf{req}^1a_nv_n) \\
\vdash_{\mathcal{M}} & (u_1\mathsf{ack}^n_{\langle p,d_2,\dots,d_{n-1}\rangle,d_n}a_1v_1, & u_2\mathsf{req}^1a_2v_2, & u_3\mathsf{req}^1a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{req}^1a_{n-1}v_{n-1}, & u_n\mathsf{rec}^1_{d_n}a_nv_n) \\
\vdash_{\mathcal{M}} & (u_1\mathsf{res}^{n-1}_{\langle p,d_2,\dots,d_{n-2}\rangle,d_{n-1}}a_1v_1, & u_2\mathsf{req}^1a_2v_2, & u_3\mathsf{req}^1a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{req}^1a_{n-1}v_{n-1}, & x_np_nb_ny_n) \\
\vdash_{\mathcal{M}} & (u_1\mathsf{ack}^{n-1}_{\langle p,d_2,\dots,d_{n-2}\rangle,d_{n-1}}a_1v_1, & u_2\mathsf{req}^1a_2v_2, & u_3\mathsf{req}^1a_3v_3, & \dots \\
 & & & u_{n-1}\mathsf{rec}^1_{d_{n-1}}a_{n-1}v_{n-1}, & x_n\mathsf{res}^1_{b_n}b_ny_n) \\
 & & & \vdots & \\
\vdash_{\mathcal{M}} & (u_1\mathsf{res}^2_{\langle p,d_1\rangle,d_2}a_1v_1, & u_2\mathsf{req}^1a_2v_2, & x_3p_3b_3y_3, & \dots \\
 & & & x_{n-1}\mathsf{res}^1_{b_{n-1}}b_{n-1}y_{n-1}, & x_n\mathsf{res}^1_{b_n}b_ny_n) \\
\vdash_{\mathcal{M}} & (u_1\mathsf{ack}^2_{\langle p,d_1\rangle,d_2}a_1v_1, & u_2\mathsf{rec}^1_{d_2}a_2v_2, & x_3\mathsf{res}^1_{b_3}b_3y_3, & \dots \\
 & & & x_{n-1}\mathsf{res}^1_{b_{n-1}}b_{n-1}y_{n-1}, & x_n\mathsf{res}^1_{b_n}b_ny_n) \\
\vdash_{\mathcal{M}} & (x_1p_1b_1y_1, & x_2p_2b_2y_2, & x_3\mathsf{res}^1_{b_3}b_3y_3, & \dots \\
 & & & x_{n-1}\mathsf{res}^1_{b_{n-1}}b_{n-1}y_{n-1}, & x_n\mathsf{res}^1_{b_n}b_ny_n) \\
\vdash_{\mathcal{M}} & (x_1\mathsf{req}^2_{\langle p\rangle}b_1y_1, & x_2\mathsf{res}^1_{b_2}b_2y_2, & x_3\mathsf{res}^1_{b_3}b_3y_3, & \dots \\
 & & & x_{n-1}\mathsf{res}^1_{b_{n-1}}b_{n-1}y_{n-1}, & x_n\mathsf{res}^1_{b_n}b_ny_n),
\end{array}$$

where, for all $2 \le i \le n$, $p_i = \mathsf{res}^1_{b_i}$ if $d_i = 0$, and $p_i = q_0^{(i)}$ otherwise; $p_1 = \mathsf{req}^2_{\langle p\rangle}$ if $d_1 = 0$, and $p_1 = p$ otherwise; and the last computation step is only executed if $p_1 = p$ or $p_2 = q_0^{(2)}$. Whenever A reaches an accepting configuration, that is, A gets into a final state such that no transition is applicable anymore, then M_1 can reach the accepting configuration, and therewith $\mathcal{M}$ can accept the input. It follows that $L(\mathcal{M}) = L(A)$.

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 automaton. In the latter case the inclusion is proper, as the marked mirror language $L_{w\#w^R} = \{w\#w^R \mid w \in \{a,b\}^+\}$ is not accepted by any one-way multi-head automaton, but it is accepted by the det-PC-R(2,1)-system $\mathcal{M}_{w\#w^R}$ that is given in the Example 18 below. □

Example 18. The marked mirror language

$$L_{w\#w^R} = \{w\#w^R \mid w \in \{a,b\}^+\}$$

is accepted by the following det-PC-R(2,1)-system $\mathcal{M}_{w\#w^R} = (M_1, M_2)$:
$M_1 = (\{q_0, q_1, q_2, q_a, q_{a2}, q_b, q_{b2}, q_e, \mathsf{req}, \mathsf{rec}_{\mathsf{mvr}}, \mathsf{res}_{\mathsf{mvr}}, \mathsf{ack}_{\mathsf{mvr}}, \mathsf{res}_a, \mathsf{res}_b, \mathsf{ack}_a, \mathsf{ack}_b, q_r, \mathsf{rec}_{\mathsf{del}}\}, \{a,b,\#\}, \{a,b,\#\}, ¢, \$, q_0, 1, \delta_1)$ 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) $\delta_2(q_r, a) = \delta_2(q_r, b) = \delta_2(q_r, \#) = \mathsf{Restart}$.

The behaviour of the system $\mathcal{M}_{w\#w^R}$ is described as follows, where the numbers in brackets correspond to the transitions used.

1. Initially M_1 moves the 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 the 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.

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).

3. If the second symbol to the right of the ¢-symbol is not the #-symbol, then 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 #.

4. 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 (as we will see later). If both symbols are equal, then M_2 sends a message to M_1 saying that M_1 now has to delete one symbol of the first syllable (19).

5. After comparing the two symbols positioned directly before and behind the #-symbol, M_1 deletes the first symbol of the second syllable (11), applies a restart operation (12), moves the window again two steps to the right (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 applies a restart operation (20, 21). Now the computation is repeated from the first step.

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. A computation of $\mathcal{M}_{w\#w^R}$ for the input word $ab\#ba$ looks as follows:

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

□

From Theorem 24 we obtain the following lower bounds for one-way and two-way PCRA systems.

Corollary 20. *$\mathsf{NL} \subseteq \mathcal{L}(\mathsf{PC\text{-}RL})$ and $\mathsf{L} \subseteq \mathcal{L}(\mathsf{det\text{-}PC\text{-}RL})$.*

5.6.2 CSL is an upper bound for PCRA systems

Even in PC systems of restarting automata the working space is linearly restricted, that is, if a system has n components and the input is of length l, then the available space is $l \cdot n$. Thus, an upper bound for the computational power of PCRA systems is the class of all context-sensitive languages.

Corollary 21. *$\mathcal{L}(\mathsf{PC\text{-}RLWW}) \subseteq \mathsf{CSL}$.*

Proof. Let $\mathcal{M}$ be a PC-RLWW-system of degree n. An LBA P that simulates $\mathcal{M}$ uses one tape with n tracks. Any local operation of the components can easily be simulated sequentially, and the erased cells of the tape can be marked with a special symbol. The LBA P works like a product automaton, and the finite control of P simulates the finite controls of the components. Thus, a communication step in $\mathcal{M}$ is just a change of state for P. □

It is one of the most interesting open questions whether the inclusion of Corollary 21 is proper.

5.6.3 Systems of shrinking restarting automata

Usually, a rewrite step of a restarting automaton has to be length-reducing. This means that, for each rewrite transition $(q, v) \in \delta(p, u)$, $|v| < |u|$ must hold. A *shrinking restarting automaton* is a generalization of the usual restarting automaton, where the rewrite step does not necessarily have to be length-reducing but weight-reducing. Shrinking restarting automata are introduced in [JO05] and considered in more detail in [JO07].

Here, a *weight function* ω assigns a positive integer to each symbol of the tape alphabet:

$$\omega : \Gamma \to \mathbb{N}^+.$$

It is extended to words such that $\omega(\varepsilon) = 0$ and $\omega(ax) = \omega(a) + \omega(x)$, where $a \in \Gamma$ and $x \in \Gamma^*$. Since the sentinels ¢ and \$ are not allowed to be removed or to appear more than once on the tape, they do not influence the weight of the tape content. Now, a restarting automaton M is called *shrinking*, if there exists such a weight function ω such that, for each rewrite transition $(q, v) \in \delta(p, u)$ of M, $\omega(v) < \omega(u)$. Weight functions that satisfy this condition are called compatible with M. Moreover, let $\#$ be a new tape symbol, $\# \notin \Gamma$. Then we define a homomorphism $r_\omega : \Gamma \to (\Gamma \cup \{\#\})^*$ by $r_\omega(a) = a\#^{\omega(a)-1}$ for all $a \in \Gamma$ and extend it to words by $r_\omega(\varepsilon) = \varepsilon$ and $r_\omega(ax) = r_\omega(a)r_\omega(x)$. It holds that $|r_\omega(u)| = \omega(u)$ for all $u \in \Gamma^*$.

Now, we consider systems of shrinking restarting automata. Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a system of shrinking restarting automata. The weight functions of the components are denoted by $\omega^{(i)}$, $1 \leq i \leq n$, and the corresponding homomorphisms are denoted by $r_\omega^{(i)}$. Systems of (deterministic) shrinking restarting automata of type $\mathsf{X} \in \mathcal{T}$ are called of type (det-)PC-sX. The corresponding language classes are denoted by $\mathcal{L}(\text{(det-)PC-sX})$. The next theorem shows that systems of shrinking restarting automata have the same computational power as systems of length-reducing restarting automata. For this, the methods of the proofs of Theorem 19, Theorem 20, and Lemma 1 of [JO07] are combined.

Theorem 25. *For all* $\mathsf{X} \in \{\mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$, $\mathcal{L}(\textit{det-PC-sX}) = \mathcal{L}(\textit{det-PC-X})$ *and* $\mathcal{L}(\textit{PC-sX}) = \mathcal{L}(\textit{PC-X})$.

Proof. Let $\mathcal{M} = (M_1, M_2, \ldots, M_n)$ be a system of type (det-)PC-sX, $\mathsf{X} \in \{\mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$, and let $M_i = (Q_i, \Sigma, \Gamma_i, ¢, \$, q_i, k_i, \delta_i)$. Further, let $\omega^{(1)}, \omega^{(2)}, \ldots, \omega^{(n)}$ be weight functions such that $\omega^{(i)}$ is compatible with M_i, and let $\omega_{max}^{(i)} = \max\{\omega^{(i)}(a) \mid a \in \Gamma_i\}$ for all $1 \leq i \leq n$. We construct a system

$$\mathcal{M}' = (M_0, M_{1,1}, \ldots, M_{1,\omega_{max}^{(1)}+1}, \ldots, M_{n,1}, \ldots, M_{n,\omega_{max}^{(n)}+1})$$

that behaves as follows. The components $M_{i,1}, \ldots, M_{i,\omega_{max}^{(i)}+1}$ form a subsystem $\mathcal{M}_i$ for all $1 \leq i \leq n$ in the same way as we have already seen in the proofs of the closure under homomorphisms (see Theorems 19 and 20). Now, $\mathcal{M}'$ works in two phases: translation and simulation. In the first phase, M_0 reads the first symbol $a \in \Sigma$ of the input directly to the right of the $\cent$-symbol, communicates it to all subsystems $\mathcal{M}'_1, \ldots, \mathcal{M}'_n$, deletes it, and performs a restart step. The subsystems receive this symbol and translate it into $r_\omega^{(i)}(a)$, which they write on their tapes to the right of the previously written translations. Afterwards, the subsystems execute a restart operation and request the next input symbol from M_0.

When M_0 has processed the whole input, then it sends messages to all subsystems in order to start with the simulation phase. Receiving such a message, subsystem $\mathcal{M}_i$ simulates the original component M_i, $1 \leq i \leq n$. For this, the window size of the components of subsystem $\mathcal{M}_i$ is $k_i\omega_{max}^{(i)} + 1$. At the beginning of the simulation phase, the tape content of subsystem $\mathcal{M}_i$ is $r_\omega^{(i)}(w)$ for an input word w. Whenever M_i reads u and moves its window one step to the left (right), $\mathcal{M}_i$ reads $r_\omega^{(i)}(u)$ and moves the window $\omega^{(i)}(u)$ steps to the left (right). If M_i rewrites u by v, then $\mathcal{M}_i$ rewrites $r_\omega^{(i)}(u)$ by $r_\omega^{(i)}(v)$. In the cases that M_i reads u and performs a restart, a communication, or an accept step, $\mathcal{M}_i$ performs the same steps while reading $r_\omega^{(i)}(u)$.

Thus, $\mathcal{M}_i$ accepts input w in $\mathcal{M}'$ if and only if M_i accepts it in $\mathcal{M}$. Moreover, if $\mathcal{M}$ is deterministic, then $\mathcal{M}'$ is deterministic, too. Finally, since each length reducing restarting automaton is a special case of shrinking restarting automata, it results that $\mathcal{L}(\textsf{(det-)PC-sX}) = \mathcal{L}(\textsf{(det-)PC-X})$ for all $\textsf{X} \in \{\textsf{RWW}, \textsf{RRWW}, \textsf{RLWW}\}$, which completes the proof. □

5.7 Decidability

In this section some typical decision problems are investigated for language classes of PCRA systems. It will be shown that even for weak types of these systems, many of the known problems are undecidable.

In [Har67] Hartmanis used the language of valid computations of Turing machines to show some undecidability results for context-free languages in a quite short way (see also [HU79]). Now, this technique is shortly introduced and used for showing some first undecidability results for PCRA systems.

First, we describe formally the model of the Turing machine and the notation of a valid computation. Since it is well-known that the classes of languages accepted by deterministic and nondeterministic Turing machines coincide, we limit ourselves to the deterministic variant. A deterministic Turing machine T is a 7-tuple $T = (Q, \Sigma, \Gamma, \delta, q_0, \Box, F)$, where

- Q is a non-empty finite set of states,
- $\Sigma \subset \Gamma$ is a non-empty finite input alphabet,
- Γ is a non-empty finite tape alphabet,
- $q_0 \in Q$ is the initial state with $q_0 \notin F$,
- $\Box \in (\Gamma \setminus \Sigma)$ is the blank symbol that indicates an empty cell of the working tape,
- $F \subset Q$ is a set of final states not containing q_0, and
- δ is the transition mapping from $Q \times \Gamma$ into $Q \times \Gamma \times \{L, N, R\}$, where L, N, and R denote the directions for moving the read/write head (left, no move, right).

For later considerations we assume without loss of generality that the sets Q and Γ are disjoint. A *configuration* of a Turing machine is a string $uaqbv$, where $u, v \in (\Gamma \setminus \{\Box\})^*$, $a, b \in \Gamma$, and $q \in Q$. It specifies that the current tape content is $uabv$, the current state is q, and the head is placed on the b (so that b is currently read by the head). Since it is known that Turing machines with a one-way infinite tape have the same computational power as Turing machines with a two-way infinite tape, we restrict ourselves to the former one, that is, T is not allowed to perform a move left step when reading the left-most symbol. Furthermore, we require that T may not write the blank symbol. If the head is at the left-hand end of the tape, then the configuration is $\Box qbv$. If it has reached the right-hand end of the already scanned part of the tape, then the configuration is $uaq\Box$. These two situations are the only ones, where a and b could be the blank symbol, thus, if $a = \square$ ($b = \square$), then $u = \varepsilon$ ($v = \varepsilon$).

For an input $w \in \Sigma^*$, the *initial configuration* is $\square q_0 w$, whenever $w \neq \varepsilon$, otherwise, it is $\square q_0 \square$. A *final configuration* is of the form uqv, where either $u = \square$ or $u \in (\Gamma \setminus \{\square\})^+$, q is a final state, and either $v = \square$ or $v \in (\Gamma \setminus \{\square\})^+$. A *computation step* of a Turing machine T is denoted by $w_1 \vdash_T w_2$, where w_1 and w_2 are configurations such that one of the following conditions holds for some $u, v \in (\Gamma \setminus \{\square\})^*$, $a, b \in \Gamma$, $c \in \Gamma \setminus \{\square\}$, and $p, q \in Q$:

1. $w_1 = uapbv$, $w_2 = uqacv$, $u \neq \varepsilon$, $a \neq \square$, and $\delta(p, b) = (q, c, L)$;
2. $w_1 = apbv$, $w_2 = \square qacv$, $a \neq \square$, and $\delta(p, b) = (q, c, L)$;
3. $w_1 = uapbv$, $w_2 = uaqcv$, and $\delta(p, b) = (q, c, N)$;
4. $w_1 = uapbv$, $w_2 = uacqv$, $a \neq \square$, $v \neq \varepsilon$, and $\delta(p, b) = (q, c, R)$;
5. $w_1 = uapb$, $w_2 = uacq\square$, $a \neq \square$, and $\delta(p, b) = (q, c, R)$;
6. $w_1 = \square pbv$, $w_2 = cqv$, $v \neq \varepsilon$, and $\delta(p, b) = (q, c, R)$; or
7. $w_1 = \square pb$, $w_2 = cq\square$, and $\delta(p, b) = (q, c, R)$.

Observe that according to the first and the second item no move left step is allowed for $a = \square$, since the left-hand end of the tape has been reached.

The reflexive and transitive closure of $\vdash_T$ is denoted by $\vdash_T^*$. A sequence of computation steps is called a *computation* of T and it holds that $w_1 \vdash_T^* w_2$ if and only if there exists a computation of T from w_1 to w_2. Then we can define the language over an alphabet Σ that is accepted by the Turing machine T:

$$L(T) = \{w \in \Sigma^* \mid \square q_0 w \vdash_T^* w' \text{ (for } w \neq \varepsilon) \text{ or } \square q_0 \square \vdash_T^* w' \text{ (for } w = \varepsilon), \\ \text{where } w' \text{ is a final configuration}\}.$$

A computation that starts with an initial configuration and ends with a final configuration is called a *valid computation.* Based on [Har67] we assume that each valid computation consists of an even number of computation steps. That means that the Turing machine T is only allowed to accept an input after processing an even number of computation steps. For a later purpose we write a valid computation in the form

$$w_0 \# w_1^R \# w_2 \# w_3^R \# \dots \# w_{r-1}^R \# w_r \#$$

for an even r, where $\cdot^R$ is the reversal operation, that is, $(a_1 a_2 \dots a_n)^R = a_n \dots a_2 a_1$ for any string $a_1 a_2 \dots a_n$. Now, the set

$$\begin{aligned} VC(T) &= \{z \mid z \text{ is a valid computation}\} \\ &= \{z \mid \exists r > 1 : z = w_0 \# w_1^R \# w_2 \# w_3^R \# \dots \# w_{r-1}^R \# w_r \#, \\ &\qquad r \text{ is even, and } \forall 0 \leq i < r : w_i \vdash_T w_{i+1}\} \end{aligned}$$

is called the language of all valid computations of the Turing machine T.

Lemma 9. $VC(T) \in \mathcal{L}(\textsf{det-mon-PC-R}(2))$. *The language of all valid computations of a Turing machine T can be accepted by a* det-mon-PC-R*-system with two components and window size 7.*

Proof. A PCRA system $\mathcal{M}$ that accepts the language $VC(T)$ for a Turing machine T consists of two components M_1 and M_2. The first component checks whether, for all configurations w_i with an even index i, the configuration w_{i+1} is a valid successor configuration (that is, whether $w_i \vdash_T w_{i+1}$ holds). According to the above defined possible forms of a computation step of a Turing machine, the encoding of a valid computation step is as follows (here δ_T is the transition mapping of T):

1. $uapbv\#v^R caqu^R$ with $u \neq \varepsilon$, $a \neq \Box$, and $\delta_T(p,b) = (q,c,L)$;
2. $apbv\#v^R caq\Box$ with $a \neq \Box$ and $\delta_T(p,b) = (q,c,L)$;
3. $uapbv\#v^R cqau^R$ with $\delta_T(p,b) = (q,c,N)$;
4. $uapbv\#v^R qcau^R$ with $a \neq \Box$, $v \neq \varepsilon$, and $\delta_T(p,b) = (q,c,R)$;
5. $uapb\#\Box qcau^R$ with $a \neq \Box$ and $\delta_T(p,b) = (q,c,R)$;
6. $\Box pbv\#v^R qc$ with $v \neq \varepsilon$ and $\delta_T(p,b) = (q,c,R)$; or
7. $\Box pb\#\Box qc$ with $\delta_T(p,b) = (q,c,R)$.

Whether $w_i \vdash_T w_{i+1}$, $w_i\#w_{i+1}^R$ respectively, holds for two configurations w_i and w_{i+1}, can be checked quite similar like testing whether a word is included in the language $\{w\#w^R \mid w \in \{a,b\}^*\}$, which is known to be deterministic context-free, and hence, which is accepted by a det-mon-R-automaton (see [Ott06]). First, the window moves right until a free #-symbol appears in the middle of the window. 'Free' means that the symbol positioned directly to the left of # is not ¢ or # and the symbol to the right of # is not \$. Then the component checks whether the string to the left of # is of the form *apb* and in the affirmative whether the string to the right of # matches with a corresponding transition of T. To check this in one step, a window of size 7 is needed: three symbols to the left of #, the # itself, and three symbols to the right of #. If there exists no transition that causes this computation step, then M_1 halts and rejects. If the string to the left of # is not of the form *apb*, the component checks whether the symbols positioned directly before and behind the # are identical working tape symbols of T. If one of the two described cases holds, the component replaces the checked string with # and restarts, otherwise it halts and rejects. This is repeated until only # is left from $w_i\#w_{i+1}^R$. If $w_0\#w_1^R$, $w_2\#w_3^R$, …, $w_i\#w_{i+1}^R$ are encodings of valid computation steps of T, then the content of M_1's tape now begins with $\cent\#^{i+2}$, and the remaining computation steps are processed in the same manner until there is no free #-symbol left.

The second component M_2 works similar to M_1 except that it checks for all configurations w_j with an odd index j if w_{j+1} is a valid successor configuration. Therefore, M_2 always reads over the first configuration w_0 in each cycle to move to the second free #-symbol, and thereby it checks whether w_0 is an inital configuration of T, that is, if $w_0 = \Box q_0 \Box$ or $w_0 = \Box q_0 v$ for a $v \in \Sigma^+$. If it is not, then M_2 halts.

Together, M_1 and M_2 verify each computation step of the given input. If this was successful, it remains to check whether the last configuration w_r is a final configuration of T. We assumed that each valid computation of T consists of an even number of computation steps. Thus, $\cent\#^r w_r\#\$$ remains on the tape of M_1 after checking the computation steps, and while searching for the next applicable #, M_1 reaches the end of the tape, where it reads $a\#\$$ for a symbol a not equal to #. While a is not a state of T and not equal to #, $a\#\$$ is replaced by $\#\$$ followed by a restart. If $q\#\$$ is reached for some final state q, then a communication between both components takes place. Reading some non-final state or the #-symbol, then M_1 gets stuck. The component M_2 changes into a corresponding communication state if it reads $\#\#\$$ at the end of the tape after checking all computation steps.

Hence, if the input string is a valid computation of T, then both components reach the according communication states, the communication can be resolved, and M_1, and therefore the whole system $\mathcal{M}$, accepts. Since T is a deterministic Turing machine, M_1 and M_2, and hence the system $\mathcal{M}$, are deterministic.

At last there are several reasons why an input is not a valid computation of T, e.g.: 1) the #-symbol is not used in a correct way (no free #), 2) a w_i is not a valid configuration for some i (including no or more than one states), 3) $w_i\#w_{i+1}^R$ is not a valid computation step, or 4) the first and the last configuration are not an initial configuration and a final configuration, respectively. Then, depending on the location where the error appears, M_1 or M_2 gets stuck during the computation, the final communication does not happen, M_1 does not reach the accepting configuration, and $\mathcal{M}$ rejects the input. □

It follows from Theorem 15 that the complement of $VC(T)$, namely $\overline{VC(T)}$, is also accepted by a det-mon-PC-R-system. Moreover, we can easily argument that only two components suffice for this, too.

Corollary 22. $\overline{VC(T)} \in \mathcal{L}(\textsf{\textit{det-mon-PC-R}}(2))$.

Proof. To construct a $\mathsf{det\text{-}mon\text{-}PC\text{-}R}(2)$-system $\mathcal{M}' = (M_1', M_2')$ that accepts $\overline{VC(T)}$ we modify the system $\mathcal{M} = (M_1, M_2)$ that accepts $VC(T)$ from the proof of Lemma 9 as follows: 1) delete any transition of M_1 that maps into Accept (M_2 does not have such transitions), 2) add accepting transitions to M_1 and M_2 for each non-defined situation except those situations, where M_2 is in a communication state. Thus, if $\mathcal{M}$ accepts, then M_1 accepts, and therefore, M_1', M_2', and $\mathcal{M}'$ do not accept. If $\mathcal{M}$ does not accept, then M_1 or M_2 (or both) do not reach the final communication, because they got stuck somewhere. Hence, M_1' or M_2', and therefore $\mathcal{M}'$, accept. In particular, $\mathcal{M}$ and therefore $\mathcal{M}'$ cannot reach a communication loop (at most one communication can be executed) or a loop in their local computation (no MVL or SCO steps are used). All in all, $\mathcal{M}'$ accepts $\overline{VC(T)}$, and thus, $\overline{VC(T)} \in \mathcal{L}(\mathsf{det\text{-}mon\text{-}PC\text{-}R}(2))$. □

Now we can use the languages $VC(T)$, $\overline{VC(T)}$, and the fact that they are accepted by $\mathsf{det\text{-}mon\text{-}PC\text{-}R}(2)$-systems to prove some undecidability results for the language class $\mathcal{L}(\mathsf{det\text{-}mon\text{-}PC\text{-}R})$.

Theorem 26 (Undecidabilities). *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two arbitrary det-mon-PC-R-systems with input alphabet Σ, and let R be a regular language. Then, the following problems are undecidable:*

1. *$L(\mathcal{M}_1) = \emptyset$?*
2. *$L(\mathcal{M}_1) = \Sigma^*$?*
3. *$L(\mathcal{M}_1) = L(\mathcal{M}_2)$?*
4. *$L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$?*
5. *$R = L(\mathcal{M}_1)$?*
6. *$R \subseteq L(\mathcal{M}_1)$?*
7. *Is $L(\mathcal{M}_1)$ finite?*

Proof. Let $T = (Q, \Sigma_T, \Gamma, \delta, q_0, B, F)$ be an arbitrary (deterministic) Turing machine. Then $L(T)$ is empty for a Turing machine T if and only if there exists no valid computation for T, that is,

$$L(T) = \emptyset \quad \Leftrightarrow \quad VC(T) = \emptyset \quad \Leftrightarrow \quad \overline{VC(T)} = (Q \,\dot\cup\, \Gamma \,\dot\cup\, \{\#\})^*.$$

1. The first result follows immediately from the well-known fact that the emptiness problem for recursively enumerable languages is undecidable.

2. Assume that this problem is decidable. Then it could be decided whether, for a system $\mathcal{M}$ with $L(\mathcal{M}) = \overline{VC(T)}$, $L(\mathcal{M}) = \Sigma^*$ or not. Because of the equivalences above, the emptiness problem for Turing machines could then be decided. This is a contradiction, since the emptiness problem is not decidable for Turing machines.

3. Assume that this is decidable. Then we construct the system $\mathcal{M}_2$ so that $L(\mathcal{M}_2) = \emptyset$ and decide whether $L(\mathcal{M}_1) = L(\mathcal{M}_2) = \emptyset$. This contradicts 1.

4. If this was decidable, then we could decide whether both $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$ and $L(\mathcal{M}_2) \subseteq L(\mathcal{M}_1)$ hold. Thus, we could decide $L(\mathcal{M}_1) = L(\mathcal{M}_2)$ which contradicts 3.

5. Take $R = \emptyset$ and prove this by reduction from 1.

6. Take $R = \Sigma^*$ and prove this by reduction from 2.

7. Let this be decidable. Then it would in particular be decidable whether $VC(T)$ is finite for an arbitrary deterministic Turing machine T. Since for every word $w \in L(T)$ there exists exactly one computation of a deterministic Turing machine, it holds that $VC(T)$ is finite if and only if $L(T)$ is finite. Thus, the question of whether a language accepted by an arbitrary deterministic Turing machine is finite, would be decidable. This is a contradiction, as it is well-known that it is undecidable whether a language accepted by a (deterministic) Turing machine is finite or not. Hence, this problem cannot be decidable for det-mon-PC-R-systems, either. □

Observe that all the questions of Theorem 26 are not even semi-decidable, since the emptiness problem is known to be not semi-decidable for Turing machines.

Corollary 23. *Theorem 26 also holds for (det-)(mon-)PC-R(R)(W)(W)-systems.*

Proof. $\mathcal{L}$(det-mon-PC-R) is a subset of all the language classes characterized by these systems. Hence, Lemma 9 and with this Theorem 26 as well can be directly applied to (det-)(mon-)PC-R(R)(W)(W)-systems. □

Another important decision question is the membership problem. For any given PC-RLWW-system $\mathcal{M}$ and any input word w, the question 'Does $\mathcal{M}$ accept w?' should be answered with yes or no. Although we know that the space used by our PCRA systems is linearly bounded, and this implies an exponential time-bound [Pap94], we give a more detailed proof for the decidability of the word problem in exponential time, using the same typical combinatorial argument as for linear bounded automata (see e.g. [Sip06]).

Theorem 27 (Membership problem). *Let $\mathcal{M}$ be an arbitrary PC-RLWW-system and w an input word. Then it is decidable in exponential time whether w is in $L(\mathcal{M})$ or not.*

Proof. Here we can use a combinatorial argument. For $s = |Q|$ states, $t = |\Gamma|$ tape symbols, and a length l of the tape content excluding ¢ and \$, there exist $s \cdot t^l \cdot (l + 2)$ different configurations a component can be in. Concerning the shortening of the tape, there are at most

$$r = \sum_{i=0}^{l} (s \cdot t^i \cdot (i + 2))$$

different configurations a component can reach during a computation. If we assume that s and t are the maximal numbers of states and tape symbols over all components, then there exist at most r^n different configurations for a computation of a system of degree n. Therefore, each computation

$$K_0 \vdash^*_{\mathcal{M}} K$$

that is longer than $r^n - 1$ steps must contain a loop, where at least one configuration is passed twice:

$$K_0 \vdash^*_{\mathcal{M}} K' \vdash^*_{\mathcal{M}} K' \vdash^*_{\mathcal{M}} K.$$

But in this case there exists a computation without the loop and hence shorter than r^n. So, for each word $w \in L(\mathcal{M})$, there exists an accepting computation with at most $r^n - 1$ computation steps. To decide whether $\mathcal{M}$ accepts the input, all (finitely many) possible computations that are shorter than r^n can be tested. □

From [JLNO04] we know that there exists a single RWW-automaton accepting the NP-complete language L_{3SAT}. Therefore, we can conclude the following: if the membership problem for R(R)WW-automata is solvable in polynomial time, then the equality „P=NP“ holds, which is still an open problem. Thus, it seems improbable that this decision problem can be solved in polynomial time for PC-R(R)WW-systems and PC-RLWW-systems.

Theorem 28 (Membership problem for deterministic systems). *Let $\mathcal{M}$ be an arbitrary det-PC-RLWW-system and w an input word. Then, it is decidable in quadratic time whether w is in $L(\mathcal{M})$ or not.*

Proof. A computation of a single deterministic[7] one-way restarting automaton without SCO transitions takes at most about l^2 steps for an input w of length l.

[7] This holds as well for a shortest accepting computation of a single nondeterministic automaton.

Due to the length reducing property in each cycle at least one symbol has to be deleted, and thus, a computation can include at most l cycles. When $\text{¢}a_1a_2\dots a_r\$$ is the content of the tape, then the maximal length of a cycle is $r+2$ steps:

$$\overset{\downarrow}{\text{¢}}a_1a_2\dots a_r\$ \xrightarrow[\text{MVR-steps}]{\text{after } r} \text{¢}a_1a_2\dots\overset{\downarrow}{a_r}\$ \xrightarrow{\text{rewrite}} \text{¢}a_1a_2\dots a_{r-1}\overset{\downarrow}{\$} \xrightarrow{\text{restart}} \overset{\downarrow}{\text{¢}}a_1a_2\dots a_{r-1}\$.$$

The downward arrows mark the positions of the window. Of course, the rewrite step has not to be performed at the end of the tape, but could be done anywhere on the tape as long as the ¢-symbol and the \$-symbol are not removed. Since we are interested in the longest possible accepting computation for a given input word, we assume that only one symbol is removed in the rewrite step, although in general k symbols can be removed with a window of size k. If $r=0$ ($\text{¢}\$$ is the tape content), then this is a tail of the computation, and at most two steps can be done, one MVR step and an accepting step. All in all for the length s of any computation of a single one-way restarting automaton without SCO transitions it holds that

$$s \le \sum_{r=0}^{l}(r+2) = 2(l+1) + \sum_{r=1}^{l} r = \frac{l^2+5l+4}{2}.$$

For deterministic one-way restarting automata with SCO transitions and for deterministic two-way restarting automata, there exist $|Q|\cdot(r+2)$ many different configurations for a set of states Q and a tape content of length r (excluding ¢ and \$). Thus, in each cycle the automaton can execute at most $|Q|\cdot(r+2)-1$ computation steps without reaching a loop. Moreover, the restart step has to be added. For the whole computation of the automaton we then have

$$s \le \sum_{r=0}^{l}|Q|(r+2) = |Q|\frac{l^2+5l+4}{2}.$$

With this inequation we also have an upper bound for the maximum number of different configurations that can be reached by a component within a system. In the further considerations we are interested in the largest such s over all components, that is,

$$s \le \left(\max_{1\le i\le n}|Q_i|\right)\frac{l^2+5l+4}{2},$$

where Q_i is the set of states of the i-th component. Moreover, the length of an accepting computation of the system is the length of the shortest accepting computation of any component. Indeed, this length is not only determined by the number of local computation steps but also by the number of communication steps. Observe that the length of the computation of a component can be longer than the number of different configurations it can reach. This stems from the fact that a component may have to wait in a communication situation, where it keeps the current configuration, while the computation of the system goes on. Thus, besides the number of local computation steps, we have to consider two more factors due to communication: 1) the degree of parallelism and 2) the number of communication steps.

The degree of parallelism can be seen as a measure for the number of local computation steps that are executed in parallel. We would expect that a high degree of parallelism results in a shorter computation of the system than a lower degree of parallelism and a more sequential computation. In fact, the degree of parallelism is closely related to the number of waiting steps of the components (computation steps of the form $\kappa \vdash \kappa$). Consider the following two extremes. If no communication takes place during the whole computation, then all components work in parallel, and the length of the computation of the system is at most s. On the other hand, if the communication leads to a highly sequential computation, that is, there is only one working component at a point of time, and all other components are waiting for communication answers, then the length of the computation is at most $n \cdot s$ plus the number of communication steps, where n is the degree of the system.

The number of different configurations containing a communication state is $|\,\mathsf{COM}(M)|$ for a component M with a fixed window content and window position. If we fix the content of all tapes and the positions of all windows, then we obtain

$$t = |\,\mathsf{COM}(M_1)| \cdot |\,\mathsf{COM}(M_2)| \cdot \ldots \cdot |\,\mathsf{COM}(M_n)| = \prod_{i=1}^{n} |\,\mathsf{COM}(M_i)|$$

different system configurations containing only communication states. Thus, we know that if a system executes more than t communication steps without any local step of any component in between, then the system reaches a previous configuration twice, and therefore, it is in a communication loop. Hence, after at most t communication steps at least one local computation step must follow in each accepting computation.

To summarize, the length of an accepting computation of a det-PC-RLWW-system of degree n for a given input word of length l is at most $t \cdot n \cdot s \in \mathcal{O}(l^2)$. To decide whether a deterministic system accepts a particular input or not, the system only has to be simulated on the input for at most $t \cdot n \cdot s$ computation steps. If the system has not halted, then it is either in a communication loop or in a loop of a local computation caused by SCO steps or the combination of MVR and MVL steps. In this case it does not accept the input. □

6 Conclusions

The aim of this thesis was to combine the model of the restarting automaton with the notion of parallel communicating systems and to investigate various properties of the resulting systems, e.g. properties of the communication structure, closure properties, computational power, and decidability questions. For this, parallel communicating restarting automata systems were defined, and several examples were given. Then, it was shown that centralized systems are as powerful as non-centralized systems for each type of restarting automaton as components. This result differs from those for other parallel communicating automata and grammar systems, where centralization decreases the computational power in various cases. Afterwards, the nonforgetting property was considered, and it was shown that this property does not yield an increase in computational power in contrast to the situation for individual restarting automata. Additionally, closure properties of language classes of these systems were established, and it emerged that the considered language classes are somehow robust in the sense that they are closed under most of the usual operations. Three of the language classes, namely $\mathcal{L}(\mathsf{PC}\text{-}\mathsf{RWW})$, $\mathcal{L}(\mathsf{PC}\text{-}\mathsf{RRWW})$, and $\mathcal{L}(\mathsf{PC}\text{-}\mathsf{RLWW})$ are so-called AFLs (abstract families of languages).

Further, the computational power of PCRA systems was investigated and compared with those of individual restarting automata, with those of parallel communicating finite automata systems, with those of one-way and two-way multi-head finite automata, with language classes of the Chomsky-hierarchy, and with other well-known language classes. It turned out that even PCRA systems of the weakest type, i.e. $\mathsf{PC}\text{-}\mathsf{R}$-systems with two components, are more powerful than the individual automata in the case of restarting automata without auxiliary symbols. Individual restarting automata with auxiliary symbols can be simulated by PCRA systems with two components of the same type without auxiliary symbols. This result was generalized in the following way: each PCRA system with n components and with auxiliary symbols can be simulated by a PCRA system with $n+1$ components and without auxiliary symbols. Moreover, it was shown that systems of $\mathsf{PC}\text{-}\mathsf{R}$-automata with window size one are strictly more powerful than one-way multi-head finite automata, and thus, than PCFA systems, although the components for themselves have the same computational power, i.e. they accept the regular languages. The comparison with two-way multi-head finite automata leads to the lower bounds L and NL for deterministic and nondeterministic PCRA systems with the weakest two-way components, i.e. RL-automata. An upper bound for all types of PCRA systems is clearly CSL. For systems of shrinking restarting automata it could be shown that they have the same computational power as systems of length-reducing restarting automata.

Then, decidability questions were considered for PCRA systems, and unfortunately, it turned out that most of the interesting problems are undecidable even for systems of two components of the weakest type, i.e. det-PC-R(2)-systems: emptiness, universality, finiteness, containment, and equality (even with a regular language). The membership problem is decidable in quadratic time for deterministic systems and in exponential time for nondeterministic systems.

Although interesting results were established in this work, a lot of open questions and approaches for further research are left. For example, we could not find any language that cannot be accepted by a PCRA system. This may result from the fact that already systems with few weak components can accept quite complicated languages that are not even growing context sensitive (like the copy language or the Gladkij language) and not even semi-linear (e.g. exponential language). In this context it would also be interesting to know whether there exist strict hierarchies with respect to the number of components for the different types of PCRA systems or whether there is a fixpoint such that more components do not yield an increase in computational power. Maybe the following language representation can be helpful to show such an infinite hierarchy at least for weak types of PCRA systems: consider alphabets of the form

$$\Sigma_r = \left\{ \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_r \end{pmatrix} \mid x_i \in \{a, b\} \text{ for all } i = 1, \ldots, r \right\}$$

and the Dyck language D, which consists of all words over the alphabet $\{a, b\}$ that can be generated by the rules $S \to \varepsilon \mid SS \mid aSb$ of a context-free grammar. Now, we define the following languages:

$$L_r = \left\{ \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_r \end{pmatrix} \in \Sigma_r^* \mid w_i \in D \text{ for all } i = 1, \ldots, r \right\},$$

where the concatenation of two vectors is intuitively the componentwise concatenation. The language L_r can in fact be accepted by a PCRA system of degree r and of any type, since L_r is a finite intersection of deterministic context-free languages for any r. The assumption is that L_r cannot be accepted by any PCRA system with $r-1$ components at least without auxiliary symbols. Whenever a component begins to try to check whether an entry of the input vector is contained in D, then it must rewrite some infix ab (of the corresponding entry of the vector). But then it changes the other entries, and it cannot check anymore whether any of the other vector entries are words of D or not. Thus, each component can check at most one entry of a vector.

Moreover, the closure of some language classes characterized by PCRA systems under some operations is still open, e.g. product, Kleene closure, positive closure, and non-erasing homomorphisms for deterministic two-way components. We can expect that some more negative results can be achieved in this context when we have more knowledge about languages that cannot be accepted by PCRA systems of a special type.

Another open question is, on the one hand, whether systems of the most general type, i.e. PC-RLWW-systems, are strictly more powerful than the individual automata, and, on the other hand, whether $\mathcal{L}$(PC-RLWW) is properly included within the class of context-sensitive languages. Particularly, this is an interesting question, since it is not even known whether $\mathcal{L}$(RLWW) is a proper subset of CSL or not.

Comparing det-PC-R-systems in which each component has window size one with PCFA systems, the former ones are strictly more powerful than the latter ones, although the individual components (det-R-automata with window size one vs. finite automata) have the same computational power, i.e. they recognize the regular languages. Thus, it seems that the communication protocol of PCRA systems is somehow stronger than that of PCFA systems. Therefore, it would be interesting to know what happens if one carries over our communication protocol to systems of finite automata. Surprisingly, we can assume that a nonsynchronized PC system of finite automata, which communicate by request and response states, can be simulated by a multi-head finite automaton and vice versa. Hence, both different communication protocols lead to the same computational power for PC systems of finite automata. For PC systems of pushdown automata a similar communication protocol was applied in [Ott13]. There asynchronous PC systems of pushdown automata are considered that use request symbols as particular pushdown symbols. It turns out that the computational power of these systems differs from that of the original PCPA systems.

Most of the typical decision problems were shown to be undecidable even for the weakest PCRA systems. Hence, it seems reasonable to consider more restricted variants, e.g. systems with a restricted number of communications within a cycle or within the computation, a restricted window size, etc. However, a counterargument for this is that some of the typical problems are not even decidable for (still weaker) DRCPCFA-systems (that are the weakest PCFA systems). Considering the examples in the context of restricted communication, one can observe that for some languages only a constant number of communications was sufficient (e.g. Gladkij language), while for other languages linearly many communication steps were needed (e.g. copy language).

Moreover, in many proofs the constructed systems consist of more components than the underlying systems. Here the question arises whether the results also hold without increasing the number of components. In this context, results on the previously mentioned hierarchy could be helpful.

Another direction for further investigation is the question of *how* are languages accepted by PCRA systems instead of *what* languages can be accepted. On the one hand, this leads to the research field of derivation languages (also called Szilard languages) that are of importance within linguistic topics. On the other hand, this may give some results about the efficiency of PCRA systems, asking how many computation steps are needed to accept specific languages. Moreover, this can be compared with other formal language devices. Consider, for instance, the copy language with middle marker. A linear bounded automaton needs at least quadratically many computation steps to accept this language (according to the length of the input word), whereas the defined PCRA system only needs linearly many steps.

When defining the PCRA systems, a communication protocol was chosen that differs from those of other PC systems in the sense that there is no implicit synchronisation (no global clock), and the components work absolutely independently of each other between two communication steps. Therefore, these systems are appropriate for distributed and concurrent applications. It should be investigated whether the basic linguistic motivation (analysis by reduction) can be extended in a parallel and distributed manner (e.g. checking several syntactic properties in parallel).

A more theoretical application of PCRA systems is the characterization of relations and the computation of transductions initiated in [HOV11]. One can simply imagine that systems of two components can be modified in a way such that one component contains the input and the other component contains the output. Now, the system accepts if and only if the pair of input and output belongs to a specified relation. Of course, this idea can be extended to systems that deal with k-ary relations instead of only binary relations by using k components.

Bibliography

[Bal09] M. Sakthi Balan. Serializing the Parallelism in Parallel Communicating Pushdown Automata Systems. In Jürgen Dassow, Giovanni Pighizzini, and Bianca Truthe, editors, *Proceedings Eleventh International Workshop on Descriptional Complexity of Formal Systems (DCFS 2009)*, volume 3 of *Electronic Proceedings in Theoretical Computer Science*, pages 59–68, 2009.

[BCCC96] Luca Breveglieri, Alessandra Cherubini, Claudio Citrini, and Stefano Crespi-Reghizzi. Multi-Push-Down Languages and Grammars. *International Journal of Foundations of Computer Science*, 7(3):253–292, 1996.

[BKM03] M. Sakthi Balan, Kamala Krithivasan, and Mutyam Madhu. Some Variants in Communication of Parallel Communicating Pushdown Automata. *Journal of Automata, Languages and Combinatorics*, 8(3):401–416, 2003.

[BKM08] Henning Bordihn, Martin Kutrib, and Andreas Malcher. On the Computational Capacity of Parallel Communicating Finite Automata. In Masami Ito and Masafumi Toyama, editors, *Developments in Language Theory*, volume 5257 of *Lecture Notes in Computer Science*, pages 146–157. Springer Berlin / Heidelberg, 2008.

[BKM10] Henning Bordihn, Martin Kutrib, and Andreas Malcher. Undecidability and Hierarchy Results for Parallel Communicating Finite Automata. In Yuan Gao, Hanlin Lu, Shinnosuke Seki, and Sheng Yu, editors, *Developments in Language Theory*, volume 6224 of *Lecture Notes in Computer Science*, pages 88–99. Springer Berlin / Heidelberg, 2010.

[BKM11] Henning Bordihn, Martin Kutrib, and Andreas Malcher. Undecidability and Hierarchy Results for Parallel Communicating Finite Automata. *International Journal of Foundations of Computer Science*, 22(7):1577–1592, 2011.

[Bud87] Anatoli O. Buda. Multiprocessor automata. *Information Processing Letters*, 25(4):257–261, 1987.

[CC05] Elena Czeizler and Eugen Czeizler. Parallel Communicating Watson-Crick Automata Systems. In Zoltán Ésik and Zoltán Fülöp, editors, *Automata and Formal Languages, 11th International Conference, AFL 2005, Dobogókő, Hungary, May 17-20*, pages 83–96. Institute of Informatics, University of Szeged, 2005.

[CC06a] Elena Czeizler and Eugen Czeizler. A Short Survey on Watson-Crick Automata. *Bulletin of the EATCS*, 88:104–119, 2006.

[CC06b] Elena Czeizler and Eugen Czeizler. On the power of parallel communicating Watson-Crick automata systems. *Theoretical Computer Science*, 358(1):142–147, 2006.

[CC06c] Elena Czeizler and Eugen Czeizler. Parallel Communicating Watson-Crick Automata Systems. *Acta Cybernetica*, 17(4):685–700, 2006.

[CCKS09] Elena Czeizler, Eugen Czeizler, Lila Kari, and Kai Salomaa. On the descriptional complexity of Watson-Crick automata. *Theoretical Computer Science*, 410(35):3250–3260, 2009.

[CDK05] George Coulouris, Jean Dollimore, and Tim Kindberg. *Verteilte Systeme. Konzepte und Design.* Pearson Studium, Munich, 3rd revised edition, 2005.

[CDKP94] Erzsébet Csuhaj-Varjú, Jürgen Dassow, Josef Kelemen, and Gheorghe Păun. *Grammar Systems: A Grammatical Approach to Distribution and Cooperation.* Gordon and Breach Science Publishers, Inc., Newark, NJ, USA, 1994.

[CKKP94] Erzsébet Csuhaj-Varjú, Josef Kelemen, Alica Kelemenová, and Gheorghe Păun. Eco(Grammar) Systems. A Preview. In Robert Trappl, editor, *Proceedings 12th European Meeting on Cybernetics and Systems Research*, pages 941–948, Singapore, 1994. World Scientific.

[CKKP97] Erzsébet Csuhaj-Varjú, Josef Kelemen, Alica Kelemenová, and Gheorghe Păun. Eco-Grammar Systems: A Grammatical Framework for Studying Lifelike Interactions. *Artificial Life*, 3(1):1–28, 1997.

[CKM07] Ashish Choudhary, Kamala Krithivasan, and Victor Mitrana. Returning and non-returning parallel communicating finite automata are equivalent. *Informatique Théorique et Applications*, 41(2):137–145, 2007.

[ČM10] Peter Černo and František Mráz. Clearing Restarting Automata. *Fundamenta Informaticae*, 104(1-2):17–54, 2010.

[ČM11] Peter Černo and František Mráz. Delta-Clearing Restarting Automata and CFL. In *Proceedings of the 15th international conference on Developments in language theory*, DLT'11, pages 153–164, Berlin, Heidelberg, 2011. Springer-Verlag.

[CMMV00] Erzsébet Csuhaj-Varjú, Carlos Martín-Vide, Victor Mitrana, and György Vaszil. Parallel Communicating Pushdown Automata Systems. *International Journal of Foundations of Computer Science*, 11(4):631–650, 2000.

[CS97] Erzsébet Csuhaj-Varjú and Arto Salomaa. Networks of Parallel Language Processors. In Gheorghe Păun and Arto Salomaa, editors, *New Trends in Formal Languages*, volume 1218 of *Lecture Notes in Computer Science*, pages 299–318. Springer Berlin / Heidelberg, 1997.

[CV] Erzsébet Csuhaj-Varjú and György Vaszil. http://www.sztaki.hu/mms/pcbib.html, 09.08.2012.

[ĎJKL98] Pavol Ďuriš, Tomasz Jurdziński, Mirosław Kutyłowski, and Krzysztof Loryś. Power of Cooperation and Multihead Finite Systems. In Kim Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 896–907. Springer Berlin / Heidelberg, 1998.

[DW86] Elias Dahlhaus and Manfred Warmuth. Membership for Growing Context Sensitive Grammars is Polynomial. *Journal of Computer and System Sciences*, 33(3):456–472, 1986.

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962.

[Fou94] Terry J. Fountain. *Parallel Computing: Principles and Practice.* Cambridge University Press, 1994.

[FPRS97] Rudolf Freund, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. Watson-Crick Finite Automata. In Harvey Rubin and David Harlan Wood, editors, *DNA Based Computers III*, volume 48 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 297–328, Philadelphia, 1997. American Mathematical Society.

[GMP07] Viliam Geffert, Carlo Mereghetti, and Giovanni Pighizzini. Complementing two-way finite automata. *Information and Computation*, 205(8):1173–1187, 2007.

[Goe09] Marcel Goehring. PC-Systems of Restarting Automata. In Jöran Mielke, Ludwig Staiger, and Renate Winter, editors, *19. Theorietag Automaten und Formale Sprachen 2009*, pages 26–27, Universität Halle-Wittenberg, Institut für Informatik, 2009. Technical Report 2009/03.

[Har67] Juris Hartmanis. Contex-free Languages and Turing Machine Computations. In Jacob T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics: Mathematical Aspects of Computer Science*, volume 19, pages 42–51. American Mathematical Society, 1967.

[Har72] Juris Hartmanis. On Non-Determinancy in Simple Computing Devices. *Acta Informatica*, 1(4):336–344, 1972.

[Har78] Michael A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[Hay94] Simon S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1994.

[HI68] Michael A. Harrison and Oscar H. Ibarra. Multi-Tape and Multi-Head Pushdown Automata. *Information and Control*, 13(5):433–470, 1968.

[HKM09] Markus Holzer, Martin Kutrib, and Andreas Malcher. Multi-Head Finite Automata: Characterizations, Concepts and Open Problems. In Turlough Neary, Damien Woods, Tony Seda, and Niall Murphy, editors, *Proceedings International Workshop on The Complexity of Simple Programs (CSP 2008)*, volume 1 of *Electronic Proceedings in Theoretical Computer Science*, pages 93–107, 2009.

[HKM11] Markus Holzer, Martin Kutrib, and Andreas Malcher. Complexity of multi-head finite automata: Origins and directions. *Theoretical Computer Science*, 412(12):83–96, 2011.

[HOV11] Norbert Hundeshagen, Friedrich Otto, and Marcel Vollweiler. Transductions Computed by PC-Systems of Monotone Deterministic

Restarting Automata. In Michael Domaratzki and Kai Salomaa, editors, *Implementation and Application of Automata*, volume 6482 of *Lecture Notes in Computer Science*, pages 163–172. Springer Berlin / Heidelberg, 2011.

[Hro97] Juraj Hromkovič. *Communication Complexity and Parallel Computing.* Springer, 1997.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Iba73] Oscar H. Ibarra. On Two-Way Multihead Automata. *Journal of Computer and System Sciences*, 7(1):28–36, 1973.

[JLNO04] Tomasz Jurdziński, Krzysztof Loryś, Gundula Niemann, and Friedrich Otto. Some Results on RWW- and RRWW-Automata and their Relation to the Class of Growing Context-Sensitive Languages. *Journal of Automata, Languages and Combinatorics*, 9(4):407–437, 2004.

[JMP93a] Petr Jančar, František Mráz, and Martin Plátek. A Taxonomy of Forgetting Automata. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 527–536. Springer Berlin / Heidelberg, 1993.

[JMP93b] Petr Jančar, František Mráz, and Martin Plátek. Forgetting automata and the Chomsky hierarchy. In *Proceedings of SOFSEM '93*, pages 41–44, 1993.

[JMP$^+$95] Petr Jančar, František Mráz, Martin Plátek, Martin Procházka, and Jörg Vogel. Restarting Automata, Marcus Grammars and Context-Free Languages. In *Developments in Language Theory*, pages 102–111, 1995.

[JMP$^+$97] Petr Jančar, František Mráz, Martin Plátek, Martin Procházka, and Jörg Vogel. Deleting Automata with a Restart Operation. In Symeon Bozapalidis, editor, *Developments in Language Theory*, pages 191–202. Aristotle University of Thessaloniki, 1997.

[JMPV95] Petr Jančar, František Mráz, Martin Plátek, and Jörg Vogel. Restarting Automata. In Horst Reichel, editor, *Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 283–292. Springer Berlin / Heidelberg, 1995.

[JMPV97] Petr Jančar, František Mráz, Martin Plátek, and Jörg Vogel. On Restarting Automata with Rewriting. In Gheorghe Păun and Arto Salomaa, editors, *New Trends in Formal Languages*, volume 1218 of *Lecture Notes in Computer Science*, pages 119–136. Springer Berlin / Heidelberg, 1997.

[JMPV98] Petr Jančar, František Mráz, Martin Plátek, and Jörg Vogel. Different Types of Monotonicity for Restarting Automata. In Vikraman Arvind and Sundar Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 343–355. Springer Berlin / Heidelberg, 1998.

[JMPV99] Petr Jančar, František Mráz, Martin Plátek, and Jörg Vogel. On Monotonic Automata with a Restart Operation. *Journal of Automata, Languages and Combinatorics*, 4(4):287–312, 1999.

[JMPV07] Petr Jančar, František Mráz, Martin Plátek, and Jörg Vogel. Monotonicity of Restarting Automata. *Journal of Automata, Languages and Combinatorics*, 12(3):355–371, 2007.

[JO05] Tomasz Jurdziński and Friedrich Otto. Shrinking Restarting Automata. In Joanna Jedrzejowicz and Andrzej Szepietowski, editors, *Mathematical Foundations of Computer Science 2005*, volume 3618 of *Lecture Notes in Computer Science*, pages 532–543. Springer Berlin / Heidelberg, 2005.

[JO06a] Tomasz Jurdziński and Friedrich Otto. Restarting automata with restricted utilization of auxiliary symbols. *Theoretical Computer Science*, 363(2):162–181, 2006.

[JO06b] Tomasz Jurdziński and Friedrich Otto. Restricting the Use of Auxiliary Symbols for Restarting Automata. In Jacques Farré, Igor Litovsky, and Sylvain Schmitz, editors, *Implementation and Application of Automata*, volume 3845 of *Lecture Notes in Computer Science*, pages 176–187. Springer Berlin / Heidelberg, 2006.

[JO07] Tomasz Jurdziński and Friedrich Otto. Shrinking Restarting Automata. *International Journal of Foundations of Computer Science*, 18(2):361–385, 2007.

[JOMP05a] Tomasz Jurdziński, Friedrich Otto, František Mráz, and Martin Plátek. On Left-Monotone Deterministic Restarting Automata. In

Cristian Calude, Elena Calude, and Michael Dinneen, editors, *Developments in Language Theory*, volume 3340 of *Lecture Notes in Computer Science*, pages 249–260. Springer Berlin / Heidelberg, 2005.

[JOMP05b] Tomasz Jurdziński, Friedrich Otto, František Mráz, and Martin Plátek. On the Complexity of 2-Monotone Restarting Automata. In Cristian Calude, Elena Calude, and Michael Dinneen, editors, *Developments in Language Theory*, volume 3340 of *Lecture Notes in Computer Science*, pages 237–248. Springer Berlin / Heidelberg, 2005.

[JOMP08] Tomasz Jurdziński, Friedrich Otto, František Mráz, and Martin Plátek. On the Complexity of 2-Monotone Restarting Automata. *Theory of Computing Systems*, 42(4):488–518, 2008.

[KMO09] Martin Kutrib, Hartmut Messerschmidt, and Friedrich Otto. On Stateless Deterministic Restarting Automata. In Mogens Nielsen, Antonín Kucera, Peter Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank Valencia, editors, *SOFSEM 2009: Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 353–364. Springer Berlin / Heidelberg, 2009.

[KR08] Martin Kutrib and Jens Reimann. Succinct description of regular languages by weak restarting automata. *Information and Computation*, 206(9-10):1152–1160, 2008.

[KW05] Dietrich Kuske and Peter Weigel. The Role of the Complementarity Relation in Watson-Crick Automata and Sticker Systems. In Cristian Calude, Elena Calude, and Michael Dinneen, editors, *Developments in Language Theory*, volume 3340 of *Lecture Notes in Computer Science*, pages 272–283. Springer Berlin / Heidelberg, 2005.

[LN10] Peter Leupold and Benedek Nagy. $5' \to 3'$ Watson-Crick Automata With Several Runs. *Fundamenta Informaticae*, 104(1-2):71–91, 2010.

[Mes07] Hartmut Messerschmidt. *CD-Systems of Restarting Automata.* PhD thesis, Universität Kassel, 2007.

[Mes08] Hartmut Messerschmidt. On Nonforgetting Restarting Automata That Are Deterministic and/or Monotone. In Markus Holzer, Martin Kutrib, and Andreas Malcher, editors, *18. Theorietag Automaten und Formale Sprachen 2008*, pages 87–91, Universität Giessen, Institut für Informatik, 2008.

[MM00] Carlos Martín-Vide and Victor Mitrana. Parallel communicating automata systems - A survey. *Journal of Applied Mathematics and Computing*, 7(2):237–257, 2000.

[MM01] Carlos Martín-Vide and Victor Mitrana. Some undecidable problems for parallel communicating finite automata systems. *Information Processing Letters*, 77(5-6):239–245, 2001.

[MMM02] Carlos Martín-Vide, Alexandru Mateescu, and Victor Mitrana. Parallel Finite Automata Systems Communicating by States. *International Journal of Foundations of Computer Science*, 13(5):733–749, 2002.

[MNO88] Robert McNaughton, Paliath Narendran, and Friedrich Otto. Church-Rosser Thue Systems and Formal Languages. *Journal of the Association for Computing Machinery*, 35(2):324–344, 1988.

[MO06] Hartmut Messerschmidt and Friedrich Otto. On Nonforgetting Restarting Automata That Are Deterministic and/or Monotone. In Dima Grigoriev, John Harrison, and Edward Hirsch, editors, *Computer Science - Theory and Applications*, volume 3967 of *Lecture Notes in Computer Science*, pages 247–258. Springer Berlin / Heidelberg, 2006.

[MO07a] Hartmut Messerschmidt and Friedrich Otto. Cooperating Distributed Systems of Restarting Automata. *International Journal of Foundations of Computer Science*, 18(6):1333–1342, 2007.

[MO07b] Hartmut Messerschmidt and Friedrich Otto. Strictly Deterministic CD-Systems of Restarting Automata. In Erzsébet Csuhaj-Varjú and Zoltán Ésik, editors, *Fundamentals of Computation Theory*, volume 4639 of *Lecture Notes in Computer Science*, pages 424–434. 2007.

[MO09] Hartmut Messerschmidt and Friedrich Otto. On Deterministic CD-Systems of Restarting Automata. *International Journal of Foundations of Computer Science*, 20(1):185–209, 2009.

[MO11] Hartmut Messerschmidt and Friedrich Otto. A Hierarchy of Monotone Deterministic Non-Forgetting Restarting Automata. *Theory of Computing Systems*, 48(2):343–373, 2011.

[Mon80] Burkhard Monien. Two-Way Multihead Automata Over a One-Letter Alphabet. *Informatique Théorique et Applications*, 14(1):67–82, 1980.

[MPJV97] František Mráz, Martin Plátek, Petr Jančar, and Jörg Vogel. Monotonic Rewriting Automata with a Restart Operation. In František Plášil and Keith Jeffery, editors, *SOFSEM'97: Theory and Practice of Informatics*, volume 1338 of *Lecture Notes in Computer Science*, pages 505–512. Springer Berlin / Heidelberg, 1997.

[MPV96] František Mráz, Martin Plátek, and Jörg Vogel. Restarting Automata with Rewriting. In Keith Jeffery, Jaroslav Král, and Miroslav Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *Lecture Notes in Computer Science*, pages 401–408. Springer Berlin / Heidelberg, 1996.

[Mrá01] František Mráz. Lookahead Hierarchies of Restarting Automata. *Journal of Automata, Languages and Combinatorics*, 6(4):493–506, 2001.

[MS04] Hartmut Messerschmidt and Heiko Stamer. Restart-Automaten mit mehreren Restart-Zuständen. In Henning Bordihn, editor, *14. Theorietag Automaten und Formale Sprachen 2004*, pages 111–116, Universität Potsdam, Institut für Informatik, 2004.

[Nag08] Benedek Nagy. On $5' \rightarrow 3'$ Sensing Watson-Crick Finite Automata. In Max Garzon and Hao Yan, editors, *DNA Computing*, volume 4848 of *Lecture Notes in Computer Science*, pages 256–262. Springer Berlin / Heidelberg, 2008.

[Nii86] H. Penny Nii. Blackboard Systems, Part One: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures. *AI Magazine*, 7(2):38–53, 1986.

[NO99a] Gundula Niemann and Friedrich Otto. Restarting automata, Church-Rosser languages, and confluent internal contextual languages. In *Developments in Language Theory*, pages 49–62. World Scientific, 1999.

[NO99b] Gundula Niemann and Friedrich Otto. Restarting automata, Church-Rosser languages, and representations of r.e. languages. In *Developments in Language Theory*, pages 103–114, 1999.

[NO01] Gundula Niemann and Friedrich Otto. On the power of RRWW-automata. In Masami Ito, Gheorghe Păun, and Sheng Yu, editors, *Words, Semigroups, and Transductions*, pages 341–355. World Scientific, 2001.

[NO03] Gundula Niemann and Friedrich Otto. Further results on restarting automata. In Masami Ito and Teruo Imaoka, editors, *Words, Languages and Combinatorics*, pages 352–369, Singapore, 2003. World Scientific.

[NO10] Benedek Nagy and Friedrich Otto. CD-Systems of Stateless Deterministic R(1)-Automata Accept All Rational Trace Languages. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, volume 6031 of *Lecture Notes in Computer Science*, pages 463–474. Springer Berlin / Heidelberg, 2010.

[NO11a] Benedek Nagy and Friedrich Otto. An Automata-Theoretical Characterization of Context-Free Trace Languages. In Ivana Cerná, Tibor Gyimóthy, Juraj Hromkovič, Keith Jefferey, Rastislav Królovic, Marko Vukolic, and Stefan Wolf, editors, *SOFSEM 2011: Theory and Practice of Computer Science*, volume 6543 of *Lecture Notes in Computer Science*, pages 406–417. Springer Berlin / Heidelberg, 2011.

[NO11b] Benedek Nagy and Friedrich Otto. CD-systems of stateless deterministic R(1)-automata governed by an external pushdown store. *RAIRO Theoretical Informormatics and Applications*, 45:413–448, 2011.

[NO11c] Benedek Nagy and Friedrich Otto. Finite-state Acceptors with Translucent Letters. In Gemma Bel-Enguix, Veronica Dahl, and Alfonso O. De La Puente, editors, *AI Methods for Interdisciplinary Research in Language and Biology*, pages 3–13, Portugal, 2011. SciTePress.

[Ott03] Friedrich Otto. Restarting Automata and Their Relations to the Chomsky Hierarchy. In Zoltán Ésik and Zoltán Fülöp, editors, *Developments in Language Theory*, volume 2710 of *Lecture Notes in Computer Science*, pages 55–74. Springer Berlin / Heidelberg, 2003.

[Ott06] Friedrich Otto. Restarting Automata. In Zoltán Ésik, Carlos Martín-Vide, and Victor Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 269–303. Springer Berlin / Heidelberg, 2006.

[Ott08a] Friedrich Otto. Lower Bounds for Nonforgetting Restarting Automata and CD-Systems of Restarting Automata. Technical report, Kasseler Informatikschriften, 2008.

[Ott08b] Friedrich Otto. On Stateless Restarting Automata. In Markus Holzer, Martin Kutrib, and Andreas Malcher, editors, *18. Theorietag Automaten und Formale Sprachen 2008*, pages 99–101, Universität Giessen, Institut für Informatik, 2008.

[Ott10] Friedrich Otto. CD-Systems of Restarting Automata Governed by Explicit Enable and Disable Conditions. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science*, volume 5901 of *Lecture Notes in Computer Science*, pages 627–638. Springer Berlin / Heidelberg, 2010.

[Ott12] Friedrich Otto. Centralized PC Systems of Pushdown Automata versus Multi-head Pushdown Automata. In Martin Kutrib, Nelma Moreira, and Rogério Reis, editors, *Descriptional Complexity of Formal Systems*, volume 7386 of *Lecture Notes in Computer Science*, pages 244–251. Springer Berlin / Heidelberg, 2012.

[Ott13] Friedrich Otto. Asynchronous PC Systems of Pushdown Automata. In Adrian-Horia Dediu, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications*, volume 7810 of *Lecture Notes in Computer Science*, pages 456–467. Springer Berlin / Heidelberg, 2013.

[Pap94] Christos H. Papadimitriou. *Computational complexity.* Addison-Wesley Publishing Company, Inc., 1994.

[Pet03] Elena Petre. Watson-Crick ω-Automata. *Journal of Automata, Languages and Combinatorics*, 8(1):59–70, 2003.

[Pet12] Holger Petersen. The Power of Centralized PC Systems of Pushdown Automata. *Computing Research Repository*, arXiv:1208.1283v2, 2012.

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[Plá01] Martin Plátek. Two-Way Restarting Automata and J-Monotonicity. In Leszek Pacholski and Peter Ružička, editors, *SOFSEM 2001: Theory and Practice of Informatics*, volume 2234 of *Lecture Notes in Computer Science*, pages 316–325. Springer Berlin / Heidelberg, 2001.

[PLO03] Martin Plátek, Markéta Lopatková, and Karel Oliva. Restarting Automata: Motivations and Applications. In Markus Holzer, editor, *Workshop Petrinetze and 13. Theorietag Formale Sprachen und Automaten*, pages 90–96. Technische Universität München, 2003.

[PRS98] Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *DNA Computing: New Computing Paradigms.* Texts in Theoretical Computer Science. Springer, 1998.

[PS89] Gheorghe Păun and Lila Santean. Parallel Communicating Grammar Systems: The Regular Case. *Analele Universitatii din Bucuresti, Seria matematica-informatica*, 2:55–63, 1989.

[Rei07] Jens Reimann. *Beschreibungskomplexität von Restart-Automaten.* PhD thesis, Justus-Liebig-Universität Gießen, 2007.

[Ros66] Arnold L. Rosenberg. On Multi-Head Finite Automata. *IBM Journal of Research and Development*, 10:388–394, September 1966.

[Ros67] Arnold L. Rosenberg. Real-Time Definable Languages. *Journal of the ACM*, 14(4):645–662, 1967.

[RS59] Michael O. Rabin and Dana S. Scott. Finite Automata and Their Decision Problems. *IBM Journal of Research and Development*, 3:114–125, April 1959.

[RS64] Michael O. Rabin and Dana S. Scott. *Sequential Machines: Selected Papers*, chapter Finite Automata and Their Decision Problems, pages 63–91. Addison-Wesley Longman Ltd, Essex, UK, 1964.

[RS97] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, vol. 1: Word, Language, Grammar.* Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[Sal73] Arto Salomaa. *Formal Languages.* Academic Press, Inc., New York, 1973.

[SCB05] L. Ridgway Scott, Terry Clark, and Babak Bagheri. *Scientific Parallel Computing.* Princeton University Press, 2005.

[Sch10] Natalie Schluter. On Lookahead Hierarchies for Monotone and Deterministic Restarting Automata with Auxiliary Symbols (Extended Abstract). In *Proceedings of the 14th international conference on Developments in language theory*, DLT'10, pages 440–441, Berlin, Heidelberg, 2010. Springer-Verlag.

[Sch11] Natalie Schluter. Restarting Automata with Auxiliary Symbols and Small Lookahead. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martn-Vide, editors, *Language and Automata Theory and Applications*, volume 6638 of *Lecture Notes in Computer Science*, pages 499–510. Springer Berlin / Heidelberg, 2011.

[She64] J. C. Shepherdson. *The Reduction of Two-Way Automata to One-Way Automata*, pages 92–97. Addison-Wesley, 1964.

[Sip78] Michael Sipser. Halting Space-Bounded Computations. In *19th Annual Symposium on Foundations of Computer Science*, pages 73–74, October 1978.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation.* Thomson Course Technology, USA, 2nd edition, 2006.

[SO07a] Heiko Stamer and Friedrich Otto. Restarting Tree Automata. In Jan van Leeuwen, Giuseppe Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and František Plášil, editors, *SOFSEM 2007: Theory and Practice of Computer Science*, volume 4362 of *Lecture Notes in Computer Science*, pages 510–521. Springer Berlin / Heidelberg, 2007.

[SO07b] Heiko Stamer and Friedrich Otto. Restarting Tree Automata and Linear Context-Free Tree Languages. In Symeon Bozapalidis and George Rahonis, editors, *Algebraic Informatics*, volume 4728 of *Lecture Notes in Computer Science*, pages 275–289. Springer Berlin / Heidelberg, 2007.

[Sta08] Heiko Stamer. *Restarting Tree Automata. Formal Properties and Possible Variations.* PhD thesis, Universität Kassel, 2008.

[Sud77] Ivan Hal Sudborough. Some Remarks on Multihead Automata. *Information Theory and Applications*, 11(3):181–195, 1977.

[VO12] Marcel Vollweiler and Friedrich Otto. Systems of Parallel Communicating Restarting Automata. In Rudolf Freund, Markus Holzer, Bianca Truthe, and Ulrich Ultes-Nitsche, editors, *4th Workshop on Non-Classical Models for Automata and Applications*, pages 197–212, Fribourg, Switzerland, 2012. Österreichische Computer Gesellschaft.

[Vol10] Marcel Vollweiler. Centralized Versus Non-Centralized Parallel Communicating Systems of Restarting Automata. In Friedrich

Otto, Norbert Hundeshagen, and Marcel Vollweiler, editors, *20. Theorietag Automaten und Formale Sprachen 2010*, number 2010/3 in Kasseler Informatikschriften, pages 130–135. Fachbereich Elektrotechnik / Informatik, Universität Kassel, 2010. http://nbn-resolving.de/urn:nbn:de:hebis:34-2010110534894.

[Vol12] Marcel Vollweiler. Closure Properties of Parallel Communicating Restarting Automata Systems. In František Mráz, editor, *22. Theorietag Automaten und Formale Sprachen 2012*, pages 133–138. MATFYZPRESS, publishing house of the Faculty of Mathematics and Physics, Charles University in Prague, 2012.

[War62] Stephen Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[Wot73] Detlef Wotschke. The Boolean closures of the deterministic and nondeterministic context-free languages. In Wilfried von Brauer, editor, *GI Gesellschaft für Informatik e. V. 3. Jahrestagung Hamburg, 8.-10. Oktober 1973*, volume 1 of *Lecture Notes in Computer Science*, pages 113–121. Springer Berlin / Heidelberg, 1973.

[WW86] Klaus Wagner and Gerd Wechsung. *Computational Complexity*. D. Reidel Publishing Company, 1986.

[YR78] Andrew C. Yao and Ronald L. Rivest. k + 1 Heads Are Better than k. *Journal of the ACM*, 25(2):337–340, 1978.

Index