CD-Systems of Restarting Automata

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

Hartmut Messerschmidt

Kassel im Dezember 2007

Erster Gutachter: Prof. Dr. Friedrich Otto
Zweiter Gutachter: Prof. Dr. Jürgen Dassow
Tag der Disputation: 8. Mai 2008

Contents

Chapter 1

Introduction

In this thesis restarting automata and extensions of restarting automata are studied. Restarting automata were introduced in [JMPV95] as a formal tool to describe analysis by reduction, which is a technique used in linguistics to analyze sentences of natural languages. The next section explains the basics of analysis by reduction and the process of grammar checking.

A (two-way) restarting automaton consists of a finite control, a read/write window of fixed size and a single flexible tape that contains the input as well as left and right border markers. Here flexible means that during a length reducing rewrite step the tape is really shortened. The restarting automaton has one initial state in which the computation starts. Its window is moved along the tape performing move-left and move-right steps until the window content is replaced by a shorter string in a rewrite step. It is allowed that the new string contains auxiliary symbols. Then the restarting automaton can again perform move-left and move-right steps until it decides to perform a restart step. A restart causes the restarting automaton to move to the left end of the tape and to reenter the initial state. The period from one restart step to the next is called a cycle. Thus, each computation of a restarting automaton can be described through a sequence of cycles. In general a restarting automaton is nondeterministic.

Restarting automata have two restrictions that come with their restart operation: they do not know what they have done in the last cycle, because their state is reset to the initial state, and they do not know where they have done the rewrite step of the last cycle, because they are forced to move to the left border of the tape. Nonforgetting restarting automata have only one of these restrictions. They are not forced to enter the initial state after a restart step, instead they can change the state as for any other operation. They do, however, need to move to the left end of the tape. Thus, they can remember what they have done in previous cycles, but not where they have done it.

A cooperating distributed (CD-) system of restarting automata consists of a finite collection of restarting automata, a successor relation, and a set of initial components. The cooperation is done similar to CD-grammar systems (see e.g. [CVD90, DP97]). The computation of a CD-system of restarting automata works as follows: A restarting automaton is chosen from the set of initial components and it performs some cycles on the input. Then another restarting automaton takes over and continues the computation. This automaton is chosen from among the successors of the first component. This is continued until an active component accepts or rejects. A mode of operation determines how long a component is active.

The two main parts of this thesis are about nonforgetting restarting automata and CD-systems of

restarting automata. It is assumed that the reader is familiar with the fundamentals of automata and formal language theory, advanced terms are introduced when they are needed.

1.1 Grammar Checking and Analysis by Reduction

In this section the linguistic background of restarting automata is discussed and the applications of restarting automata in linguistics are given. The complete process of language recognition is not described here, the intention of this section is rather to give an overview over the process and to describe some special problems.

Analysis by reduction and restarting automata are mainly of interest for languages with free word order, such as slavic languages, Russian and German. Analysis by reduction simplifies or shortens a sentence while preserving the correctness or incorrectness of it. This is done until a simple sentence is obtained, such that its correctness is easily decidable.

But how is analysis by reduction done, and what is a sentence?

A sentence is a sequence of symbols. In natural languages there exists a special blank symbol that separates the words in a sentence. So in natural languages a sentence is a sequence of words and punctuation marks, where words are sequences of letters. In fact a word consists of smaller parts, so called morphemes, for now it is sufficient to regard a sentence as a sequence of words.

To analyze a sentence it is necessary to retrieve information about it, to give a meaning to the words in it. This is done by a morphological, a syntactical and perhaps a semantical analysis. It is not always possible to do these analyses completely, because for example most of the words have more than one meaning. Therefore after these analyses the ambiguity of the sentence must be reduced. When the sentence is disambiguated the next step is the analysis by reduction. On the next pages this process is described and the methods for this process are given.

1.1.1 Syntactical and Morphological Analysis

The first step is to find the corresponding tags for all words or groups of words in the sentence. Every sequence of letters (word) gets a morphological tagset, which contains for example the part of speech, number, gender, person, case and time. It gets a syntactical tagset for the syntactical meaning, (e.g. Subject, Predicate, Object) and as part of the semantical analysis, it gets a structural semantic tagset to describe the meaning of words and subsentences. The last two tagsets can be assigned to words and groups of words.

To have a closer look at this analysis and at the ambiguity of words and word groups we give two examples:

The subsentence or word group "*Peter, Paul and Mary*" can be an enumeration but it need not be one. In the sentence "*I am living next door to Peter, Paul and Mary are living in another street.*", it is not an enumeration. A morphological tag for the word *talks* can be:

Part-of speech: verb

Number: singular

Person: third

Time: present

but if we have no further knowledge about the word the following tag is also valid:

Part-of speech: noun

Number: plural

Person: -

Time: -

So a word normally gets more than one tag in each tagset.

Morphological analysis is mostly done by checking the regular cases and by doing a lookup in a large database for the irregular ones. Up to now there are no effective ways to make morphological analysis by an algorithm, so the only way is a database which has all irregular tags to a given sequence of letters. To divide the regular from the irregular cases morphemes can be used.

A morpheme is the smallest part of a word that carries a semantic interpretation. For example "houses" has two morphemes "house" and "s" where "s" is a plural suffix. The exact way to do morphological, syntactical or semantical analysis is not part of this work. It is important to note that after the analysis every word has a tagset of all its possible tags. We have seen that more than one tag is possible. If a word has different tags it is called ambiguous. But normally not all these tags are valid with respect to the full sentence, therefore disambiguation is used to reduce the ambiguity. After this, if no discrepancy was detected, the analysis by reduction can start to analyze the sentence.

1.1.2 Morphological Disambiguation

There are different types of analyses, which lead to different tagsets for a word or a word group. All of these tagsets need to be disambiguated. To understand the method of disambiguation it is sufficient to have a look at the morphological disambiguation.

An ambiguous sentence can be described in two different ways, in the columns notation or in the readings notation. In the columns notation there is a column assigned to every word. This column contains all tags from the tagset for this word. All possible meanings are obtained by taking one tag from each tagset.

In the readings notation, all possible meanings of the sentence are given. This means that for each word one tag is chosen from its tagset and all possible combinations of tags are written in different rows. Every such combination is a different reading.

For example assume that a and b are words with the tagset $\{A, B\}$ for a and with the tagset $\{B\}$ for b, and that "$a\ a\ b$ " is a sentence. Then the columns notation for this sentence is $\{A, B\}, \{A, B\}, \{B\}$, while the readings notation is the following set of readings: $\{(AAB), (ABB), (BAB), (BBB)\}$. This formal "sentence" is used to describe the concepts and the behavior without expecting too much linguistic knowledge.

The columns notation is more compact and gives a better overview about the ambiguity, but it has also some disadvantages, as we will see during the explanation of disambiguation.

Disambiguation is the process in which the context of a word is used to eliminate some of its tags. In languages with free word order it is not always a local context only, sometimes the whole sentence must be used as context. This is done with several rules, for example a rule may state that no verb can follow an article or that a following noun is not allowed to have a gender different from that of the article.

Disambiguation rules are negative rules, as they state what is not possible. With these negative rules it is possible to remove tags from a word if we are in the column notation, or to reject readings in the readings notation.

In the ideal case all but one tag are removed from every word, thus every word has only one morphological meaning. This is equivalent to the fact that only one reading remains of the sentence. If it is not possible to fully disambiguate a sentence, then there can be a difference between the columns and the readings notation.

In the readings notation all readings which are incorrect are removed, this gives the maximal disambiguation of the sentence by the disambiguation rules. In the columns notation a tag can only be removed if it is incorrect in all possible readings. Thus a sentence can be less ambiguous as the columns notation suggests.

For example, assume the sentence $a\ a\ b$ remains ambiguous, say the readings (AAB) and (BBB) are correct. Then in the columns notation a has the tagset $\{A, B\}$ and b has the tagset $\{B\}$. But this columns notation corresponds to the four readings (AAB), (ABB), (BAB) and (BBB). Thus, in the readings notation a better disambiguation can be achieved.

Note that a partial disambiguation can have different reasons: either the sentence contains a "genuine ambiguity", which means that more than one possible reading of the sentence is correct, or the disambiguation rules are not complete.

On the other hand, if no reading of a sentence is left, then the sentence is not correct. This does not mean that the word where the last tag was deleted is incorrect. It only says that the whole sentence is incorrect. To find the error in the sentence and to make suggestions for a correction other techniques must be used.

Even if these rules are applied in non-deterministic order they are error and correctness preserving. This means that correct sentences are never found to be incorrect and that incorrect sentences are never changed in a way that they become correct.

Note: To do better disambiguation it is possible to apply rules that are true for almost all sentences, but not for all. This is called a stochastic disambiguation. As natural languages are not defined formally, there are sometimes cases which cannot be described by rules, or there are rules which are correct only for most of the cases.

1.1.3 Analysis by Reduction

The disambiguated sentence is analyzed by the analysis by reduction. Analysis by reduction consists of a stepwise simplification of a sentence, so that the correctness and incorrectness of the sentence is preserved. These properties are called the Correctness Preserving and the Error Preserving Property.

Analysis by reduction is a well known method in language recognition in the Czech republic. There are two ways of doing an analysis by reduction. A first very formal way starts with a fully disambiguated sentence. But Czech children learn analysis by reduction already at the primary school, so there is a second more informal way, which is done with plain text. We will present an example which uses this informal analysis by reduction. For many sentences even children at the primary school know how to reduce them, without understanding their morphological, syntactical and semantical structure and meaning.

Analysis by reduction is a stepwise reduction of a given sentence, which continues until either an error is detected or a correct simplified sentence is obtained. In the following example, a sentence is given and it is assumed that the sentence is fully disambiguated, so each word has only one tag in each of its tagsets. The sentence is so simple that these tags do not need to be written to understand the example. Therefore the informal way of analysis by reduction is used.

Example 1.1.1.
The sentence:

Peter, Paul and Mary eat delicious ice cream.

can be reduced to

Peter, Paul and Mary eat ice cream.

or one of the enumeration parts can be deleted

Peter and Mary eat delicious ice cream.

After each of these reductions the other one can be applied, therefore these reductions are independent from each other and both reductions lead to the sentence:

Peter and Mary eat ice cream.

This sentence is in such a simple form that its correctness can be seen easily.

There are two more reductions from the original sentence:

Paul and Mary eat delicious ice cream.
Peter and Paul eat delicious ice cream.

The second one contains a rewrite instead of only a deletion and these reductions are not independent from each other, because only one of them can be applied to the sentence. If one more name is deleted, then the subject is no longer an enumeration and the number of the subject changes from plural to singular.

Apart from independent reductions there can be dependencies between some reductions. The reduction from "Peter, Paul and Mary" to "Peter and Paul" must be done in one step, because neither "Peter, Paul and eat delicious ice cream" nor "Peter and Paul Mary eat delicious ice cream" are correct sentences.

In the formal analysis by reduction the sentence must be fully disambiguated. If it is not fully disambiguated, then the analysis by reduction must be done for all remaining readings of the sentence. For example the subsentence "Peter, Paul and Mary" can only be reduced to "Peter and Mary" if it is an enumeration.

To do so one needs to know exactly which tag belongs to which word or word group, in order to rewrite the sentence. If the subsentence "*Peter, Paul and Mary*" is an enumeration, then it can be reduced to "*Peter and Mary*". But in Section 1.1.1 the following sentence was given: "*I am living next door to Peter, Paul and Mary are living in another street.*" In this sentence "*Peter, Paul and Mary*" is not an enumeration and the reduction to "Peter and Mary" would contradict the Correctness Preserving Property.

Another important property of analysis by reduction is that it also preserves the meaning of the whole sentence: "I believe in Santa Claus" must not be rewritten into "I believe Santa Claus". Here both sentences are correct, but they express different things.

Restarting automata were introduced to model analysis by reduction, but they can be used for both the disambiguation and the analysis by reduction. During the disambiguation tags are deleted from tagsets, while in the analysis by reduction words of the sentence itself are deleted or rewritten. Therefore one could think about merging these two steps by using different restarting automata for the disambiguation and for the analysis by reduction. These restarting automata can work together as a cooperating distributed (CD-) system of restarting automata to analyze the sentence. We will see later in this work that a CD- system of restarting automata corresponds in some sense to nonforgetting restarting automata.

1.2 Summary and Results

We will now give a summary of the main part of this thesis. The main results are stated and the definitions and automata models are informally described.

Restarting Automata

In Chapter 2 restarting automata are studied. Definitions and known results are stated, for an overview see e.g. [Ott03, Ott06]. An unrestricted restarting automaton is called RLWW-automaton. There are various restricted types of restarting automata which are obtained by combining two types of restrictions:

(a) Restrictions on the movement of the read/write window (expressed by the first part of the class name):

- RL- denotes no restriction,
- RR- means that no move left steps are allowed, thus the automaton scans the tape only once from left to right between two restart steps,
- R- means that no move left steps are allowed and each rewrite step is immediately followed by a restart.

(b) Restrictions on the rewrite-instructions (expressed by the second part of the class name):

- -WW denotes no restriction,
- -W means that no auxiliary symbols are available,
- ε means that no auxiliary symbols are available and that each rewrite step is simply a deletion (that is, if the rewrite operation $u \to v$ occurs in the transition relation of M, then v is obtained from u by deleting some symbols).

In this work ε represents the empty string.

The equivalence of nondeterministic RLWW- and RRWW-automata is stated in [Plá01]. This equivalence allows to introduce a shorter form of description for restarting automata, so called meta-instructions. The Error Preserving Property (EPP) and the Correctness Preserving Property (CPP) are stated and a reduction from RRWW- to RRW- and then to RR-automata is given [JOMP04, Ott06]. Also deterministic, monotone, deterministic monotone and weakly monotone restarting automata are introduced. Restarting automata are called monotone, if the place where the rewrite of a cycle occur is restricted. In two following cycles the right distance of the rewrite is not increasing. It is called weakly monotone, if there exists a constant, such that the right distance is increased by at most this constant.

For monotone restarting automata it is known that CFL = $\mathcal{L}$(mon-RWW) = $\mathcal{L}$(mon-RRWW) [JMPV99], DCFL = $\mathcal{L}$(det-mon-R) = $\mathcal{L}$(det-mon-RRWW) [JMPV99] and $\mathcal{L}$(det-mon-RLWW) = $\mathcal{L}$(det-mon-RL) [JMOP05] hold. Here CFL stands for the context-free languages and DCFL for the deterministic context-free languages. $\mathcal{L}$(RRWW) describes the class of languages that are accepted by RRWW-automata.

Deterministic restarting automata with auxiliary symbols recognize exactly the Church-Rosser languages [NO00, NO03], while weakly monotone restarting automata with auxiliary symbols characterize the growing context-sensitive languages [Nie02, JLNO04]. For deterministic, monotone and weakly monotone restarting automata without auxiliary symbols there exists a strict hierarchy for the different types of restrictions for restarting automata. For unrestricted restarting automata it is an open question whether the inclusion $\mathcal{L}$(RWW) $\subseteq$ $\mathcal{L}$(RRWW) is proper or not.

In the following, generalizations of restarting automata are considered: the shrinking restarting automata, the nonforgetting restarting automata, and the restarting automata with many rewrites per cycle. It has been shown that shrinking RRWW-automata are as expressive as shrinking nonforgetting restarting automata with many rewrites per cycle [JO07]. This language class coincides with the class of languages accepted by finite change automata (see [vBV79]).

In the sections 2.1 - 2.7 almost all results were already known. Only Lemma 2.2.6 and Theorem 2.2.7 contain new results. A restarting automaton is correctness preserving, if there exists no cycle such that an incorrect word is derived from a correct one. Each deterministic restarting automaton is per definition correctness preserving. Correctness preserving restarting automata are defined, and it is shown that correctness preserving X-automata are as expressive as deterministic X-automata, for $\mathsf{X} \in \{\mathsf{R}, \mathsf{RL}, \mathsf{RW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RLWW}\}$. For RR- and RRW-automata a separation language is given to show the strictness of the inclusion between deterministic and correctness preserving restarting automata. The error detection distance is a generalization of the correctness preservation. A restarting automaton can make an error, but if it always detects this error after c cycles it has error detection distance c.

It is shown that the property of having a finite error detection distance does not add expressive power to restarting automata. It is shown for all types of restarting automata that correctness preserving restarting automata are as expressive as restarting automata with finite error detection distance. The results in the Sections 2.8 and 2.9 are joint work with Friedrich Otto, and an extended abstract appeared in [MO08].

Nonforgetting Restarting Automata

Chapter 3 is one of the main chapters of this thesis. There a generalization of restarting automata, the nonforgetting (nf-) restarting automata are studied. They were first mentioned in [MS04]. During this chapter, restarting automata are often called forgetting restarting automata to distinguish them from nonforgetting restarting automata.

The same restricted types as for forgetting restarting automata are applied and it is also shown that each nf-RLWW-automaton can be simulated by a nf-RRWW-automaton. Similarly, meta-instructions are defined and error and correctness preserving properties are given. These properties vary from those for forgetting restarting automata; in fact they are real extensions of the normal EPP and CPP.

For deterministic nonforgetting restarting automata it is shown that already the weakest model, that is the det-nf-R-automaton, accepts languages that are not growing context-sensitive. Thus all types of deterministic nonforgetting restarting automata are strictly more expressive than their forgetting counterparts.

A normalform for deterministic (nonforgetting) RLWW-automata is derived that allows to describe (nonforgetting) deterministic RLWW-automata by meta-instructions.

Unfortunately, there is not a complete hierarchy for deterministic nonforgetting restarting automata. It is an open question whether the inclusions $\mathcal{L}(\text{det-nf-R(W)(W)}) \subseteq \mathcal{L}(\text{det-nf-RR(W)(W)}) \subseteq \mathcal{L}(\text{det-nf-RL(W)(W)})$ are proper. In addition we will see that det-nf-RW-automata accept the language of all valid computations of a Turing machine, and therefore, many decision problems for these automata are undecidable. As an encoding of this language is accepted by a det-nf-R-automaton these undecidable results are valid for all kinds of deterministic nonforgetting restarting automata.

Monotone nonforgetting RRWW-automata are as expressive as monotone forgetting RRWW-automata as they accept exactly the context-free languages. For deterministic monotone restarting automata the known results do not carry over from forgetting to nonforgetting restarting automata. While $\mathcal{L}(\text{det-mon-nf-R}) = \mathcal{L}(\text{det-mon-nf-RWW}) = \mathsf{DCFL}$ holds, we show that already

det-mon-nf-RR-automata accept languages that are not deterministic context-free. In fact we obtain a proper hierarchy between DCFL and $\mathcal{L}$(det-mon-RL) (see Figure 1.1), which equals the LR-regular languages [Ott07] .

$$\begin{array}{ccccc} \text{LRR} & = & \mathcal{L}(\text{det-mon-nf-sh-RLWW}) & = & \mathcal{L}(\text{det-mon-RL}) \\ & & \uparrow & & \\ & & \mathcal{L}(\text{det-mon-nf-RRW}) & & \\ & & \uparrow & & \\ & & \mathcal{L}(\text{det-mon-nf-RR}) & & \\ & & \uparrow & & \\ \text{DCFL} & = & \mathcal{L}(\text{det-mon-nf-R}) & = & \mathcal{L}(\text{det-mon-nf-RWW}) \end{array}$$

Figure 1.1: The taxonomy of deterministic monotone nonforgetting restarting automata

This is the first time that it is shown that RWW-automata are strictly less powerful than the RRWW-automata of the same type. In addition, both automata models describe well known language classes. The result $\mathcal{L}$(det-mon-RL) = $\mathcal{L}$(det-mon-RLWW) is extended to shrinking and nonforgetting restarting automata, that is, the following holds:

$$\mathcal{L}(\text{det-mon-RL}) = \mathcal{L}(\text{det-mon-nf-sh-RLWW}) = \mathcal{L}(\text{det-mon-nf-RRWW}) = \text{LRR}.$$

In the section about shrinking nonforgetting restarting automata it is shown that the class of languages accepted by shrinking det-RLWW-automata does not increase if the restarting automaton is nonforgetting or can perform many rewrites per cycle. This is a result similar to the one about nondeterministic shrinking RRWW-automata. Another possible way to define the notion of shrinking for nonforgetting restarting automata is introduced and some properties for this language class are shown.

Then the notion of correctness preservation and the error detection distance is carried over to nonforgetting restarting automata. Here the new correctness preserving property is used. Many of the results obtained for forgetting restarting automata are as well shown for nonforgetting restarting automata. But there are two open questions: Does a finite error detection distance increase the expressive power of nf-R(W)(W)-automata, and are correctness preserving nf-RR(W)(W)-automata more expressive than det-nf-RR(W)(W)? The second question is shown to be equivalent to the open question whether $\mathcal{L}$(det-nf-R(W)(W)) equals $\mathcal{L}$(det-nf-RR(W)(W)) or not. In the last section of this chapter cyclic restarting automata are introduced and compared to forgetting restarting automata with many rewrites per cycle.

CD-Systems of Restarting Automata

In Chapter 4, the second main part of this work, cooperating distributed (CD-) systems of restarting automata are studied. As explained above, they are collections of restarting automata, and the cooperating is controlled by a mode of operation, which determines how long a component is active. The following modes of operation are considered:

$= j$	:	execute exactly j cycles;
$\leq j$	:	execute up to j cycles;
$\geq j$	:	execute at least j cycles;
t	:	continue until no more cycle can be executed.

After a formal definition of CD-systems of restarting automata, meta-instructions, the Error preserving and the Correctness preserving property are defined. These properties are closely related to the ones for nonforgetting restarting automata. The original properties are no useful tools for CD-systems either. Then it is shown that not only the EPP and CPP of nonforgetting restarting automata and CD-systems are closely related: A nonforgetting restarting automaton can be simulated by a CD-system of restarting automata working in mode = 1. The other direction is true for all modes of operations and almost all types of restricted restarting automata. The only open question is whether CD-systems of R-, RW-, and RWW-automata working in mode t can be simulated by nonforgetting restarting automata of the same type or not.

Deterministic CD-systems of restarting automata are considered as well, in fact three different notions of determinism are introduced.

A CD-system of restarting automata can be called deterministic, if each component is a deterministic restarting automaton. This is the way in which deterministic CD-grammar systems are defined. But a computation of such a deterministic CD-system is not necessarily completely deterministic. The initial component can be chosen nondeterministically from among the initial components and the same is true for a successor component. Therefore these CD-systems are called locally deterministic.

If we restrict a deterministic CD-system to only one initial component and require that each component has exactly one successor, then each computation of such a CD-system is completely deterministic. But the restriction to only one successor is a rather serious one, so these systems are called strictly deterministic.

A third notion of determinism is introduced, that guarantees a deterministic computation without the restriction to only one successor component. A CD-system of restarting automata is called globally deterministic, if each component is deterministic, if there is only one initial component and if each restart is assigned with a successor from among the possible successors. Thus a computation of a globally deterministic CD-system of restarting automata is completely deterministic, but a component can have more than one successor.

Only globally and strictly deterministic CD-systems of restarting automata working in a deterministic mode of operation are completely deterministic and therefore only those are per definition correctness preserving.

Globally deterministic CD-systems of restarting automata working in mode = 1 are shown to be related to deterministic nonforgetting restarting automata as in the nondeterministic case.

Further it is shown that in mode = 1 all types of strictly deterministic CD-systems are strictly more powerful than deterministic restarting automata of the same type, and that strictly deterministic CD-systems without auxiliary symbols are strictly less powerful than globally deterministic CD-systems.

Locally deterministic CD-systems seem less restricted than globally deterministic ones, but they cannot control the choice of a successor component. However it is shown that at least in mode = 1 they can simulate globally deterministic CD-systems. For CD-systems of restarting automata without auxiliary symbols working in mode = 1, it is shown that the expressive power of locally deterministic CD-systems of restarting automata lies in between that of globally deterministic and nondeterministic CD-systems. Thus the following hierarchy for CD-X-systems is achieved. Here X denotes a R(R)(W)-automaton:

$$\mathcal{L}(\text{det-X}) \subsetneq \mathcal{L}_{=1}(\text{det-strict-CD-X}) \subset \mathcal{L}_{=1}(\text{det-global-CD-X}) \subset \mathcal{L}_{=1}(\text{det-local-CD-X}) \subset \mathcal{L}_{=1}(\text{CD-X}).$$

Finally, CD-systems of shrinking restarting automata are considered and it is shown that the relation to nonforgetting restarting automata can be extended to shrinking restarting automata.

Chapter 2

Restarting Automata

The restarting automaton is the automata model for analysis by reduction as described in the previous chapter. It simulates one reduction step by one cycle of the automaton. It is also a useful tool in formal language theory, as we will see in this chapter. Restarting automata were first specified in [JMPV95], and it has been shown that many common language classes can be characterized by different types of restarting automata. Deterministic restarting automata characterize the Church-Rosser languages, monotone restarting automata accept exactly the context-free languages, deterministic monotone restarting automata characterize the deterministic context-free languages, while weakly monotone restarting automata can be used to describe the growing context-sensitive languages. All these restrictions will be described in this chapter. There are also some extensions of restarting automata, which are studied in this and in the following chapters, but all types of restarting automata only characterize certain context-sensitive and NP languages.

A restarting automaton consists of a finite control, one flexible tape containing the finite input as well as left and right border markers, and a window of finite size. The tape is flexible as it is shortened in every rewrite step. A restarting automaton moves its window over the tape, by using move-right and move-left steps. It can also perform rewrite and restart steps. In a rewrite step the window content u is rewritten into a word v, which is strictly shorter than u and the window is placed to the right of v, in a restart step the window is placed over the left end of the tape and the automaton reenters the initial state. In addition the automaton can halt and accept, and if it is not able to do another step it halts and rejects. It is required that rewrite and restart steps alternate, when ignoring move operations, with a rewrite step coming first. The period from one restart-step to the next is called a cycle.

Figure 2.1: Schematic representation of a restarting automaton

2.1 Definitions and Examples

Most of the formal definitions, results and examples in this chapter are taken from [Ott03] or [Ott06]. Some are slightly varied, generalized or rearranged.

Definition 2.1.1. *A* two-way restarting automaton*,* RLWW*-automaton for short, is a one-tape machine that is described by an 8-tuple* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$*, where* Q *is the finite set of states,* Σ *is the finite input alphabet,* Γ *is the finite tape alphabet containing* Σ*, the symbols* $¢, \$ \notin \Gamma$ *are markers for the left and right border of the work space, respectively,* $q_0 \in Q$ *is the initial state,* $k \geq 1$ *is the size of the* read/write window*, and*

$$\delta : Q \times \mathcal{PC}^{(k)} \to \mathfrak{P}((Q \times (\{\mathsf{MVR}, \mathsf{MVL}\} \cup \mathcal{PC}^{\leq (k-1)})) \cup \{\mathsf{Restart}, \mathsf{Accept}\})$$

is the transition relation.

Here $\mathfrak{P}(S)$ denotes the powerset of the set S, $\mathcal{PC}^{(k)}$ is the set of *possible contents* of the read/write window of M, where

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

and

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

The transition relation describes five different types of transition steps:

(1.) A *move-right step* is of the form $(q', \mathsf{MVR}) \in \delta(q, u)$, where $q, q' \in Q$ and $u \in \mathcal{PC}^{(k)}$, $u \neq \$$. If M is in state q and sees the string u in its read/write window, then this move-right step causes M to shift the read/write window one position to the right and to enter state q'. However, if the contents u of the read/write window is only the symbol \$, then no shift to the right is possible.

(2.) A *move-left step* is of the form $(q', \mathsf{MVL}) \in \delta(q, u)$, where $q, q' \in Q$ and $u \in \mathcal{PC}^{(k)}$, $u \notin ¢ \cdot \Gamma^*$. It causes M to shift the read/write window one position to the left and to enter state q'. This, however, is only possible if the window is not already at the left end of the tape.

(3.) A *rewrite step* is of the form $(q', v) \in \delta(q, u)$, where $q, q' \in Q$, $u \in \mathcal{PC}^{(k)}$, $u \neq \$$, and $v \in \mathcal{PC}^{\leq (k-1)}$ such that $|v| < |u|$. It causes M to replace the contents u of the read/write window by the string v, and to enter state q'. Further, the read/write window is placed immediately to the right of v. However, some additional restrictions apply in that the border markers ¢ and \$ must not disappear from the tape nor that new occurrences of these markers are created. Further, the read/write window must not move across the right border marker \$, that is, if the string u ends in \$, then so does the string v, and after performing the rewrite operation, the read/write window is placed on the \$-symbol.

(4.) A *restart step* is of the form $\mathsf{Restart} \in \delta(q, u)$, where $q \in Q$ and $u \in \mathcal{PC}^{(k)}$. It causes M to move its read/write window to the left end of the tape, so that the first symbol it sees is the left border marker ¢, and to re-enter the initial state q_0.

(5.) An *accept step* is of the form $\mathsf{Accept} \in \delta(q, u)$, where $q \in Q$ and $u \in \mathcal{PC}^{(k)}$. It causes M to halt and accept.

If no instruction with a left-hand side (q, u) exists, then the automaton halts and rejects, when it reaches state q while it sees tape content u in its window. A *halting configuration* is either an accepting or a rejecting configuration. A restarting automaton performs exactly one rewrite operation between two restart steps. In general an automaton M is non-deterministic that means that for a left-hand side (q, u) there can be more than one instruction. If for every left-hand side (q, u) there exists at most one instruction, then M is called deterministic and the prefix det- is used to describe the class of deterministic restarting automata.

A configuration of a restarting automaton is given by a word $\alpha q \beta$ where $q \in Q$ is the current state and $\alpha\beta$ is the current tape content including the border markers, with the read/write window scanning the first k letters of the word β. A configuration $q_0 ¢ \omega \$$ with $\omega \in \Gamma^*$ is called a *restarting configuration*, and if $\omega \in \Sigma^*$ then it is called an *initial configuration*. The transition relation transforms a configuration $\alpha_0 q_0 \beta_0$ into a successor configuration $\alpha_1 q_1 \beta_1$. This is denoted as $\alpha_0 q_0 \beta_0 \vdash \alpha_1 q_1 \beta_1$ and the type of the transition relation is given as an index. $\vdash^*_{\mathsf{MVR}}$ describes a sequence of MVR steps.

Definition 2.1.2. *The language accepted by a restarting automaton M, $L(M)$, consists of all words $w \in \Sigma^*$ for which there is an accepting computation of M starting from the initial configuration $q_0 ¢ w \$$. The complete language, or characteristic language, of M, $L_C(M)$ for short, consists of all words $w \in \Gamma^*$ which are accepted by M.*

Example 2.1.3. Let $M := (Q, \Sigma, \Sigma, ¢, \$, q_0, 3, \delta)$ be the deterministic RLWW-automaton that is defined by taking $Q := \{q_0, q_c, q_d, q_r\}$, $\Sigma := \{a, b, c, d\}$, and δ as given by the following table:

(1)	$\delta(q_0, x)$	$=$	(q_0, MVR)	for all $x \in \{aaa, aab, abb, abc, bbb, bbc, bbd\}$,			
(2)	$\delta(q_0, ¢c\$)$	$=$	Accept,	(9)	$\delta(q_c, abc)$	$=$	(q_r, c),
(3)	$\delta(q_0, ¢d\$)$	$=$	Accept,	(10)	$\delta(q_c, bbc)$	$=$	(q_c, MVL),
(4)	$\delta(q_0, ¢ab)$	$=$	(q_0, MVR),	(11)	$\delta(q_c, bbb)$	$=$	(q_c, MVL),
(5)	$\delta(q_0, ¢aa)$	$=$	(q_0, MVR),	(12)	$\delta(q_c, abb)$	$=$	(q_r, b),
(6)	$\delta(q_0, bc\$)$	$=$	(q_c, MVL),	(13)	$\delta(q_d, bbd)$	$=$	(q_d, MVL),
(7)	$\delta(q_0, bd\$)$	$=$	(q_d, MVL),	(14)	$\delta(q_d, bbb)$	$=$	(q_d, MVL),
(8)	$\delta(q_r, -)$	$=$	Restart,	(15)	$\delta(q_d, abb)$	$=$	(q_r, ε).

Obviously, M accepts the strings c and d immediately. So let $w \in \Sigma^+ \setminus \{c, d\}$. Starting from the initial configuration $q_0 ¢ w \$$, M will get stuck (and therewith reject) while scanning w from left to right unless w is of the form $w = a^m b^n c$ or $w = a^m b^n d$ for some positive integers m and n. In these cases the configuration $¢ a^m b^{n-1} q_0 bc\$$ or $¢ a^m b^{n-1} q_0 bd\$$ is reached by performing the transition steps (4) or (5) and (1) repeatedly, and then either the state q_c (6) or the state q_d is entered (7). Now M performs MVL-steps until the read/write window gets back to the boundary between the syllables a^m and b^n. If the actual state is q_c that is, if w ends in c, then a factor ab is deleted (12); if the actual state is q_d that is, if w ends in d, then a factor abb is deleted (15). In both cases M enters the state q_r and restarts (8) independent of the string in the read/write window. Thus, we see that M accepts the following language

$$L_1 := \{\, a^n b^n c \mid n \geq 0 \,\} \cup \{\, a^n b^{2n} d \mid n \geq 0 \,\},$$

which is a well-known example of a context-free language that is not deterministic context-free.

There are various restricted types of restarting automata. They are obtained by combining two types of restrictions:

(a) Restrictions on the movement of the read/write window (expressed by the first part of the class name):

- RL- denotes no restriction,
- RR- means that no MVL-steps are allowed, thus the automaton scans the tape only once from left to right between two restart steps,
- R- means that no MVL-steps are allowed and each rewrite step is immediately followed by a restart.

(b) Restrictions on the rewrite-instructions (expressed by the second part of the class name):

- -WW denotes no restriction,
- -W means that no auxiliary symbols are available (that is, $\Gamma = \Sigma$),
- ε means that no auxiliary symbols are available and that each rewrite step is simply a deletion (that is, if the rewrite operation $u \to v$ occurs in the transition relation of M, then v is obtained from u by deleting some symbols).

A restarting automaton works in phases. A phase, called *cycle*, starts in a restarting configuration followed by some MVR-, MVL- and a single Rewrite-step and then by a Restart-step which leads to a new restarting configuration. If no further Restart-step is performed, then every finite computation ends with a halting configuration. Such a phase is called a *tail*. In every cycle exactly one rewrite step must be performed, while in a tail at most one rewrite step can be performed.

Let $x, y \in \Gamma^*$, then a cycle of a restarting automaton M from one restarting configuration $q_0 ¢ x \$$ to another restarting configuration $q_0 ¢ y \$$ is denoted by $(q_0, ¢x\$) \vdash_M^c (q_0, ¢y\$)$; an accepting tail starting from $q_0 ¢ x \$$ is denoted by $(q_0, ¢x\$) \vdash_M^c \mathsf{Accept}$. As the border markers and the initial state remain unchanged, this relation can be seen as a reduction relation $x \vdash_M^c y$ over Γ^*

In the following chapters nonforgetting restarting automata and CD-systems of restarting automata will be introduced, and for these automata there is a difference between the relation $\vdash_M^c$ over restarting configurations and the induced relation over words. There we will need information in addition to the words to describe cycles.

In each cycle an RRWW-automaton can scan its tape only once from left to right. Nevertheless it has been shown in [Plá01] that nondeterministic RRWW-automata accept the same languages as RLWW-automata.

Each cycle of each computation of an RLWW-automaton M consists of three phases: first M scans its tape performing MVR- and MVL-instructions, then it executes a Rewrite step, and finally it scans its tape again performing MVR- and MVL-steps. Hence, in the first and the last phase of each cycle M behaves like a nondeterministic two-way finite-state acceptor (2NFA). In analogy to the proof that the language accepted by a 2NFA is regular (see, e.g, [HU79]), the following result can be established.

Theorem 2.1.4. [Plá01]

Let $M_L = (Q_L, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_L)$ be an RLWW-automaton. Then there exists an RRWW-automaton $M_R = (Q_R, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_R)$ such that, for all $u, v \in \Gamma^*$,

$$q_0 ¢ u \$ \vdash_{M_L}^c q_0 ¢ v \$ \quad \textit{if and only if} \quad q_0 ¢ u \$ \vdash_{M_R}^c q_0 ¢ v \$,$$

and the languages $L(M_L)$ and $L(M_R)$ coincide.

This result is valid only in non-deterministic mode. In deterministic mode there is a difference in the expressive power between RRWW- and RLWW-automata.

In [Ott03] it is shown that for every (det-)RRWW-automaton M, there exists a (det-)RRWW-automaton M' which accepts the same language and which performs restart steps only when it sees the right border marker. Thus, every cycle of an RRWW-automaton can be divided into four parts:

$$q_0\¢\omega\$ = q_0\¢\omega_1 u\omega_2\$ \underset{\textsf{MVR}}{\vdash^*} \¢\omega_1 q_1 u\omega_2\$ \underset{\textsf{Rewrite}}{\vdash} \¢\omega_1 v q_2\omega_2\$ \underset{\textsf{MVR}}{\vdash^*} \¢\omega_1 v\omega_2 q_3\$ \underset{\textsf{Restart}}{\vdash} q_0\¢\omega_1 v\omega_2\$.$$

In the first part there are only move right operations, then one rewrite step is performed, followed by more move right operations, and finally a restart is executed when the right border marker is reached. During the move right parts the RRWW-automaton works like a finite automaton. Thus, these parts can be described by regular expressions. We can use this fact to shorten the description of the transition relation δ by using *meta-instructions*.

Definition 2.1.5. *A* meta-instruction *of an* RRWW*-automaton M is either of the form $(E_1, u \to v, E_2)$ or (E_1, Accept), where E_1 and E_2 are regular expressions and $u, v \in \Gamma^*$ are words with $|u| > |v|$. The rule $u \to v$ stands for a* Rewrite *step of M. To perform a cycle, M chooses a meta-instruction of the form $(E_1, u \to v, E_2)$. On trying to execute this meta-instruction, M will get stuck (and so reject) starting from the configuration $q_0\¢\omega\$$, if ω does not admit a factorization of the form $\omega = \omega_1 u\omega_2$ such that $\¢\omega_1 \in E_1$ and $\omega_2\$ \in E_2$. On the other hand, if ω does have factorizations of this form, then one such factorization is chosen nondeterministically, and $q_0\¢\omega\$$ is transformed into $q_0\¢\omega_1 v\omega_2\$$. In order to describe the tails of accepting computations we use meta-instructions of the form (E_1, Accept), where the strings of the regular language E_1 are accepted by M in tail computations.*

A rewriting meta-instruction *of an* RWW*-automaton M is of the form $(E_1, u \to v)$, because it restarts immediately after the rewrite.*

Restrictions on the second part of a restarting automaton ($-\mathsf{W}$ and ε) only effect the word v in the meta-instruction, because they restrict the form of the rewrite operation. For $-\mathsf{W}$, no auxiliary symbols may appear in v, and for ε, v must be a scattered subword of u.

Example 2.1.6. Let M_1 be the RRWW-automaton with input alphabet $\{a, b, c, d\}$ and without auxiliary symbols that is described by the following sequence of meta-instructions:

$$\begin{array}{llllll} (1) & (\¢ \cdot a^*, & ab \to \varepsilon, & b^* \cdot c\$), & (3) & (\¢ c\$, \quad \mathsf{Accept}), \\ (2) & (\¢ \cdot a^*, & abb \to \varepsilon, & b^* \cdot d\$), & (4) & (\¢ d\$, \quad \mathsf{Accept}). \end{array}$$

It is easily seen that $L(M_1) = L_1$ holds.

The restarting automaton from Example 2.1.3 is a det-RL-automaton, because it only deletes in its rewrite steps, and it is deterministic. The language L_1 can be recognized by an RR-automaton, because the restarting automaton in Example 2.1.6 performs only MVR-steps, but is not deterministic. The following example shows that L_1 can be recognized by an RWW-automaton.

Example 2.1.7. The following sequence of meta-instructions defines an RWW-automaton M_2 for the language L_1. M_2 has input alphabet $\Sigma := \{a, b, c, d\}$ and tape alphabet $\Gamma := \Sigma \cup \{C, D\}$:

$$\begin{array}{llllll} (1) & (\¢ \cdot a^*, & ab \to C), & (5) & (\¢ Cc\$, & \mathsf{Accept}), \\ (2) & (\¢ \cdot a^*, & abb \to D), & (6) & (\¢ Dd\$, & \mathsf{Accept}), \\ (3) & (\¢ \cdot a^*, & aCb \to C), & (7) & (\¢ c\$, & \mathsf{Accept}), \\ (4) & (\¢ \cdot a^*, & aDbb \to D), & (8) & (\¢ d\$, & \mathsf{Accept}). \end{array}$$

2.2 Properties and Lemmata

In this section some well known properties and lemmata on restarting automata are presented.

First, as in every cycle the length of the tape is reduced by at least one, it is easily seen that all restarting automata only accept languages from the classes CSL and NP. Deterministic restarting automata accept only certain languages from the classes DCSL (deterministic context-sensitive language) and P.

Lemma 2.2.1.

- $\mathcal{L}(\mathsf{RLWW}) \subseteq \mathsf{NP} \cap \mathsf{CSL}$.
- $\mathcal{L}(\mathsf{det\text{-}RLWW}) \subseteq \mathsf{P} \cap \mathsf{DCSL}$.

Proof. An accepting computation for an input of length n has at most n cycles. Therefore a (deterministic) Turing machine can simulate a (deterministic) RRWW-automaton in time $O(n^2)$. The Turing machine needs only linear space. Thus the inclusions hold. □

Two very useful properties of restarting automata are the so called error and correctness preserving properties. They state that a word not belonging to the language accepted by an automaton is never reduced to a word belonging to the language, and that in an accepting computation each word derived belongs to the language, too.

Proposition 2.2.2 (Error Preserving Property)**.**

Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an RLWW*-automaton, and let u, v be words over its input (tape) alphabet Σ (Γ). If $q_0¢u\$ \vdash_M^{c^*} q_0¢v\$$ holds and $u \notin L(M)$ ($L_C(M)$), then $v \notin L(M)$ ($L_C(M)$), either.*

Proposition 2.2.3 (Correctness Preserving Property)**.**

Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an RLWW*-automaton, and let u, v be words over its* input *alphabet Σ. If $q_0¢u\$ \vdash_M^{c^*} q_0¢v\$$ is an initial segment of an* accepting *computation of M, then $v \in L(M)$.*

There is a stronger notion of correctness preservation: If for a word $w \in L_C(M)$ each computation of the restarting automaton M is an accepting computation, M is called strongly correctness preserving.

Definition 2.2.4. *Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an* RLWW*-automaton. M is called (strongly) correctness preserving, if for all words $u \in L_C(M)$ and all cycles $q_0¢u\$ \vdash_M^c q_0¢v\$$ it follows that $v \in L_C(M)$.*

Observe that every deterministic restarting automaton is strongly correctness preserving.

Proposition 2.2.5 (Pumping Lemma)**.**

For any RLWW*-automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, there exists a constant p such that the following holds: Assume that $q_0¢uvw\$ \vdash_M^c q_0¢uv'w\$$, where $u = u_1u_2u_3$ and $|u_2| = p$. Then there exists a factorization $u_2 = z_1z_2z_3$ such that z_2 is non-empty, and*

$$q_0¢u_1z_1(z_2)^iz_3u_3vw\$ \vdash_M^c q_0¢u_1z_1(z_2)^iz_3u_3v'w\$$$

holds for all $i \geq 0$ that is, z_2 is a 'pumping factor' in the above cycle. Similarly, such a pumping factor can be found in any factor of length p of w. Such a pumping factor can also be found in any factor of length p of a word accepted in a tail computation.

In [MS04] we have shown that nonforgetting restarting automata (see chapter 3) without auxiliary symbols accept only regular languages over a unary alphabet. This result is valid for all types of restarting automata without auxiliary symbols. Here we give the version for "normal" restarting automata, to prove it we first show a result about RRWW-automata that are restricted to perform their rewrites at the right border marker.

Lemma 2.2.6. *Let M be an* RRWW*-automaton M such that M performs all of its rewrites while seeing the right border marker* \$ *in its window. Then $L(M)$ is regular.*

Proof. As the automaton performs all of its rewrites while seeing the right border marker \$, this RRWW-automaton is in fact an RWW-automaton.

We will construct an NFA A for $L(M)$. If $x_0 \in L(M)$, then there exists an accepting computation starting from x_0, which can be described by a sequence of cycles:

$$x_0 \vdash_M^c x_1 \vdash_M^c x_2 \ldots \vdash_M^c x_n \vdash_M^c \mathsf{Accept}.$$

As the rewrites of M are performed only at the right end of the tape, there exists a factorization of $x_0 = xu_nu_{n-1}\ldots u_2u_1$, such that $x_i = xu_n\ldots u_{i+1}v_i$, $0 < i < n$, and $x_n = xv_n$ hold. Here $u_iv_{i-1}\$ \to v_i\$$ is the ith rewrite step. It is required that $v_0 = \varepsilon$ holds.

A simulates the regular conditions of all meta-instructions simultaneously in its finite control. As M performs all its rewrite steps at the right border, the input x_0 is divided into two parts: a prefix x which is not changed during the sequence of cycles, and a suffix $u_nu_{n-1}\ldots u_2u_1$ which is rewritten step-by-step into v_n during this sequence.

The sequence of cycles is simulated from right to left as follows. First A guesses an accepting meta-instruction. Then it checks for the prefix $\mathfrak{c}x$ of it and remembers the missing suffix $v_n\$$. This suffix needs to be a right-hand side of a rewrite step $u_nv_{n-1}\$ \to v_n\$$ of a meta-instruction $(E_n, u_nv_{n-1}\$ \to v_n\$)$ such that $\mathfrak{c}x \in E_n$ holds. That is, the meta-instruction is applicable to $q_0\mathfrak{c}xu_nv_{n-1}\$$.

Then A looks for the prefix u_n of this rewrite instruction. The remaining suffix $v_{n-1}\$$ again has to be the right-hand side of the rewrite step of a meta-instruction $(E_{n-1}, u_{n-1}v_{n-2}\$ \to v_{n-1}\$)$, such that $\mathfrak{c}xu_n \in E_{n-1}$ holds. This is repeated until the whole input is consumed and A has found a meta-instruction that is applicable to $q_0\mathfrak{c}x_1\$ = q_0\mathfrak{c}xu_nu_{n-1}\ldots u_2u_1\$$, transferring it to $q_0\mathfrak{c}x_0\$ = q_0\mathfrak{c}xu_nu_{n-1}\ldots u_2v_1\$$.

Thus A accepts the input x_0 if and only if it finds an accepting sequence of cycles of M. □

Theorem 2.2.7. [MS04] *A language over a unary alphabet is regular if and only if it is accepted by an* RLW*-automaton.*

Proof. Each regular language can be accepted by an RLW-automaton in one cycle, so $\mathsf{REG} \subseteq \mathcal{L}(\mathsf{RLW})$ holds.

The other direction is a bit more complicated. An RLW-automaton can be simulated by a RRW-automaton (Theorem 2.1.4). A restarting automaton without auxiliary symbols can only delete some symbols when the tape alphabet is unary. Thus over a unary alphabet $\Sigma = \{a\}$, RRW-automata are in fact only RR automata. Next it does not matter where the rewrite occurs. Each meta-instruction of an RR-automaton of the form $(\mathfrak{c}E_1, a^l \to \varepsilon, E_2\$)$ checks the same conditions as the meta-instruction $(\mathfrak{c}E_1 \cdot E_2, a^l\$ \to \$)$ of an R-automaton. Thus over a unary alphabet each RR-automaton can be simulated by an R-automaton. The only difference is that the R-automaton needs a window of size $k + 1$ to see the right border marker and delete up to k symbols.

Now the assertion follows from Lemma 2.2.6. □

As the unary language L_{expo} is not regular, this theorem gives an easy separation language for restarting automata.

Corollary 2.2.8. *The language $L_{\mathsf{expo}} = \{a^{2^n} \mid n \geq 0\}$ is not accepted by any* RLW*-automaton M.*

Next we give a result from [Ott06], which describes the role of auxiliary symbols in restarting automata.

Theorem 2.2.9. [Ott06] *A language L is accepted by a (deterministic)* RLWW*-automaton if and only if there exists a (deterministic)* RLW*-automaton M_1 and a regular language E such that $L = L(M_1) \cap E$.*

This characterization yields the following closure property.

Corollary 2.2.10. [Ott06] *The language classes $\mathcal{L}$(RLWW), $\mathcal{L}$(RRWW), and $\mathcal{L}$(RWW) and their deterministic counterparts are closed under the operation of intersection with regular languages.*

As a consequence the following result is obtained from Theorem 2.2.9.

Corollary 2.2.11. [Ott06] *The language class $\mathcal{L}$(RLWW) is reducible in linear time and constant space to the language class $\mathcal{L}$(RLW).*

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an RLWW-automaton. Take $M_1 = (Q, \Gamma, \Gamma, ¢, \$, q_0, k, \delta)$ and φ the identity mapping on Σ^* This mapping is obviously a reduction from the language $L(M)$ to the language $L(M_1)$. □

This reduction works as well for $\mathcal{L}$(RRWW) and $\mathcal{L}$(RWW) and their deterministic counterparts.

In the next paragraph we recall a reduction from RLW-automata to RL-automata from [JOMP04] (see also [Ott06]). All letters are encoded in such a way that we can get every word from any longer word by just deleting some symbols.

Let $\Gamma_1 = \{a_1, \ldots, a_m\}$ be a finite alphabet, let m, k be positive integers, and let $\Gamma_2 := \{0, 1, c, d\}$. We define an encoding $\varphi_{k,m} : \Gamma_1 \to \Gamma_2^+$ as follows:

$$a_i \mapsto c1^{m+1-i}0^i(cd1^{m+1}0^{m+1})^k, \quad (1 \leq i \leq m).$$

Then, for all $1 \leq i \leq m$,

$$|\varphi_{k,m}(a_i)| = (m+1) \cdot (2k+1) + 2k + 1 = (m+2) \cdot (2k+1).$$

The encoding $\varphi_{k,m}$ is naturally extended to strings by taking

$$\varphi_{k,m}(x_1 x_2 \ldots x_n) := \varphi_{k,m}(x_1) \ldots \varphi_{k,m}(x_n) \text{ for all } x_1, \ldots, x_n \in \Gamma_1,\ n \geq 0.$$

Observe that $\varphi_{k,m} : \Gamma_1^* \to \Gamma_2^*$ is indeed an encoding that is, it is an injective mapping. It has the following important property.

Lemma 2.2.12. [JOMP04]
Let $u \in \Gamma_1^k$ and $v \in \Gamma_1^*$ with $|v| < k$. Then $\varphi_{k,m}(v)$ is a scattered subword of $\varphi_{k,m}(u)$.

Based on this encoding another reduction is obtained.

Theorem 2.2.13. [JOMP04] *If a language L is accepted by a (deterministic)* RLW*-automaton M with tape alphabet Γ_1 of size m and read/write window of size k, then there exists a (deterministic)* RL*-automaton M' which accepts the language $\varphi_{k,m}(L) \subseteq \Gamma_2^*$.*

Obviously each encoding of the form $\varphi_{k,m}$ can be computed in linear time and constant space. Thus, the following result holds (see [JOMP04] or [Ott06]).

Corollary 2.2.14. *The language class $\mathcal{L}$(RLW) is reducible in linear time and constant space to the language class $\mathcal{L}$(RL).*

Again, an analogous result holds for the classes $\mathcal{L}$(RRW) and $\mathcal{L}$(RW) and the corresponding deterministic classes.

With these reductions and the language $L_{\mathsf{copy}} = \{\omega\#\omega | \omega \in \{a,b\}^*\}$ it can be shown that $\mathcal{L}$(R) contains languages that are not growing context-sensitive(see [JOMP04]).

2.3 Monotone Restarting Automata

In [JMPV97b] the notion of monotonicity is introduced for restarting automata. We use the definitions from [JMPV07], which are a bit more general. Again, many definitions and results and their proofs are taken from [Ott03].

Definition 2.3.1. *Let M be an* RLWW*-automaton. Each computation of M can be described by a sequence of cycles $C_1, C_2 \ldots, C_n$, where C_n is the last cycle, which is followed by the tail of the computation. Each cycle C_i of this computation contains a unique configuration of the form $\mathfrak{c}xquy\$$ such that q is a state and $(q', v) \in \delta(q, u)$ is the* Rewrite *step that is applied during this cycle.*

By $D_r(C_i)$ we denote the right distance $|y\$|$ of this cycle, and $D_l(C_i) := |\mathfrak{c}x|$ is the left distance of this cycle. The sequence of cycles $C_1, C_2, \ldots, C_n$ is called monotone *if $D_r(C_1) \geq D_r(C_2) \geq \ldots \geq D_r(C_n)$ holds.*

A computation of M is called monotone if the corresponding sequence of cycles is monotone. Observe that the tail of the computation is not taken into account here.

Finally, the RLWW*-automaton M is called* monotone *if each of its computations that starts from an initial configuration is monotone.*

The prefix mon- is used for monotone restarting automata.

In this section we will describe the expressive power of the various types of monotone restarting automata and give example languages to separate them. First we present a result for the most powerful monotone type of restarting automata.

Theorem 2.3.2. [JMPV99] $\mathcal{L}(\mathsf{mon\text{-}RLWW}) = \mathcal{L}(\mathsf{mon\text{-}RRWW}) = \mathcal{L}(\mathsf{mon\text{-}RWW}) = \mathsf{CFL}$.

A corresponding result holds for deterministic monotone restarting automata. With the characterization of the deterministic context-free languages by LR(0)-grammars (see, e.g., [HU79]) it can be shown that already monotone det-R-automata accept all deterministic context-free languages, which yields the following result.

Theorem 2.3.3. [JMPV97b, JMPV97a, JMPV98, JMPV99]
For all types $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RWW}, \mathsf{RRWW}\}$, $\mathcal{L}(\mathsf{det\text{-}mon\text{-}X}) = \mathsf{DCFL}$.

In [JMOP05] the following result about deterministic monotone restarting automata is shown.

Theorem 2.3.4. [JMOP05]
$\mathcal{L}(\text{det-mon-RL}) = \mathcal{L}(\text{det-mon-RLWW})$

With the language from Example 2.1.3 it can be shown that $\text{DCFL} \subsetneq \mathcal{L}(\text{det-mon-RL})$ holds. Thus, in the deterministic case, RL(W)(W)-automata are more powerful than RR(W)(W)-automata.

The following languages are used to separate the different types of monotone restarting automata.

Lemma 2.3.5.

$$
\begin{array}{lcll}
L_1 & := & \{\, a^n b^n c \mid n \geq 0 \,\} \cup \{\, a^n b^{2n} d \mid n \geq 0 \,\} & \in \mathcal{L}(\text{mon-RR}) \setminus \mathcal{L}(\text{RW}), \\
L_2 & := & \{\, a^n b^n \mid n \geq 0 \,\} \cup \{\, a^n b^m \mid m > 2n \geq 0 \,\} & \in \mathcal{L}(\text{mon-RWW}) \setminus \mathcal{L}(\text{RLW}), \\
L_3 & := & \{f, ee\} \cdot \{\, a^n b^n \mid n \geq 0 \,\} \cup \{g, ee\} \cdot \{\, a^n b^m \mid m > 2n \geq 0 \,\} & \in \mathcal{L}(\text{mon-RW}) \setminus \mathcal{L}(\text{RL}), \\
L_4 & := & \{\, a^n b^m \mid 0 \leq n \leq m \leq 2n \,\} & \in \mathcal{L}(\text{mon-R}) \setminus \mathcal{L}(\text{det-RLW}).
\end{array}
$$

Together with Theorem 2.3.4 these facts yield all the relations displayed in figure 2.2.

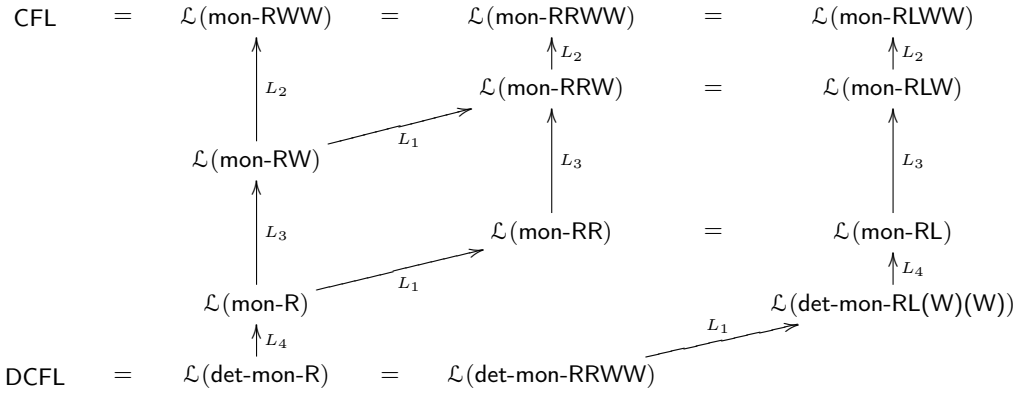

Figure 2.2: The taxonomy of monotone restarting automata

Now we turn to the decidability of monotonicity. For the following theorem a generalization of the simulation of monotone restarting automata by pushdown automata is used together with the fact that the set of non-monotone computations is regular.

Theorem 2.3.6. [JMPV07] *It is decidable whether a given* RRWW*-automaton is monotone.*

By Theorem 2.1.4 we can associate with each RLWW-automaton M an RRWW-automaton M_R such that, in each computation, M_R executes the same sequence of cycles as M. Hence, M is monotone if and only if M_R is. Thus, Theorem 2.3.6 holds for RLWW-automata in general.

2.4 Deterministic Restarting Automata

In this section we will regard deterministic restarting automata. They are more limited than the non-deterministic ones, but they are more often used in practice. Deterministic RRWW-automata

recognize exactly the Church-Rosser languages, so we first give the main properties of this language class here.

Definition 2.4.1. *Let Σ be a finite alphabet. A* string-rewriting system *R on Σ is a subset of $\Sigma^* \times \Sigma^*$. It induces several binary relations on Σ^*:*

- *The* single-step reduction relation $\rightarrow_R := \{ (u\ell v, urv) \mid u, v \in \Sigma^*, (\ell \rightarrow r) \in R \}$ *is the most basic of these.*
- *The* reduction relation $\rightarrow_R^*$ *induced by R is the reflexive and transitive closure of $\rightarrow_R$.*

If $u \rightarrow_R^* v$, then u is an ancestor *of v, and v is a* descendant *of u. If there is no $v \in \Sigma^*$ such that $u \rightarrow_R v$ holds, then the string u is called* irreducible *(mod R). By $\mathrm{IRR}(R)$ we denote the set of all irreducible strings. If R is finite, then $\mathrm{IRR}(R)$ is obviously a regular language. The string-rewriting system R is called*

- length-reducing *if $|\ell| > |r|$ holds for each rule $(\ell \rightarrow r) \in R$,*
- confluent *if, for all $u, v, w \in \Sigma^*$, $u \rightarrow_R^* v$ and $u \rightarrow_R^* w$ imply that v and w have a common descendant.*

If a string-rewriting system R is length-reducing, then each reduction sequence starting with a string of length n has itself at most length n. If, in addition, R is confluent, then each string $w \in \Sigma^*$ has a unique irreducible descendant $\hat{w} \in \mathrm{IRR}(R)$, which can be computed from w in linear time (see, e.g., [BO93]). This observation was one of the main reasons to introduce the Church-Rosser languages in [MNO88, Nar84].

Definition 2.4.2. *A language $L \subseteq \Sigma^*$ is a* Church-Rosser language *if there exists an alphabet $\Gamma \supsetneq \Sigma$, a finite, length-reducing, confluent string-rewriting system R on Γ, two strings $t_1, t_2 \in (\Gamma \setminus \Sigma)^* \cap \mathrm{IRR}(R)$, and a symbol $Y \in (\Gamma \setminus \Sigma) \cap \mathrm{IRR}(R)$ such that, for all $w \in \Sigma^*$, $t_1 w t_2 \rightarrow_R^* Y$ if and only if $w \in L$.*

Definition 2.4.3. *A context-sensitive grammar $G = (\Sigma, N, S, P)$ is called a* growing context-sensitive *grammar, if, for each production $(l \rightarrow r) \in P$, $|l| < |r|$ or $l = S$ holds. Further S must not appear on the right-hand side of any production. A language is called* growing context-sensitive *if it can be generated by a growing context-sensitive grammar.* GCSL *denotes the class of all growing context-sensitive languages.*

By CRL we denote the class of Church-Rosser languages. It is known that the inclusions DCFL $\subset$ CRL $\subset$ GCSL $\subset$ CSL are proper, and that the classes CRL and CFL are incomparable under set inclusion [BO98, MNO88, Nar84]. The Church-Rosser languages have been characterized by certain types of two-pushdown automata.

Definition 2.4.4. *A* two-pushdown automaton (TPDA) *with* pushdown windows of size *k is a nondeterministic automaton $P = (Q, \Sigma, \Gamma, \delta, k, q_0, \bot, t_1, t_2, F)$, where Q is the finite set of states, Σ is the finite input alphabet, Γ is the finite tape alphabet containing Σ, $q_0 \in Q$ is the initial state, $\bot \in \Gamma \setminus \Sigma$ is the bottom marker of the pushdown stores, $t_1, t_2 \in (\Gamma \setminus \Sigma)^*$ is the preassigned contents of the first and second pushdown store, respectively, $F \subseteq Q$ is the set of final (or halting) states, and δ is the transition relation. To each triple (q, u, v), where $q \in Q$ is a state, $u \in \Gamma^k \cup \{\bot\} \cdot \Gamma^{<k}$ is the contents of the topmost part of the first pushdown store, and $v \in \Gamma^k \cup \Gamma^{<k} \cdot \{\bot\}$ is the contents of the topmost part of the second pushdown store, it associates a finite set of triples from $Q \times \Gamma^* \times \Gamma^*$. The automaton P is a* deterministic two-pushdown automaton (DTPDA)*, if δ is a (partial) function.*

A configuration of a (D)TPDA is described by uqv, where $q \in Q$ is the actual state, $u \in \Gamma^*$ is the contents of the first pushdown store with the first symbol of u at the bottom and the last symbol of u at the top, and $v \in \Gamma^*$ is the contents of the second pushdown store with the last symbol of v at the bottom and the first symbol of v at the top. For an input string $w \in \Sigma^*$, the corresponding initial configuration is $\bot t_1 q_0 w t_2 \bot$. The computation relation of a (D)TPDA P is denoted by $\vdash_P^*$. The (D)TPDA P accepts with empty pushdown stores that is, $L(P) := \{ w \in \Sigma^* \mid \bot t_1 q_0 w t_2 \bot \vdash_P^* q \text{ for some } q \in F \}$ is the *language accepted by* P.

Observe that the input is provided to a TPDA as the initial contents of its second pushdown store, and that in order to accept a TPDA is required to empty its pushdown stores. Thus, it is forced to consume its input completely.

A (D)TPDA is called *shrinking* if there exists a *weight-function* $\varphi : Q \cup \Gamma \to \mathbb{N}_+$ such that, for all transitions $(p, u', v') \in \delta(q, u, v)$, $\varphi(u'pv') < \varphi(uqv)$. Here φ is extended to a morphism $\varphi : (Q \cup \Gamma)^* \to \mathbb{N}$ in the obvious way. A (D)TPDA is called *length-reducing* if $|u'v'| < |uv|$ holds for all transitions $(p, u', v') \in \delta(q, u, v)$. Obviously, the length-reducing TPDA is a special case of the shrinking TPDA.

The following characterizations have been established in [BO98, Nie02, NO98, NO05].

Theorem 2.4.5.

(a) *A language is Church-Rosser, if and only if it is accepted by a shrinking* DTPDA*, if and only if it is accepted by a length-reducing* DTPDA*.*

(b) *A language is growing context-sensitive, if and only if it is accepted by a shrinking* TPDA*, if and only if it is accepted by a length-reducing* TPDA*.*

Based on this result the following theorem, which gives a characterization of the Church-Rosser languages by deterministic restarting automata with auxiliary symbols, has been established.

Theorem 2.4.6. [NO00, NO03]
$$\begin{array}{lccccc} \text{(a)} & \mathsf{CRL} & = & \mathcal{L}(\text{det-RWW}) & = & \mathcal{L}(\text{det-RRWW}). \\ \text{(b)} & \mathsf{GCSL} & \subseteq & \mathcal{L}(\mathsf{RWW}) & \subseteq & \mathcal{L}(\mathsf{RRWW}). \end{array}$$

Lemma 2.4.7. [MNO88] *The language* $L_{\mathsf{expo}} = \{a^{2^n} | n \in \mathbb{N}\}$ *is a Church-Rosser language.*

Apart from the restarting automata with auxiliary symbols, the following inclusions have been established for deterministic restarting automata, where the separation languages are defined as follows:

$$\begin{aligned} L_6 &:= L_{6,1} \cup L_{6,2} \cup L_{6,3}, \\ L_{6,1} &:= \{ (ab)^{2^n - i} c (ab)^i \mid n \geq 0, 0 \leq i \leq 2^n \}, \\ L_{6,2} &:= \{ (ab)^{2^n - 2i} (abb)^i \mid n \geq 1, 0 \leq i \leq 2^{n-1} \}, \\ L_{6,3} &:= \{ (abb)^{2^n - i} (ab)^i \mid n \geq 0, 0 \leq i \leq 2^n \}, \\ L_7 &:= L_{7,1} \cup L_{7,2}, \\ L_{7,1} &:= \{ a^{2^n - 2i} c a^i \mid n \geq 1, 0 \leq 2i < 2^n \}, \\ L_{7,2} &:= \{ a^i d a^{2^n - 2i} \mid n \geq 1, 0 \leq 2i < 2^n \}. \end{aligned}$$

The language L_6 is in $\mathcal{L}(\text{det-RR}) \backslash \mathcal{L}(\mathsf{RW})$ and L_7 is in $\mathcal{L}(\text{det-RW}) \backslash \mathcal{L}(\mathsf{RL})$.

It can be shown that the copy language $L_{\mathsf{copy}} = \{\omega \# \omega | \omega \in \{a, b\}^*\}$ is in $\mathcal{L}(\text{det-RLWW})$. By using the reductions from Theorem 2.2.9 and Theorem 2.2.13 it can be shown that there exist languages

$L'_{\mathsf{copy}} \in \mathcal{L}(\mathsf{det\text{-}RLW})$ and $\tilde{L}_{\mathsf{copy}} \in \mathcal{L}(\mathsf{det\text{-}RL})$, which are not in CRL. Remark that CRL is closed under the operations of intersection with regular sets and inverse morphisms [Nie02].

The exponential language together with these reductions can be used to separate $\mathcal{L}(\mathsf{det\text{-}R})$ from DCFL, because CFL, as well as DCFL, is closed under intersection with regular sets and inverse morphisms [HU79], too.

Corollary 2.4.8. *The language classes* $\mathcal{L}(\mathsf{det\text{-}R})$ *and* CFL *are incomparable with respect to inclusion.*

Obviously $\mathcal{L}(\mathsf{det\text{-}R})$ is not contained in $\mathcal{L}(\mathsf{det\text{-}mon\text{-}RL})$, because the latter class is included in CFL.

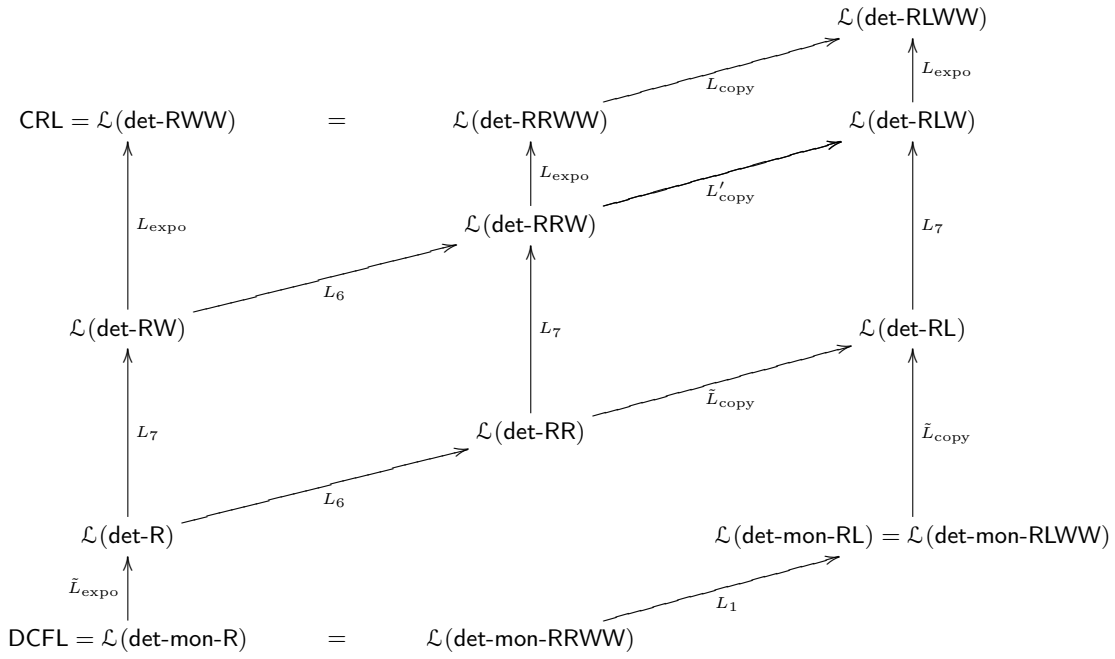

Figure 2.3: The taxonomy of deterministic restarting automata

Theorem 2.4.9. [Ott06] $\mathcal{L}(\mathsf{det\text{-}mon\text{-}RL}) \subsetneq \mathsf{CRL}$

This result can be shown just as Theorem 2.4.6, since a deterministic monotone RL-automaton can be simulated by a shrinking TPDA just like a deterministic RR-automaton.

On the other hand the language $L_{\mathsf{pal}} = \{\omega\omega^R \mid \omega \in \{a,b\}^*\}$ is included in $\mathcal{L}(\mathsf{mon\text{-}RWW})$ while in [JL02] it is shown that this language is not a Church-Rosser language. With the reduction from Corollary 2.2.14, we get the following result.

Corollary 2.4.10. *The language classes* $\mathcal{L}(\mathsf{mon\text{-}R})$ *and* CRL *are incomparable with respect to inclusion.*

2.5 Weakly Monotone Restarting Automata

The class GCSL is included in $\mathcal{L}$(RWW), see Theorem 2.4.6. However, in the language class $\mathcal{L}$(R) there are already languages which are not growing context-sensitive. Thus we have the following result:

Corollary 2.5.1. [JLNO04] *The class* GCSL *is properly contained in* $\mathcal{L}$(RWW).

The class of growing context-sensitive languages (GCSL) can be characterized by restarting automata that fulfill a weaker form of monotonicity [JLNO04], the so-called weakly monotone restarting automata. They are defined as follows.

Definition 2.5.2. *Let M be an* RLWW*-automaton, and let $c \geq 0$ be an integer. A sequence of cycles $C_1, C_2, \ldots, C_n$ of M is called* weakly c-monotone *if $D_r(C_{i+1}) \leq D_r(C_i) + c$ holds for all $i = 1, \ldots, n-1$.*

A computation of M is called weakly c-monotone *if the corresponding sequence of cycles is weakly c-monotone. Observe that the tail of the computation is not taken into account.*

Finally, the RLWW*-automaton M is called* weakly c-monotone *if, for each restarting configuration $q_0 \text{¢} w \$$ of M, each computation of M starting with $q_0 \text{¢} w \$$ is weakly c-monotone.*

The RLWW*-automaton M is called* weakly monotone *if it is weakly c-monotone for some integer constant $c \geq 0$. The prefix* wmon- *is used to denote classes of weakly monotone restarting automata.*

Note that, in contrast to monotone restarting automata, for weakly monotone restarting automata every computation from every restarting configuration needs to be weakly monotone, not just those that start from initial configurations. This slightly different definition does not affect the accepted language, because without auxiliary symbols every restarting configuration is an initial configuration. With auxiliary symbols exactly GCSL is accepted independent of the definition of weak monotonicity, because a TPDA simulating a weakly monotone restarting automaton always starts in an initial configuration.

The advantage of this stronger definition is that weak monotonicity becomes decidable.

Theorem 2.5.3. [MO05] *It is decidable whether a given* RLWW*-automaton M is weakly monotone. In the affirmative the smallest integer c can be determined for which M is weakly c-monotone.*

Deterministic RRWW-automata can only perform a rewrite step if they have seen at least one symbol of the string v from the last rewrite step $u \to v$. Otherwise they would have done this rewrite in the previous cycle. Thus the following lemma holds.

Lemma 2.5.4. *Each deterministic* RRWW*-automaton is weakly monotone.*

This result is not valid for det-RLWW-automata. For example the deterministic restarting automaton for the copy language is not weakly monotone. We will see that weak monotonicity is a strong restriction for various kinds of restarting automata.

Lemma 2.5.5. [MO05] *An* RLWW*-automaton $M = (Q, \Sigma, \Gamma, \text{¢}, \$, q_0, k, \delta)$ is weakly monotone, if and only if it is weakly c-monotone for some constant $c < |Q|^2 \cdot |\Gamma|^k + 2k$.*

The proof of Theorem 2.4.6 (a) can be generalized to a proof of the following characterization for weakly monotone restarting automata with auxiliary symbols.

Theorem 2.5.6. [Nie02] GCSL = $\mathcal{L}$(wmon-RWW) = $\mathcal{L}$(wmon-RRWW) = $\mathcal{L}$(wmon-RLWW).

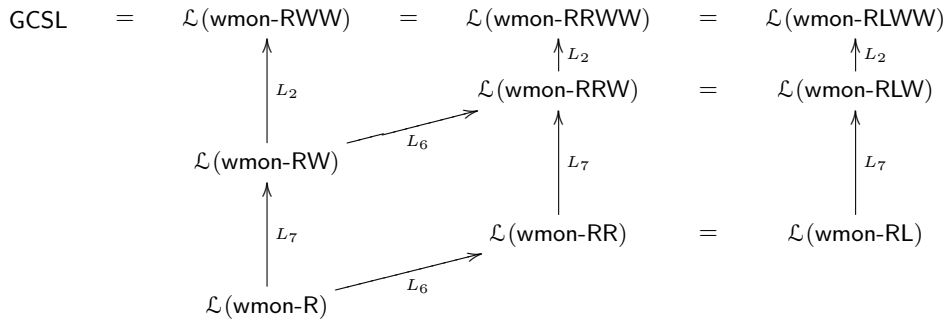

Figure 2.4: The taxonomy of weakly monotone restarting automata

These results lead to the following diagram.

For all types of restarting automata $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RWW}, \mathsf{RRWW}\}$ monotonicity is a stronger restriction than weak monotonicity and the same holds for determinism. Because of Corollaries 2.4.8 and 2.4.10, we obtain proper inclusions for each of these language classes, and even the language class $\mathcal{L}(\mathsf{wmon\text{-}R})$ is not included in any language class obtained by deterministic or monotone restarting automata. On the other hand $\mathcal{L}(\mathsf{det\text{-}RL})$ is not included in GCSL because of the language L_{copy} and its reductions (see Section 2.4).

2.6 Unrestricted Restarting Automata

In this section we will regard nondeterministic non monotone restarting automata. For restarting automata without auxiliary symbols the trivial inclusions obtained in the previous sections hold and they are proper, which can be shown using almost the same separation languages. It is an open problem, which other inclusions exist and (if any) which of them are proper. The main question displayed in Figure 2.5 is whether $\mathcal{L}(\mathsf{RWW}) = \mathcal{L}(\mathsf{RRWW})$ holds or not. However, all results about the equivalence of restarting automata with auxiliary symbols are obtained using their equivalence to the common languages classes GCSL, CRL, CFL and DCFL.

In all but the monotone case, the separation language L_2 can be replaced by the exponential language, L_7 is already some kind of exponential language. With the encoding from Theorem 2.2.13 one can find a language that is in $\mathcal{L}(\mathsf{R})$, but as the exponential language and all its reductions are not monotone this language is not monotone either. The first separation language for this case was described in [JMPV97b] and named L_5'.

It follows from Theorem 2.2.9 that RRW and RW-automata are not closed under intersection with regular sets and with Corollary 2.2.14 it follows that there are languages in $\mathcal{L}(\mathsf{R})$ that are not in GCSL and this yields the following Corollary.

Corollary 2.6.1. *The language class* $\mathcal{L}(\mathsf{R})$ *is incomparable to* CFL, CRL *and* GCSL *with respect to inclusion.*

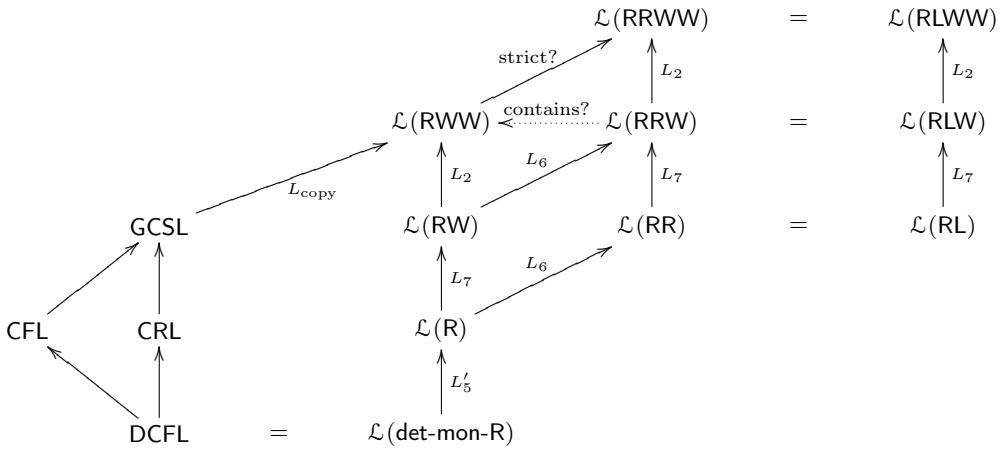

Figure 2.5: The taxonomy of nondeterministic restarting automata

2.7 Generalizations of Restarting Automata

One generalization of the standard restarting automaton, nonforgetting restarting automata, is considered in the next chapter, which is one of the main parts of this dissertation. But there are other generalizations, which have been studied before. We will mention a few of them here.

Restarting automata are defined as length reducing automata, as they have to reduce the length of the tape in every rewrite step. A slightly different model is the shrinking restarting automaton, first introduced in [OJ03] and then studied in [JO07]. These restarting automata are less restricted than length reducing restarting automata.

Definition 2.7.1. *A shrinking restarting automaton* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ *does not need to shorten its tape in each rewrite step. Instead there must exist a weight function* $\varphi : \Gamma \to \mathbb{N}_+$*, such that, for every rewrite step* $u \to v$*,* $\varphi(u) > \varphi(v)$ *holds.*

The language class that is accepted by shrinking RLWW*-automata is called* $\mathcal{L}$(sh-RLWW)

It is clear that with the weight function $\omega \to |\omega|$ every length reducing restarting automata is shrinking. On the other hand shrinking restarting automata are in some cases more powerful than length reducing restarting automata.

First, we present some results where shrinking restarting automata are not more expressive. The following Lemma is obvious, because the property of being shrinking can only have a benefit, if the restarting automaton can alter the tape content.

Lemma 2.7.2. $\mathcal{L}(\text{sh-R}) = \mathcal{L}(\text{R})$ *and* $\mathcal{L}(\text{sh-RR}) = \mathcal{L}(\text{RR})$.

Shrinking RRWW-automata are not more powerful than their length-reducing counterparts if they are deterministic, monotone, deterministic and monotone or weakly monotone. This can be proven with the corresponding classical automata model.

Theorem 2.7.3.

$$\begin{aligned} \mathsf{CRL} &= \mathcal{L}(\text{det-sh-RWW}) &&= \mathcal{L}(\text{det-sh-RRWW}).\\ \mathsf{CFL} &= \mathcal{L}(\text{mon-sh-RWW}) &&= \mathcal{L}(\text{mon-sh-RRWW}).\\ \mathsf{DCFL} &= \mathcal{L}(\text{det-mon-sh-R}) &&= \mathcal{L}(\text{det-mon-sh-RRWW}).\\ \mathsf{GCSL} &= \mathcal{L}(\text{sh-wmon-RWW}) &&= \mathcal{L}(\text{sh-wmon-RRWW}). \end{aligned}$$

It is an open question whether unrestricted sh-RRWW-automata are more powerful than RRWW-automata.

But in [JO07] it has been shown that the class of shrinking RRWW-automata yields a very robust language class, because the property of being nonforgetting, or the ability to perform many rewrites per cycle, does not increase the expressive power of shrinking RRWW-automata.

Definition 2.7.4. *A restarting automaton with up to (exactly) $c > 1$ rewrites per cycle is allowed to do up to $c > 1$ (must perform exactly $c > 1$) rewrite steps per cycle. A meta-instruction for an* RRWW*-automaton with (up to) $c > 1$ rewrites per cycle is of the form*

$$(\mathrm{¢}E_1,\ u_1 \to v_1,\ E_2,\ u_2 \to v_2,\ \ldots, u_c \to v_c,\ E_c\$)$$

Definition 2.7.5. *A nonforgetting restarting automaton,* nf-RLWW*-automaton for short, is a nine tuple $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta, Q_R)$, where:*

Q is a finite set of states, Σ is a finite input alphabet, $\Gamma \supseteq \Sigma$ is a finite tape alphabet, $¢, \$ \notin \Gamma$ are border markers, $q_0 \in Q$ is the initial state, $k \geq 1$ is the window size, $\delta : Q \times \mathcal{P}C^{(k)} \to \mathfrak{P}((Q \times (\{\mathsf{MVR}, \mathsf{MVL}\} \cup \mathcal{P}C^{\leq(k-1)})) \cup \{\mathsf{Restart}, \mathsf{Accept}\})$ is the transition relation and $Q_R \subset Q$ is the set of restarting states that contains q_0.

Nonforgetting restarting automata have the ability to restart in any of the restarting states $q \in Q_R$ instead of restarting in the initial state q_0.

In [JO07] it has also been shown that shrinking RRWW-automata are characterized by finite-change automata (FC). Finite-change-automata are a restricted type of Turing machines and they accept only deterministic context-sensitive languages. They were introduced by von Braunmühl and Verbeek [vBV79].

Proposition 2.7.6. [JO07] $\mathcal{L}(\text{sh-RRWW}) = \mathcal{L}(\text{nf-sh-RRWW}) = \mathcal{L}(\mathsf{FC}) \subseteq \mathsf{DCSL}$.

In this work a (nonforgetting) restarting automaton (with c rewrites per cycle) is called shrinking, if all of its rewrites are shrinking.

Nonforgetting restarting automata are studied in detail in the next chapter. To show the expressive power of restarting automata with c rewrites per cycle we give a small example. There exists an R-automaton with two rewrites per cycle for the Language $L_{\mathsf{copy}} = \{w\#w | w \in \{a, b\}^*\}$. Here it is required that an R-automaton with many rewrites per cycle restarts immediately after the last rewrite of the cycle.

Example 2.7.7. *The automaton M for L_{copy} is described by the following meta-instructions.*

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

The automaton deletes the first symbol from the tape and performs MVR*-steps until it sees the #, then it checks whether the first symbol after the # coincides with the symbol from the first delete. In the affirmative it deletes it and restarts. Otherwise it halts and rejects. The automaton accepts, when it sees only the separator on the tape.*

2.8 Correctness Preserving Restarting Automata

In this and the following sections the results about correctness preserving restarting automata from [MO08] by the author and Friedrich Otto are presented. First we restate the definition of (strongly) correctness preserving restarting automata from page 17.

Definition 2.8.1. *Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an* RLWW*-automaton. M is called (strongly) correctness preserving, if for all words $u \in L_C(M)$ and all cycles $q_0¢u\$ \vdash_M^c q_0¢v\$$ it follows that $v \in L_C(M)$.*

The following example shows the difference between correctness preserving and non correctness preserving restarting automata.

Example 2.8.2. Let $\tilde{L_1} := \{ a^n b^n c, a^n c b^n c \mid n \geq 0 \} \cup \{ a^n b^m d, a^n d b^m d \mid m > 2n \geq 0 \}$. Then $\tilde{L_1}$ is accepted by the RW-automaton M_1 that is given through the following sequence of meta-instructions:

(1)	$(¢ \cdot a^+, abb \to cb)$,	(4)	$(¢ \cdot a^+, adbbb \to db)$,
(2)	$(¢ \cdot a^+, abbb \to db)$,	(5)	$(¢ \cdot \{\varepsilon, ab, c, acb\} \cdot c \cdot \$, \mathsf{Accept})$,
(3)	$(¢ \cdot a^+, acbb \to cb)$,	(6)	$(¢ \cdot \{\varepsilon, abb, d, adbb\} \cdot b^+ \cdot d \cdot \$, \mathsf{Accept})$.

It is easily seen that $L(M_1) = \tilde{L_1}$ holds. Further, starting from the configuration $q_0¢a^n b^n c\$$ for a sufficiently large value of n, M_1 can execute the cycle $a^n b^n c \vdash_{M_1}^c a^{n-1} d b^{n-2} c$. As $a^n b^n c \in \tilde{L_1}$, while $a^{n-1} d b^{n-2} c \notin \tilde{L_1}$, we see that M_1 is not correctness preserving.

On the other hand, the language $\tilde{L_1}$ is accepted by the RR-automaton M_2 that is described through the following meta-instructions (see, e.g., [Ott06]):

(1)	$(¢ \cdot a^*, ab \to \varepsilon, b^* \cdot c \cdot \$)$,	(4)	$(¢ \cdot a^*, adbb \to d, b^+ \cdot d \cdot \$)$,
(2)	$(¢ \cdot a^*, acb \to c, b^* \cdot c \cdot \$)$,	(5)	$(¢ \cdot \{c, cc\} \cdot \$, \mathsf{Accept})$,
(3)	$(¢ \cdot a^*, abb \to \varepsilon, b^+ \cdot d \cdot \$)$,	(6)	$(¢ \cdot \{\varepsilon, d\} \cdot b^+ \cdot d \cdot \$, \mathsf{Accept})$.

The automaton M_2 is nondeterministic, but it is correctness preserving. Indeed, starting from a configuration of the form $q_0¢a^n b^n c\$$, M_2 may execute the rewrite step $abb \to \varepsilon$, thereby transforming the tape content into $¢a^{n-1} b^{n-2} c\$$, but it will then detect its error at the right end of the tape, where it encounters the symbol c. Analogous considerations apply to the other cases. On the other hand, it can be shown that $\tilde{L_1}$ is not accepted by any deterministic RRW-automaton.

Thus, we see that correctness preserving RR- (RRW-)automata are more expressive than deterministic RR-(RRW-)automata.

For R-, RW-, and RWW-automata we obtain the following result relating nondeterministic automata that are strongly correctness preserving to deterministic automata of the same type.

Theorem 2.8.3. *For any $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$, if $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ is a correctness preserving* RX*-automaton, then there exists a deterministic* RX*-automaton M' satisfying $L_C(M') = L_C(M)$ and $L(M') = L(M)$.*

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be a correctness preserving RX-automaton. We describe a deterministic RX-automaton $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k, \delta')$ such that $L_C(M') = L_C(M)$ holds. The automaton M' acts in the same way as M, but when scanning its tape from left to right it applies the first rewrite instruction that becomes applicable. In case several such instructions exist, the tie is broken based on a linear ordering of the rewrite instructions. Then M' is obviously a deterministic RX-automaton.

As M' has the same Accept instructions as M, and as all rewrite steps that M' can execute are also valid rewrite steps of M, we see that $L_C(M') \subseteq L_C(M)$ holds. So it remains to verify the converse inclusion.

Let $w \in \Gamma^*$. Assume that w is accepted by M, that is, $w \in L_C(M)$. If starting from the restarting configuration $q_0¢w\$$, M can execute a rewrite operation, then M' will definitely execute a rewrite operation in this situation, that is, we obtain a cycle of the form $w \vdash^c_{M'} w'$ for some $w' \in \Gamma^*$. From the definition of M' it follows that we also have the cycle $w \vdash^c_M w'$. However, as M is correctness preserving, this means that $w' \in L_C(M)$ holds. By induction it now follows that M' accepts on input w, that is, $L_C(M') = L_C(M)$. As M' and M have the same input alphabet, we obtain $L(M') = L(M)$, too. □

In the following we will extend this result to RLWW-automata. In fact, we consider two generalizations of these automata. First of all we consider shrinking RLWW-automata. In fact, it is easily seen from the proof of Theorem 2.8.3 that this theorem even holds for RWW- and RW-automata that are shrinking.

Secondly, we admit (shrinking) restarting automata that are allowed to perform up to c rewrite operations per cycle for some constant $c \geq 1$. Next we recall the following useful technical result which is a slight extension of Theorem 2.1.4.

Theorem 2.8.4. *For any* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$*, if* $M_L = (Q_L, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_L)$ *is a (shrinking)* RLX*-automaton that executes up to* $c \geq 1$ *rewrite steps per cycle, then there exists a (shrinking)* RRX*-automaton* $M_R = (Q_R, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_R)$ *that also executes up to* c *rewrite steps per cycle such that, for all* $u, v \in \Gamma^*$*,* $u \vdash^c_{M_L} v$ *if and only if* $u \vdash^c_{M_R} v$*. In particular,* $L_C(M_L) = L_C(M_R)$ *and* $L(M_L) = L(M_R)$*.*

Hence, correctness preserving RR-automata are as expressive as correctness preserving RL-automata, but as seen in Example 2.8.2 they are more expressive than deterministic RR-automata. Thus, Theorem 2.8.3 does not carry over to RR-automata. On the other hand, it is easily seen that deterministic RL-automata are more expressive than deterministic RR-automata, too; for example, one can easily design a deterministic RL-automaton for the language L_1 considered in Example 2.8.2. The following result now relates deterministic RL-automata to correctness preserving RL- (and therewith RR-) automata.

Theorem 2.8.5. *For any* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$*, if* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ *is a correctness preserving (shrinking)* RLX*-automaton that performs up to* $c \geq 1$ *rewrite steps per cycle, then there exists a (shrinking) deterministic* RLX*-automaton* M' *performing up to* c *rewrite steps per cycle such that* $L_C(M') = L_C(M)$ *and* $L(M') = L(M)$*.*

The proof is an adaptation of the proof of the corresponding result for t-sRL-automata presented in [MMOP06]. Theorem 2.8.4 enables us to describe any (shrinking) RLX-automaton that executes up to $c \geq 1$ rewrite steps per cycle by accepting meta-instructions and by generalized rewriting meta-instructions of the form

$$I = (¢ \cdot E_0, u_1 \to v_1, E_1, u_2 \to v_2, \ldots, E_{s-1}, u_s \to v_s, E_s \cdot \$),$$

where $1 \leq s \leq c$, $E_0, E_1, \ldots, E_s$ are regular languages, and $u_1 \to v_1, \ldots, u_s \to v_s$ are the rewrite steps executed by I. This meta-instruction can be applied to a word of the form $w = x_0u_1x_1u_2 \cdots x_{s-1}u_sx_s$ satisfying $x_i \in E_i$ for all $i = 0, \ldots, s$, and it yields the word $x_0v_1x_1v_2 \cdots x_{s-1}v_sx_s$.

Proof. Let M be a correctness preserving (shrinking) RLX-automaton that performs up to $c \geq 1$ rewrite steps per cycle. By the remark above M can be described through a finite sequence of meta-instructions $I_1, \ldots, I_i$. We will present a deterministic (shrinking) RLX-automaton M' recognizing the same language as M. First, for each $j = 1, \ldots, i$, we construct a finite-state acceptor A_j for the set of words to which meta-instruction I_j is applicable. The automaton M' will then proceed as follows:

1. M' scans the current word $w \in \Gamma^*$ on its tape from left to right simulating all the acceptors $A_1, \ldots, A_i$ in parallel. At the right sentinel M' knows which meta-instructions of M are applicable to the current word. If none is applicable, then M' halts and rejects; if any of the applicable meta-instructions is accepting, then M' halts and accepts. Otherwise, any correct application of any of the applicable meta-instructions will yield a word w' such that $w' \in L_C(M)$ if and only if $w \in L_C(M)$, as M is correctness preserving. Thus, M' simply chooses one of these applicable meta-instructions, e.g., the one with the smallest index. By I we denote this meta-instruction.
2. M' simulates an application of I on its current tape content.

It remains to show how M' simulates an application of I to the configuration $q_0¢w\$$. Let $w = y_1 \cdots y_n$, where $y_1, \ldots, y_n \in \Gamma$, and assume that

$$I = (¢ \cdot E_0, u_1 \to v_1, E_1, u_2 \to v_2, E_2, \ldots, E_{s-1}, u_s \to v_s, E_s \cdot \$),$$

where $1 \leq s \leq c$. M' must determine a factorization of the form $w = x_0u_1x_1u_2 \cdots x_{s-1}u_sx_s$ such that $x_i \in E_i$ for all $i = 0, \ldots, s$, and replace the factors $u_1, u_2, \ldots, u_s$ by the words $v_1, v_2, \ldots, v_s$, respectively. As w may have many such factorizations, M' must choose one of them deterministically. For this task M' will use finite-state acceptors $M_1, \ldots, M_s$ and $M_1^R, \ldots, M_s^R$, which accept the following regular languages:

$$\begin{array}{rclcrcl}
L(M_1) & = & E_0 \cdot u_1, & & (E_1 \cdot u_2 \cdot E_2 \cdot u_3 \cdots E_{s-1} \cdot u_s \cdot E_s)^R & = & L(M_1^R), \\
L(M_2) & = & E_1 \cdot u_2, & & (E_2 \cdot u_3 \cdots E_{s-1} \cdot u_s \cdot E_s)^R & = & L(M_2^R), \\
 & \vdots & & & & \vdots & \\
L(M_{s-1}) & = & E_{s-2} \cdot u_{s-1}, & & (E_{s-1} \cdot u_s \cdot E_s)^R & = & L(M_{s-1}^R), \\
L(M_s) & = & E_{s-1} \cdot u_s, & & (E_s)^R & = & L(M_s^R).
\end{array}$$

After step (1) above (that is, when choosing the meta-instruction I), M' is at the right sentinel. Now it scans its tape again, this time from right to left, thereby simulating the finite-state acceptors $M_1^R, \ldots, M_s^R$ in parallel. For each $1 \leq j \leq s$ and $0 \leq \ell \leq n-1$, let $q(j, \ell)$ denote the state of M_j^R after reading the word $y_n \cdots y_{\ell+1}$. When reaching the left sentinel, M' changes direction again. Now, while moving to the right, M' simulates the finite-state acceptor M_1. Simultaneously, it recomputes the internal states of all the acceptors $M_1^R, \ldots, M_s^R$ on the respective tape symbol that is, it runs these acceptors in reverse. This it can do due to the following technical result from [AHU69] (pages 212–213).

Lemma 2.8.6. *Let A be a deterministic finite-state acceptor. For each word x and each integer i, $1 \leq i \leq |x|$, let $q_A(x, i)$ be the internal state of A after processing the prefix of length i of x.*

Then there exists a deterministic two-way finite-state acceptor A' such that, for each input x and each $i \in \{2, 3, \ldots, |x|\}$, if A' starts its computation on x in state $q_A(x, i)$ with its head on the i-th symbol of x, then A' finishes its computation in state $q_A(x, i-1)$ with its head on the $(i-1)$-th symbol of x. During this computation A' only visits (a part of) the prefix of length i of x.

As meta-instruction I is applicable to the configuration $q_0 ¢ w \$$, w belongs to the set $E_0 \cdot u_1 \cdot E_1 \cdot u_2 \cdot E_2 \cdot u_3 \cdots E_{s-1} \cdot u_s \cdot E_s$. Hence, there is a smallest index ℓ_1 such that $y_1 \cdots y_{\ell_1} \in L(M_1)$ and $y_{\ell_1+1} \cdots y_n \in [L(M_1^R)]^R$. That is, after scanning $y_1 \cdots y_{\ell_1}$, the finite-state acceptor M_1 is in an accepting state, and simultaneously $q(1, \ell_1)$ is an accepting state of M_1^R. On reaching this position, M' replaces the suffix u_1 of $y_1 \cdots y_{\ell_1}$ by v_1, aborts the simulations of M_1 and M_1^R, and starts to simulate M_2 from its initial state. Now M' looks for an index $\ell_2 > \ell_1$ such that M_2 is in an accepting state after processing $y_{\ell_1+1} \cdots y_{\ell_2}$, and $q(2, \ell_2)$ is an accepting state of M_2^R. Once this position is reached, M' replaces the suffix u_2 by v_2, aborts the simulations of M_2 and M_2^R, and starts to simulate M_3. This process is then continued for $i = 3, 4, \ldots, s$. In this way, M' does indeed apply the meta-instruction I to its current tape content.

It is easy to see that the RLX-automaton M' constructed in the way described above is deterministic, and that it accepts the same languages as the given RLX-automaton M. □

2.9 The Error-Detection Distance

Here we introduce the notion of error-detection distance. It is a generalization of the correctness preserving property.

Definition 2.9.1. *Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an* RLWW*-automaton, and let i be a non-negative integer. We say that M has* error-detection distance *i, if, for all words $w \in L_C(M)$ and all partial computations $w \vdash_M^c w_1 \vdash_M^c \cdots \vdash_M^c w_m$, if $w_1 \notin L_C(M)$, then $m \leq i$. That is, if the first cycle $w \vdash_M^c w_1$ transforms the word $w \in L_C(M)$ into a word $w_1 \notin L_C(M)$, which means that M has made a mistake, then starting from the restarting configuration $q_0 ¢ w_1 \$$, M can execute at most $i - 1$ more cycles before it halts and rejects, that is, before it detects its error.*

Obviously, the property of being correctness preserving corresponds to error-detection distance 0. Our first result now states that restarting automata with bounded error-detection distance are algorithmically well-behaved. Here we will use the notation $|M|$ to denote the *size* of a restarting automaton M, that is, the length of a description of M.

Theorem 2.9.2. *For each $i \geq 0$, the following uniform membership problem is solvable in time $O(|M|^{i+1} \cdot n^{i+2})$:*

INSTANCE : *An* RLWW*-automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ that has error detection distance i, and a string $w \in \Gamma^*$ of length n.*

QUESTION : *Is w accepted by M, that is, does $w \in L_C(M)$ hold?*

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be the given RLWW-automaton that has error detection distance i, and let $w \in \Gamma^*$ be the given input word of length n. In order to decide whether or not $w \in L_C(M)$ holds, we construct a tree $T = T(M, w, i)$ inductively as follows from the RLWW-automaton M, the string w, and the constant i.

The root of T is labelled by w. If M accepts w in a tail computation, then $w \in L_C(M)$, and we accept. Otherwise, if no meta-instruction of M is applicable to w, then M rejects w, and so do we. Finally, we create a new node with label w_1 for each string $w_1 \in \Gamma^*$ for which M can execute a

cycle of the form $w \vdash_M^c w_1$, and we introduce directed edges from the root to each of these nodes. Each of these nodes can be constructed in time $O(|M| \cdot n)$. As $|w| = n$, and as for each factor of length k of w, the number of applicable rewrite transitions is bounded by a constant that depends on M, we see that there are at most $O(|M| \cdot n)$ many such strings w_1. Thus, this part of the tree is constructed in time $O(|M|^2 \cdot n^2)$.

Next, if there is an accepting tail computation for any of the strings w_1, then $w_1 \in L_C(M)$, and therewith $w \in L_C(M)$. Hence, we terminate the process and accept. On the other hand, if no meta-instruction is applicable to any of the strings w_1, then M rejects all these strings, and so it rejects w, that is, we terminate the process and reject as well. Finally, for each string w_1, we create a new node with label w_2 for each string such that M can execute the cycle $w_1 \vdash_M^c w_2$, and add a directed edge from the node with label w_1 to all these new nodes. As there are at most $O(|M| \cdot n)$ successor nodes for each node at level 1, we obtain at most $O(|M|^2 \cdot n^2)$ nodes at level 2. Hence, this part takes time at most $O(|M|^3 \cdot n^3)$.

This process of creating nodes and edges is repeated until either we accept, or we reject, or until we have created all possible nodes up to level i. Thus, we have obtained at most $\sum_{j=0}^{i} O(|M|^j \cdot n^j) = O(|M|^i \cdot n^i)$ many nodes, and this construction takes time $O(|M|^{i+1} \cdot n^{i+1})$.

Finally, if M accepts any string occurring as a label in the tree T in a tail computation, then $w \in L_C(M)$, and we accept. If no meta-instruction of M is applicable to any string w_i that occurs as a label of a node at level i, then M rejects all these strings, and so it rejects w, that is, $w \notin L_C(M)$. If, for some string w_i labelling a node at level i, M can execute a cycle of the form $w_i \vdash_M^c v$, then we replace w by the string w_1 that is the immediate descendant of the root on the path p from the root to the node with label w_i. Indeed, with the path p we have constructed a sequence of cycles of the form $w \vdash_M^c w_1 \vdash_M^c w_2 \vdash_M^c \cdots \vdash_M^c w_i \vdash_M^c v$, and as M has error detection distance i, this shows that $w_1 \in L_C(M)$ if and only if $w \in L_C(M)$. Thus, we can now repeat the above construction with the string w_1. As $|w_1| < |w|$, we see that this overall process will terminate after at most n rounds. Hence, in this way we can determine whether or not $w \in L_C(M)$ holds. The whole process takes time at most $O(|M|^{i+1} \cdot n^{i+2})$. □

2.9.1 Bounded Error-Detection Distance

Here it is shown that, for each type X of RLWW-automaton, an X-automaton of bounded error-detection distance is equivalent in expressive power to an X-automaton that is correctness preserving. We first establish this result for R-, RW-, and RWW-automata.

Theorem 2.9.3. *For any* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$*, if* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ *is an* RX*-automaton that has error-detection distance* $i \geq 1$*, then there exists an* RX*-automaton* $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k', \delta')$ *with error-detection distance* 0 *such that* $L_C(M') = L_C(M)$ *and* $L(M') = L(M)$*.*

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be an RX-automaton that has error-detection distance $i = 1$. We will construct a correctness preserving RX-automaton $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k', \delta')$ satisfying $L_C(M') = L_C(M)$.

In general a rewrite step or a cycle $w_1uw_2 \vdash_{M'}^c w_1vw_2$ of M' is correctness preserving, if M can perform this cycle as well and if M can execute another cycle starting from w_1vw_2, because M has error detection distance 1. A correctness preserving rewrite step is called *safe*, if M' knows that it is correctness preserving.

Let $q_0¢w\$$ be a restarting configuration of M. Assume that $w = w_1uw_2$ for some rewrite step $u \to v$ of M, and assume that this is the leftmost applicable rewrite step. Thus, by applying this rewrite step, M can execute the cycle $w = w_1uw_2 \vdash_M^c w_1vw_2$. If w_1vw_2 contains the lefthand

side of a rewrite operation $u' \to v'$ such that u' overlaps with the prefix w_1v of w_1vw_2, then $w_1vw_2 = w_1'u'w_2' \vdash_M^c w_1'v'w_2'$. As by hypothesis M has error-detection distance 1, we see that $w_1uw_2 \in L_C(M)$ holds if and only if $w_1vw_2 \in L_C(M)$ holds. Thus, in this situation the above application of the rewrite step $u \to v$ is *safe*.

If no such safe rewrite is found, then no rewrite is done and the head moves on, looking for another possible rewrite on w_1uw_2 and on w_1vw_2. If a second rewrite for w_1uw_2 is found, then it is safe, because M can perform the leftmost rewrite afterwards. A rewrite $u_1 \to v_1$ on $x_1 = w_1vw_2 = w_1vw_{2,1}u_1w_{2,2}$ is not safe, we only know that the first rewrite step would have been safe, which does not help because M has no MVL-operations to reach the place of this rewrite step again. We have to move on, scanning the tape content w_1uw_2, x_1 and $x_2 = w_1vw_{2,1}v_1w_{2,2}$, until we find a rewrite which is the second possible rewrite for one of the x_i.

Now M' can simulate M as follows. Starting from the restarting configuration $q_0¢w_1uw_2\$$, M' detects the factor u. While doing this it determines whether there exists a rewrite operation of M for which the lefthand side overlaps with the prefix w_1v of w_1vw_2. If such an overlap exists, then M' simply applies the rewrite step $u \to v$ and restarts. If no such overlap exists, then M' moves on, scanning w_2. At the same time it simulates the behavior of M on the tape content $x_1 = w_1vw_2$ in its finite-state control. In this way M' tries to detect a safe rewrite step that it can apply to $¢w_1uw_2\$$.

In detail:

Each of the meta-instructions $I_i = (E_i, u_i \to v_i)$ of the automaton M is simulated simultaneously, by a finite automaton F_i in the finite control of M'. Everytime M can perform a rewrite step $u \to v$, then each of the F_i simulates the tape content w_1uw_2 and w_1vw_2. After the first possible rewrite each F_i has two marked states, one with the marker "old" for the tape content w_1uw_2 and one with the marker "new" for the tape content w_1vw_2. When there is a possible rewrite for w_1vw_2, then the old marker "new" is changed to "old" and a new marker "new" is created. Thus there are two markers "old" and one marker "new". For every possible rewrite with the marker "new", there is a new marker "new" created and the old one is renamed to "old", such that always one marker "new" and some markers "old" exist. If there is a possible rewrite for an "old" marker, then this rewrite is applied, and if the right border marker is reached without the possibility to perform a safe rewrite, M' rejects. Whenever an accepting meta-instruction is applicable, M' accepts, no matter whether it is an "old or "new" marker.

Obviously, M' is an RX-automaton. There are two things left to prove: the new automaton M' is correctness preserving and $L_C(M') = L_C(M)$ holds.

We will show that all rewrites of M' are safe. M' performs a rewrite if it sees another rewrite which overlaps with the current one, therefore this rewrite is safe.

The only other time M' performs a rewrite is, when there is a possible rewrite with the marker "old". This means that there is a tape content w_k and there are two different rewrites applicable to this tape content. This means that, after this cycle, M is able to perform another cycle with a rewrite which is more to the left. Thus this rewrite is safe. And there is a sequence of cycles of M, which leads to these two possible rewrites.

Claim: $L_C(M') = L_C(M)$

$L_C(M') \subseteq L_C(M)$: As M' has the same accepting meta-instructions as M and as all rewrite steps that M' can execute are also valid rewrite steps of M, it follows that each accepting computation of M' is also a valid accepting computation of M.

$L_C(M') \supseteq L_C(M)$: If a word w belongs to $L_C(M)$, then there exists an accepting sequence of cycles starting from $¢w\$$. Assume that w is not accepted by M' and that there is no cycle of M'

applicable to w. If there exists a word $w \in L_C(M) \setminus L_C(M')$, then for each word v with $w \vdash_{M'}^{c^*} v$ it follows that $v \in L_C(M) \setminus L_C(M')$. $v \in L_C(M)$ holds because M' is correctness preserving and $L_C(M') \subseteq L_C(M)$ holds, and $v \notin L_C(M')$ holds because of the error preserving property of M'. As each computation of M' starting with w is finite, it follows that there exist words such that no meta-instruction of M' is applicable to them.

Assume $w \in L_C(M) \setminus L_C(M')$ and there is no cycle of M' applicable to w. This is the case when M' scans the tape completely from left to right and finds no safe rewrite step. Then M can perform only one sequence of cycles on w and this sequence is strictly monotone (the right distance decreases in each cycle by more than the windowsize of M), because otherwise there would be a safe rewrite for M'. The sequence ends with a reject, because otherwise M' would accept w, therefore $w \notin L_C(M)$. This contradicts the assumption $w \in L_C(M) \setminus L_C(M')$.

If M has error-detection distance $i > 1$, then the technique described above can be used to combine, in each computation of M, the last cycle with the following tail computation. In this way the error-detection distance is reduced by 1. Thus, if M is an RX-automaton with error-detection distance $i > 1$, then we obtain an RX-automaton M' with error-detection distance $i - 1$ such that $L_C(M') = L_C(M)$ and $L(M') = L(M)$. □

A corresponding result also holds for RRX- and RLX-automata.

Theorem 2.9.4. *For any* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$*, if* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ *is an* RRX- *or* RLX-*automaton that has error-detection distance* $i \geq 1$*, then there exists an* RRX- *or* RLX-*automaton* $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k, \delta')$ *with error-detection distance at most* $i-1$ *such that* $L_C(M') = L_C(M)$ *and* $L(M') = L(M)$*.*

Proof. The idea is similar to the one used in the proof of the preceding result. However, for RRX- and RLX-automata the situation is slightly less involved, as such an automaton, after executing a rewrite operation, can completely scan the resulting tape content before performing a restart operation. □

By applying Theorem 2.9.3 or Theorem 2.9.4 repeatedly we obtain the following consequence.

Corollary 2.9.5. *For any* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, if* M *is an* X*-automaton that has bounded error-detection distance, then there exists a correctness preserving* X*-automaton* M' *such that* $L_C(M') = L_C(M)$ *and* $L(M') = L(M)$*.*

Thus, for all these types of restarting automata bounded error-detection distance limits the expressive power to that of correctness preserving automata.

In combination with Theorem 2.9.2 this means that the membership problem for the language $L_C(M)$ is decidable in quadratic time, if M is an RLWW-automaton with bounded error-detection distance. Observe, however that the resulting uniform algorithm includes the transformation of a given RLWW-automaton of bounded error-detection distance into an equivalent RLWW-automaton that is correctness preserving.

By combining Theorems 2.8.3 and 2.8.5 with Corollary 2.9.5 we obtain the following result.

Corollary 2.9.6. (a) *For any* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$*, if* M *is an* RX*-automaton that has bounded error-detection distance, then there exists a deterministic* RX*-automaton* M' *satisfying* $L(M) = L(M')$ *and* $L_C(M) = L_C(M')$*.*

(b) *For any* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$*, if* M *is an* RRX- *or an* RLX*-automaton that has bounded error-detection distance, then there exists a deterministic* RLX*-automaton* M' *satisfying* $L(M') = L(M)$ *and* $L_C(M') = L_C(M)$*.*

With the possible exception of RLWW-automata, it is well-known that, for all types X of restarting automata, nondeterministic X-automata are strictly more powerful than deterministic X-automata. However, Corollary 2.9.6 shows that for the various types of R- and RL-automata, the deterministic variant is as expressive as the nondeterministic variant with bounded error-detection distance. This clearly shows that for these types of restarting automata, the nondeterministic variants are more expressive than the corresponding deterministic variants only if they have unbounded error-detection distance. Thus, it is the ability of a nondeterministic RWW- or RLWW-automaton to make an error that is only detected an unbounded number of cycles later that really contributes to the expressive power of these automata. On the negative side this means that the error-detection distance is not a useful complexity measure for restarting automata, as there are essentially only two cases: error-detection distance 0 (that is, strongly correctness preserving automata) and unbounded error-detection distance.

Chapter 3

Nonforgetting Restarting Automata

In this chapter we consider restarting automata with more than one restarting state, the so-called nonforgetting restarting automata. These automata are able to restart in different restarting states.

This generalization of restarting automata is linguistically and theoretically motivated. The linguistic motivation is that not all reduction steps are independent from each other. Sometimes more than one step needs to be done, to maintain the correctness or incorrectness of a given sentence. To solve this problem, we need to have many rewrites per cycle or perform them sequentially and remember the information in the restarting states.

The theoretical or technical motivation is the following: a restarting automaton has two restrictions that come with its restart operation:

- It cannot remember what it has done in the previous cycle.
- It cannot remember where the last rewrite or restart was performed on the tape.

So we can consider restarting automata with only one of these restrictions. Removing the first restriction leads to nonforgetting restarting automata. Restarting automata without the second restriction are not considered here. The "normal" restarting automata are often called forgetting restarting automata in this chapter to distinguish them from the nonforgetting ones.

3.1 Definitions and Examples

Nonforgetting restarting automata were first introduced in a joint paper by the author and Heiko Stamer [MS04]. Some of the results on nonforgetting restarting automata that are deterministic, monotone or deterministic monotone have been announced in [MO06].

Definition 3.1.1. *A nonforgetting restarting automaton,* nf-RLWW*-automaton for short, is a nine tuple* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta, Q_R)$*, where*

- Q *is a finite set of states,*
- Σ *is a finite input alphabet,*
- $\Gamma \supseteq \Sigma$ *is a finite tape alphabet,*

- $¢, \$ \notin \Gamma$ *are border markers,*
- $q_0 \in Q$ *is the initial state*
- $k \geq 1$ *is the window size*
- $\delta : Q \times \mathcal{P}C^{(k)} \to \mathfrak{P}((Q \times (\{\mathsf{MVR}, \mathsf{MVL}\} \cup \mathcal{P}C^{\leq(k-1)})) \cup \{\mathsf{Restart} \times Q_R\} \cup \{\mathsf{Accept}\})$ *is the transition relation and*
- $Q_R \subset Q$ *is the set of restarting states that contains* q_0*.*

The only formal difference between forgetting and nonforgetting restarting automata is the fact that nonforgetting restarting automata are able to restart not only in their initial state, but in different restarting states. Nonforgetting restarting automata can be described by the eight tuple $(Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, if the number of restarting states that are needed to accept a language is not taken into account. In this case $Q_R = Q$ holds and Q_R is not given explicitly.

The restrictions on the movement and the rewrite step can be applied to nonforgetting restarting automata without changes (see Section 2.1).

A configuration of a nonforgetting restarting automaton is described by $\alpha q \beta$ where $q \in Q$ is the current state and $\alpha\beta$ is the current tape content including the border markers, with the read/write window on the first k letters of β. A configuration $q¢w\$$ with $q \in Q_R$ is called a restarting configuration; it is called an initial configuration only if $q = q_0$ and $w \in \Sigma^*$ hold. Thus, a nonforgetting restarting automaton can distinguish between initial and certain restarting configurations even if no auxiliary symbols are present.

Definition 3.1.2. *The language accepted by a nonforgetting restarting automaton* M*,* $L(M)$*, consists of all words* $w \in \Sigma^*$ *for which there is an accepting computation of* M *starting from the initial configuration* $q_0¢w\$$*. The complete language of* M*,* $L_C(M)$ *for short, consists of all words* $w \in \Gamma^*$ *for which there exists an accepting computation starting from the restarting configuration* $q_0¢w\$$*.*

The meta-instructions for nf-RRWW-automata, which again can shorten the description of the transition relation, differ from the meta-instructions of forgetting restarting automata. A meta-instruction describes a cycle from one restarting configuration to the next. As a restarting configuration of a nonforgetting restarting automata contains a state $q \in Q_R$, which need not be the initial state, meta-instructions are defined as follows.

Definition 3.1.3. *A* meta-instruction *of a* nf-RRWW*-automaton* M *is either of the form* $(q, E_1, u \to v, E_2, p)$ *or* $(q, E_1, \mathsf{ACCEPT})$*, where* E_1, E_2 *are regular expressions and* $u, v \in \Gamma^*$ *are words with* $|u| > |v|$*.* $q \in Q_R$ *is the current restarting state and* $p \in Q_R$ *is the restarting state for the next cycle of the computation.*

To perform a cycle, M *chooses a meta-instruction of the form* $(q, E_1, u \to v, E_2, p)$*. On trying to execute this meta-instruction,* M *will get stuck (and so reject) starting from the configuration* $q_1¢w\$$*, if* $q \neq q_1$ *or if* w *does not admit a factorization of the form* $w = w_1uw_2$ *such that* $¢w_1 \in E_1$ *and* $w_2\$ \in E_2$*. On the other hand, if* $q = q_1$ *and* w *does have a factorization of this form, then one such factorization is chosen nondeterministically, and* $q¢w\$$ *is transformed into* $p¢w_1vw_2\$$*. In order to describe the tails of accepting computations we use meta-instructions of the form* $(q, E_1, \mathsf{ACCEPT})$*, where the strings from the regular language* E_1 *are accepted by* M *in tail computations when starting in state* q*.*

Rewriting meta-instructions for nf-RWW*-automata are of the form* $(q, E_1, u \to v, p)$*. Rewriting restrictions apply only to* v*. For a* nf-RRW*-automaton the word* v *must not contain auxiliary*

symbols, this implies that u, E_1 and E_2 do not contain auxiliary symbols either. For a nf-RR-*automaton, v must be a scattered subword of u, which again implies that u, v, E_1 and E_2 do not contain auxiliary symbols.*

Accepting meta-instructions are equal for all types of nonforgetting restarting automata.

A cycle of M from one restarting configuration $p\¢w_0\$$ to another restarting configuration $q\¢w_1\$$ is written as $p\¢w_0\$ \vdash_M^c q\¢w_1\$$ or as a relation over pairs consisting of restarting states and words: $(p, w_0) \vdash_M^c (q, w_1)$. An accepting tail is described by $p\¢w_0\$ \vdash_M^c$ ACCEPT or by $(p, w_0) \vdash_M^c$ ACCEPT.

Some results on forgetting restarting automata carry over to nonforgetting ones. First of all nonforgetting restarting automata do not need MVL-operations, if they are nondeterministic.

Theorem 3.1.4. [Plá01] *Let $M_L = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_L, Q_R)$ be a* nf-RLWW-*automaton. Then there exists a* nf-RRWW-*automaton $M_R = (Q', \Sigma, \Gamma, ¢, \$, q_0, k, \delta_R, Q_R)$ where $Q_R \subseteq Q'$ such that, for all $u, v \in \Gamma^*$ and $q_1, q_2 \in Q_R$,*

$$q_1\¢u\$ \vdash_{M_L}^c q_2\¢v\$ \quad \text{if and only if} \quad q_1\¢u\$ \vdash_{M_R}^c q_2\¢v\$,$$

and the languages $L(M_L)$ and $L(M_R)$ coincide.

Proof. The proof is the same as for forgetting restarting automata [Plá01]. We use two sets of crossing sequences to simulate the two two-way-finite automata before and after the rewrite by one-way-finite automata [HU79]. The only difference is that we have to consider not only the starting state q_0 at the left border of the tape, but all restarting states. So we get different sets of crossing sequences, if the automaton starts in different restarting states. □

A nonforgetting restarting automaton can have different states that it may be in after a restart step. With this, it is possible to remember what has been done in the last steps. The largest difference to forgetting restarting automata is that a nonforgetting restarting automaton can remember whether it is in an initial or in a restarting configuration. Because of this the "normal" error preserving and correctness preserving properties of restarting automata are not of much help for nonforgetting restarting automata. But first we look at a short example to illustrate the expressive power of this type of automaton.

Example 3.1.5 (Exponential Languages)**.**

The language $L_{\text{expo}} = \{a^{2^n} \mid n \in \mathbb{N}\}$ is recognizable by the det-nf-RWW-*automaton with one auxiliary symbol B with the following meta-instructions:*

$(q_0, ¢a^*, aaaa\$ \to Baa\$,\ q_1)$ $(q_1, ¢a^*, aaB \to Ba,\ q_1)$
$(q_1, ¢, B \to \varepsilon,\ q_0)$ $(q_0, ¢aa\$,\ \text{ACCEPT})$
$(q_0, ¢a\$,\ \text{ACCEPT})$

In the initial state M scans the tape completely from left to right. If it sees any symbols different from a it halts and rejects. Otherwise it creates a B at the right border and restarts in q_1. This B is used to halve the number of a's while it moves across them from right to left. When the B reaches the left border marker it is deleted and the initial state is reentered. This halving is repeated until M encounters an error or until it accepts the string a or aa.

The language $L'_{\text{expo}} = \{a^{2^n} b \mid n \in \mathbb{N}\}$ is recognizable by the det-nf-RW-*automaton with the following meta-instructions:*

$(q_0, ¢a^+, aab\$ \to ba\$,\ q_1)$ $(q_1, ¢a^*, aab \to ba,\ q_1)$

$(q_1, ¢b \to \varepsilon,\ q_1)$ $(q_1, ¢a^*, aaaa\$ \to baa\$,\ q_1)$

$(q_1, ¢aa\$,\ \mathsf{ACCEPT})$ $(q_0, ¢ab\$,\ \mathsf{ACCEPT})$

$(q_0, ¢aab\$,\ \mathsf{ACCEPT})$

Here the b is used instead of the auxiliary symbol B from the previous automaton. After the first cycle it follows for each configuration $q¢w\$$ in an accepting computation that $w \notin L'_{\mathsf{expo}}$ holds. This is the reason that the initial state is never reached again after the first cycle.

The language L'_{expo} can be encoded using the encoding from Theorem 2.2.13, or, in this case with the easier encoding $\phi(a) = ab$ and $\phi(b) = b$. This encoded version of L'_{expo} can be accepted by a deterministic nf-R-automaton. Thus already a nf-R-automaton can recognize an encoding of the exponential language. Remark that all these automata are deterministic.

3.2 Properties and Lemmata

Now we come to the error and the correctness preserving properties. Nonforgetting restarting automata fulfill the old Error and Correctness Preserving properties, but these do no longer yield effective proof methods, because a nonforgetting restarting automaton does not need to reach its initial state after every restart (In fact it does not have to reach it ever again).

Lemma 3.2.1 (Error Preserving Property 1)**.**

Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta, Q_R)$ be a nf-RLWW*-automaton, and let u, v be words over its* input *alphabet. If $q_0¢u\$ \vdash_M^{c^*} q_0¢v\$$ holds and $u \notin L(M)$, then $v \notin L(M)$, either.*

So we have to find new properties, which are more useful. First we define different types of configurations for nonforgetting restarting automata.

Definition 3.2.2.

A restarting configuration $q¢w\$$ is called an accepting configuration*, if $q¢w\$ \vdash^c \mathsf{ACCEPT}$ holds.*

A restarting configuration $q¢w\$$ is called a promising configuration*, if there exists a computation $q¢w\$ \vdash^{c^*} p¢w'\$$ and $p¢w'\$$ is an accepting configuration.*

The set of all promising configurations of M is called $P_C(M)$.

For each initial promising configuration $q_0¢w\$$, it follows that there exists an accepting computation for $w \in \Sigma^*$, that is, w is accepted by M and belongs to $L(M)$. For each promising restarting configuration $q¢w\$$ with $q = q_0$, it follows that w belongs to the complete language of M ($w \in L_C(M)$). Restarting configurations can be described either by $q¢w\$$ or by the pair (q, w). With this definition we can define an extension of the error preserving property.

Lemma 3.2.3 (Error Preserving Property 2)**.**

Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta, Q_R)$ be a nf-RLWW*-automaton, let u, v be words over its tape alphabet, and let $p, q \in Q_R$. If $q¢u\$ \vdash_M^{c^*} p¢v\$$ holds and $q¢u\$$ is not a promising configuration, then $p¢v\$$ is neither.*

Lemma 3.2.4 (Correctness Preserving Property)**.**

Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta, Q_R)$ be a nf-RLWW*-automaton, let u be a word over its tape alphabet, and let $q \in Q_R$. If $q¢u\$$ is a promising configuration, then $q¢u\$$ is an accepting configuration or there exists a word v and a restarting state p, such that $q¢u\$ \vdash_M^c p¢v\$$ holds and $p¢v\$$ is also a promising configuration.*

The strong Correctness Preservation is defined as follows.

Definition 3.2.5. *Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta, Q_R)$ be a* nf-RLWW*-automaton. M is called (strongly) correctness preserving, if every successor restarting configuration of a promising configuration is again a promising configuration.*

These two lemmata allow us to use the error and correctness preserving properties for nonforgetting restarting automata. Deterministic nonforgetting restarting automata are always strongly correctness preserving.

For forgetting restarting automata the new and the old version of these lemmata coincide.

As the pumping lemma for restarting automata describes only the behavior in one cycle or in a tail, it is also valid for nonforgetting restarting automata with the only change that the restarting states in the cycle can be different from the initial state.

Proposition 3.2.6 (Pumping Lemma)**.**

For any nf-RLWW*-automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, there exists a constant p such that the following holds: Assume that $q_1¢uvw\$ \vdash_M^c q_2¢uv'w\$$, where $u = u_1u_2u_3$ and $|u_2| = p$. Then there exists a factorization $u_2 = z_1z_2z_3$ such that z_2 is non-empty, and*

$$q_0¢u_1z_1(z_2)^iz_3u_3vw\$ \vdash_M^c q_0¢u_1z_1(z_2)^iz_3u_3v'w\$$$

holds for all $i \geq 0$, that is, z_2 is a 'pumping factor' in the above cycle. Similarly, such a pumping factor can be found in any factor of length p of w. Such a pumping factor can also be found in any factor of length p of a word accepted in a tail computation.

3.3 Deterministic Restarting Automata

Deterministic RWW- and RRWW-automata accept the *Church-Rosser languages* (CRL) (see Definition 2.4.2 on page 22), which are in some sense the deterministic variants of the *growing context-sensitive languages* (GCSL). While the language $L_{\mathsf{copy}} := \{ w\#w \mid w \in \{a, b\}^* \}$ is not even growing context-sensitive [BO98], it is accepted by the det-nf-R-automaton that is given in the following example.

Example 3.3.1. *The language L_{copy} is accepted by a* det-nf-R*-automaton M with two restarting states.*

(i)	$(q_0, ¢a \cdot \{a, b\}^* \cdot \#, a \to \varepsilon, q_1)$,	(iv)	$(q_1, ¢, a \to \varepsilon, q_0)$,
(ii)	$(q_0, ¢b \cdot \{a, b\}^* \cdot \#, b \to \varepsilon, q_1)$,	(v)	$(q_1, ¢, b \to \varepsilon, q_0)$.
(iii)	$(q_0, ¢\#\$, \mathsf{Accept})$,		

M remembers the first symbol on the tape in its finite control and moves to the right until it sees the separator #. If the first symbol after the separator coincides with the first symbol on the tape, then M deletes it and restarts in state q_1, otherwise it halts and rejects ((i) and (ii)) In state q_1 M simply deletes the first symbol on the tape and restarts in the initial state ((iv) and (v)). M

knows that if it is in the restarting state q_1, then it was already checked that the first symbols from the first and the second part of the word coincide. The meta-instructions (i) and (vi) or (ii) and (v) are used alternatingly until M accepts in the initial state while seeing the tape content ¢#$.

Thus, we see that deterministic nonforgetting restarting automata are more powerful than their forgetting counterparts. The question is how powerful are they exactly?

It is still open whether $\mathcal{L}(\mathsf{det\text{-}nf\text{-}RLWW}) = \mathcal{L}(\mathsf{nf\text{-}RLWW})$ holds or not, but it is a conjecture that these classes do not coincide.

The next lemma gives a normalform for deterministic (nonforgetting) restarting automata

Lemma 3.3.2. *For every (nonforgetting)* det-RL- *(*det-RLW-*,* det-RLWW-*) automaton M, there exist a (nonforgetting)* det-RL-*(*det-RLW-*,* det-RLWW-*) automaton M' that performs its restart operations immediately after its rewrite operations, such that $L(M) = L(M')$ holds. If M is a forgetting restarting automaton, then so is M'.*

Proof. In the following proof let $\mathsf{X} \in \{\varepsilon, \mathsf{W}, \mathsf{WW}\}$. In each cycle

$$(q, w_1uw_2) \vdash_M^c (p, w_1vw_2),$$

a det-nf-RLX-automaton M moves over the tape performing MVR- and MVL-steps until it performs one Rewrite-step. After this it can scan the tape again, but we will show that it is sufficient to scan the tape prior to the rewrite step. For this we construct a $\mathsf{det-nfRLX}$-automaton M' with $L(M) = L(M')$ and M' restarts immediately after the rewrite step.

A det-nf-RLX-automaton is correctness preserving, therefore with Theorem 3.8.3 there exists a correctness preserving nf-RRX-automaton $\tilde{M}$ that accepts the same language and M performs the same cycles as $\tilde{M}$. Thus after M has found the place of the rewrite deterministically it can again move over the tape, but as $\tilde{M}$ can perform the same cycle there exists a meta-instruction $(q, E_1, u \to v, E_2, p)$ for that cycle that only checks regular conditions for w_1 and w_2. The only problem is that $\tilde{M}$ does not need to find the place of the rewrite step deterministically.

The det-nf-RLX-automaton M' works as follows. It first scans the tape completely from left to right, looking for accepting meta-instructions of $\tilde{M}$. If it detects that $\tilde{M}$ would accept, it accepts as well. Otherwise it simulates M until M would perform a rewrite step $u \to v$. M' does not perform this Rewrite-step, instead it moves to the left border marker and determines all meta-instructions of $\tilde{M}$ such that w_1 fulfills the prefix-condition and the rewrite step is $u \to v$. Then it simulates M a second time to find the place for the rewrite again and checks whether w_2 fulfills the suffix-condition of one of these meta-instructions. If at least one meta-instruction $(q, E_1, u \to v, E_2, p)$ of $\tilde{M}$ is applicable, that is $w_1 \in E_1$ and $w_2 \in E_2$ holds, then $\tilde{M}$ can performs the cycle $(q, w_1uw_2) \vdash_{\tilde{M}}^c (p, w_1vw_2)$. Thus also M can perform this cycle and M' therefore simulates M a third time to find the place for the rewrite step again, performs it and restarts in state p. If no meta-instruction fulfills these conditions, then $\tilde{M}$ cannot perform a cycle with this particular rewrite step and therefore M does not perform a cycle with this rewrite step. Therefore M rejects after the rewrite step and M' can reject without performing this rewrite step.

Thus M' simulates M cycle by cycle, but it restarts immediately after the rewrite step. As M' simulates M cycle by cycle, it follows that $L(M) = L(M')$. □

Now we state a Theorem from [MS04], which was already mentioned in a weaker form in the previous chapter. Also the new result about RRWW-automata that are restricted to perform their rewrites at the right border marker carries over to nf-RRWW-automata.

Lemma 3.3.3. *Let M be a* nf-RRWW*-automaton such that M performs all of its rewrites while seeing the right border marker \$ in its window. Then $L(M)$ is regular.*

Proof. As the automaton performs all of its rewrites while seeing the right border marker \$, each nf-RRWW-automaton is in fact an nf-RWW-automaton.

We will construct an NFA A for $L(M)$. If $x_0 \in L(M)$, then there exists an accepting computation starting from $q_0 ¢ x_0 \$$, which can be described by a sequence of cycles:

$$(q_0, x_0) \vdash_m^c (q_{j_1}, x_1) \vdash_m^c (q_{j_2}, x_2) \ldots \vdash_m^c (q_{j_n}, x_n) \vdash_m^c \mathsf{ACCEPT}$$

As the rewrites of M are performed only at the right end of the tape, there exists a factorization of $x_0 = x u_n u_{n-1} \ldots u_2 u_1$, such that $x_i = x u_n u_{n-1} \ldots u_{i+1} v_i$, $0 < i < n$, and $x_n = x v_n$ hold. Here $u_i v_{i-1} \$ \to v_i \$$ is the ith rewrite step. It is required that $v_0 = \varepsilon$ holds.

A simulates the regular conditions of all meta-instructions simultaneously in its finite control, remembering, in which restarting state they started. As M performs all its rewrite steps at the right border, the input x_0 is divided into two parts: a prefix x which is not changed during the sequence of cycles, and a suffix $u_n u_{n-1} \ldots u_2 u_1$ which is rewritten step-by-step into v_n during this sequence.

The sequence of cycles is simulated from right to left as follows. First A guesses an accepting meta-instruction $(q_{j_n}, ¢ x v_n \$)$. Then it checks for the prefix $¢x$ of it and remembers the missing suffix $v_n \$$. This suffix needs to be a right-hand side of a rewrite step $u_n v_{n-1} \$ \to v_n \$$ of a meta-instruction $(q_{j_{n-1}}, E_n, u_n v_{n-1} \$ \to v_n \$, q_{j_n})$ such that $¢x \in E_n$ holds. That is, the meta-instruction is applicable to $q_{j_{n-1}} ¢ x u_n v_{n-1} \$$.

Then A looks for the prefix u_n of this rewrite instruction. The remaining suffix $v_{n-1} \$$ again has to be the right-hand side of the rewrite step of a meta-instruction $(q_{j_{n-2}}, E_{n-1}, u_{n-1} v_{n-2} \$ \to v_{n-1} \$, q_{j_{n-1}})$, such that $¢ x u_n \in E_{n-1}$ holds. This is repeated until the whole input is consumed and A has found a meta-instruction that is applicable to $q_0 ¢ x_0 \$ = q_0 ¢ x u_n u_{n-1} \ldots u_2 u_1 \$$, transferring it to $q_{j_1} ¢ x_1 \$ = q_{j_1} ¢ x u_n u_{n-1} \ldots u_2 v_1 \$$.

Thus A accepts the input x_0 if and only if it finds an accepting sequence of cycles of M. □

Theorem 3.3.4. [MS04] *A language over an unary alphabet is regular if and only if it is accepted by a* nf-RLW*-automaton.*

Proof. Each regular language can be accepted by a nf-RLW-automaton in one cycle, so $\mathsf{REG} \subseteq \mathcal{L}(\text{nf-RLW})$ holds.

The other direction is a bit more complicated. A nf-RLW-automaton can be simulated by a nf-RRW-automaton (Theorem 3.1.4). A restarting automaton without auxiliary symbols can only delete some symbols when the tape alphabet is unary. Thus over a unary alphabet nf-RRW-automata are in fact only nf-RR automata. Next it does not matter where the rewrite occurs. Each meta-instruction of a nf-RR-automaton of the form $(q, ¢E_1, a^l \to \varepsilon, E_2 \$, p)$, where $\Sigma = \{a\}$ and $l \leq k$, checks the same conditions as the meta-instruction $(q, ¢E_1 \cdot E_2, a^l \$ \to \$, p)$ of a nf-R-automaton. Thus over a unary alphabet each nf-RR-automaton can be simulated by a nf-R-automaton. The nf-R-automaton only needs a window of size $k+1$ to see the right border marker and delete up to k symbols.

Now the assertion follows from Lemma 3.3.3. □

With this result and the deterministic nonforgetting RWW-automaton from Example 3.1.5, we see that the language L_{expo} separates (deterministic) nonforgetting restarting automata with auxiliary symbols from those without auxiliary symbols.

Corollary 3.3.5.

$$L_{\mathsf{expo}} \in \mathcal{L}(\mathsf{det\text{-}nf\text{-}RWW}) \setminus \mathcal{L}(\mathsf{nf\text{-}RRW}),$$
$$L'_{\mathsf{expo}} \in \mathcal{L}(\mathsf{det\text{-}nf\text{-}RW}) \setminus \mathcal{L}(\mathsf{nf\text{-}RR}).$$

Proof. From Example 3.1.5 we know that the exponential languages belong to the stated language classes. As L_{expo} is not regular, it follows from Theorem 3.3.4 that it does not belong to $\mathcal{L}(\mathsf{nf\text{-}RRW})$.

It remains to show that $L'_{\mathsf{expo}} \notin \mathcal{L}(\mathsf{nf\text{-}RR})$ holds.

L_{expo} is a unary non-regular language and L'_{expo} differs from L_{expo} only by the occurrence of an additional b at the end of the tape. For a nf-RR-automaton the b at the end cannot be used. Therefore, if there exists a nf-RR-automaton that accepts L'_{expo}, then it follows, that there exists also a nf-RR-automaton that accepts L_{expo}. This contradicts Theorem 3.3.4. Thus $L'_{\mathsf{expo}} \notin \mathcal{L}(\mathsf{nf\text{-}RR})$ holds. □

Thus we have the situation depicted in Figure 3.1, where $\mathcal{L}_1 \to \mathcal{L}_2$ expresses the fact that the language class $\mathcal{L}_1$ is properly contained in the class $\mathcal{L}_2$, while $\mathcal{L}_1 \xrightarrow{?} \mathcal{L}_2$ indicates an inclusion for which it is still open whether it is proper or not. Where no arrow is given the language classes are incomparable with respect to inclusion.

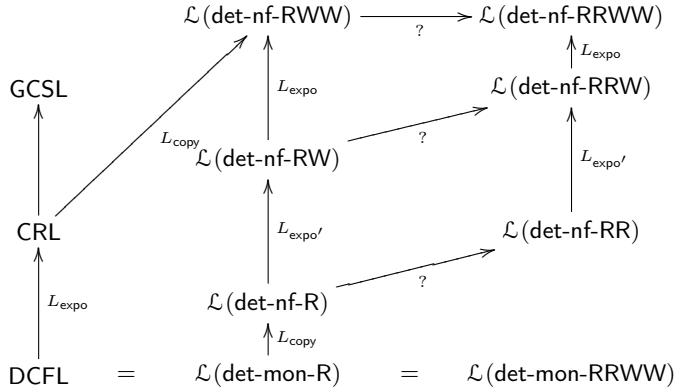

Figure 3.1: The taxonomy of deterministic nonforgetting restarting automata

In addition it is open whether, for any choice of $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$, any of the inclusions $\mathcal{L}(\mathsf{det\text{-}nf\text{-}RX}) \subseteq \mathcal{L}(\mathsf{det\text{-}nf\text{-}RRX}) \subseteq \mathcal{L}(\mathsf{det\text{-}nf\text{-}RLX})$ are proper or not. The problem in showing whether these inclusions are proper or not is that a nonforgetting restarting automaton can scan the tape completely and then remember information in the restarting state. So a det-nf-R-automaton is able to scan the whole tape finitely many times during a computation. It can do this by deleting the last letter and remembering it in the restarting state. It is also open whether the ability to move left and right increases the expressive power of deterministic nonforgetting restarting automata or not.

3.3.1 Forgetting versus Nonforgetting Restarting Automata

Deterministic forgetting restarting automata are much less powerful than nonforgetting ones. As $L_{\mathsf{copy}} \in \mathcal{L}(\mathsf{det\text{-}nf\text{-}R})$, which is not even a growing context-sensitive language, the following corollary can be shown.

Corollary 3.3.6. *Let* $\mathsf{X} \in \{\mathsf{R},\mathsf{RR},\mathsf{RW},\mathsf{RRW},\mathsf{RWW},\mathsf{RRWW}\}$ *then the following proper inclusion holds:* $\mathcal{L}(\mathsf{det\text{-}X}) \subsetneq \mathcal{L}(\mathsf{det\text{-}nf\text{-}X})$

To further illustrate the expressive power of det-nf-RW automata, we now consider the language

$$L_{\mathsf{copy}^*} := \{\, (w\#)^* \mid w \in \{a,b\}^* \,\}.$$

Lemma 3.3.7. $L_{\mathsf{copy}^*} \in \mathcal{L}(\mathsf{det\text{-}nf\text{-}RW})$.

Proof. The det-nf-RW automaton M for the language L_{copy^*} is described by the following meta-instructions, here $c,d,e \in \{a,b\}$ and $x \in \{a,b\}^{\leq 2}$ hold:

$(1)(q_0, ¢(x\#)^*\$, \mathsf{Accept})$,
$(2)(q_0, ¢\{a,b\}^{\geq 3}\#\$, \mathsf{Accept})$
$(3)(q_0, ¢(\{a,b\}^{\geq 3}\#)^+\{a,b\}^{\geq 3}, \#\$ \to \$, q_1)$
$(4)(q_1, ¢\{a,b\}^+, cd\# \to \#\#, q_{cd})$
$(5)(q_{cd}, ¢(\{a,b\}^+\#\#)^+\{a,b\}^+, cd\#e \to \#\#e, q_{cd})$
$(6)(q_{cd}, ¢(\{a,b\}^+\#\#)^+\{a,b\}^+, cd\$ \to \#\$, q_{\#})$
$(7)(q_{\#}, ¢(\{a,b\}^+\#)^*\{a,b\}^+, \#\# \to \#, q_{\#})$
$(8)(q_{\#}, ¢(\{a,b\}^+\#)^*\{a,b\}^+, \#\$ \to \$, q_1)$
$(9)(q_1, ¢(x\#)^+x\$, \mathsf{Accept})$

M accepts a word belonging to L_{copy^*} without performing a single cycle, if $w \leq 2$ or less than two syllables are on the tape . In all other cases M checks in the first cycle whether the input is of the form $w_1\#w_2\#\ldots\#w_m\#$ and each syllable has length at least three. In the affirmative it deletes the last $\#$ and restarts in q_1, in the negative it halts and rejects.

In q_1 the last two symbols of w_1 are replaced by a new copy of the symbol $\#$, thus creating an occurrence of a factor $\#\#$, and stored in the restarting state of M. In the next cycle M compares the stored symbols to the last two symbols of w_2. If they do not agree, then M halts and rejects, otherwise the last two symbols of w_2 are also replaced by the symbol $\#$. This continues until all syllables $w_2,\ldots,w_m$ have been processed. Then M enters the restarting state $q_{\#}$, which causes M to replace each factor of the form $\#\#$ by the symbol $\#$, one at a time, proceeding from left to right. Once this task is completed, every syllable w_i has been shortened by two symbols, which have been verified to agree for all syllables. Now M reenters the restarting state q_1, proceeding to process the now shortened word.

This process continues until either a disagreement between some factors is found, or until all factors have been shortened to length at most 2, and their agreement can be checked simply by scanning the tape from left to right. □

Essentially the same method can be used to design a det-nf-RW automaton that accepts the language VALC(T) of all valid computations of a Turing machine T. This language consists of words of the form $w_0\#w_1\#\ldots\#w_n\#$, where w_0 is an initial configuration of T, w_n is an accepting configuration of T, and w_{i+1} is an immediate successor configuration of the configuration w_i for all $0 \leq i \leq n-1$. Each configuration w_i is of the form $t_0t_1\ldots t_{j-1}qt_jt_{j+1}\ldots t_m$, where $t_0t_1\ldots t_m$ is the support of the tape inscription and q is the current state of T.

Lemma 3.3.8. *VALC*$(T) \in \mathcal{L}(\mathsf{det\text{-}nf\text{-}RW})$.

Proof. To accept VALC(T), a det-nf-RW automaton M first checks that all syllables w_i are configurations of T, that w_0 is an initial configuration and that w_n is an accepting configuration. This is done in the first cycle. Also M checks in this cycle whether, for all $0 \leq i \leq n-1$, w_{i+1} is a possible successor configuration of w_i. This means that, if w_i contains the factor $t_{j-1}qt_jt_{j+1}$, where q is a state of T, and w_{i+1} contains the factor $t_{l-1}pt_lt_{l+1}$, where p is a state of T, then either

- $\delta(q, t_j) = (p, t_{l-1}, R)$ and $t_{j+1} = t_l$, or
- $\delta(q, t_j) = (p, t_l, N)$, $t_{j-1} = t_{l-1}$, and $t_{j+1} = t_{l+1}$, or
- $\delta(q, t_j) = (p, t_{l+1}, L)$ and $t_{j-1} = t_l$

holds, where δ denotes the transition function of T. When M reaches the end of the tape, the last occurrence of the symbol # is removed, and, if all these tests were positive, M restarts in a restarting state q_1 that indicates that the initial check was done. In this first cycle, M cannot possibly check whether the various configurations are consistent with each other, that means that every w_i is a successor of w_{i-1}. In the first cycle this check is done only for the last four letters. For the rest of each syllable it is done in the next phase, where a variant of the method to accept the language L_{copy^*} is used to shorten the tape content. Also it is verified letter by letter that the states occur at the correct places within the various configurations, and that the tape content of each configuration is consistent with the tape content of the next configuration.

In contrast to L_{copy^*}, M remembers the last four letters in its restarting state, because in a valid computation the configurations of a Turing machine are not equal to each other. Near the states that occur in every configuration, the configurations differ.

The consistency check between w_{i-1} and w_i is done as follows:

In the first cycle M verifies that either the last four letters of both syllables are tape symbols of T and they coincide, or that one of the last four letters $a_1a_2a_3a_4$ of w_{i-1} is a state of the Turing machine and one of the following cases holds, with $b_1b_2b_3b_4$ being the last four letters of w_i, t, t' and t_i, $1 \leq i \leq 4$, are tape symbols of T, and p, q are states of T:

- $a_1a_2a_3a_4 = qt_1t_2t_3$ and
  - $\delta(q, t_1) = (p, t, R)$ and $b_1b_2b_3b_4 = tpt_2t_3$, or
  - $\delta(q, t_1) = (p, t, N)$ and $b_1b_2b_3b_4 = ptt_2t_3$ or
  - $\delta(q, t_1) = (p, t, L)$ and $b_1b_2b_3b_4 = t'tt_2t_3$,
- $a_1a_2a_3a_4 = t_1qt_2t_3$ and
  - $\delta(q, t_2) = (p, t, R)$ and $b_1b_2b_3b_4 = t_1tpt_3$, or
  - $\delta(q, t_2) = (p, t, N)$ and $b_1b_2b_3b_4 = t_1ptt_3$ or
  - $\delta(q, t_2) = (p, t, L)$ and $b_1b_2b_3b_4 = pt_1tt_3$,
- $a_1a_2a_3a_4 = t_1t_2qt_3$ and
  - $\delta(q, t_3) = (p, t, N)$ and $b_1b_2b_3b_4 = t_1t_2pt$ or
  - $\delta(q, t_3) = (p, t, L)$ and $b_1b_2b_3b_4 = t_1pt_2t$.

In the first cycle it is not allowed that the last letter of a syllable is a state, because then the syllable is not a valid configuration of a Turing machine. In all other cycles the syllables may already be shortened and therefore have to be only prefixes of valid configurations. Here the last letter of a syllable can be a state. It is even possible that a state is the last letter of a syllable, when none of the last four symbols of the previous syllable is a state. As the conditions for the last two symbols and the deleted parts of each syllable were already checked, we have some more conditions for the last four letters of each syllable that are allowed:

- $a_1a_2a_3a_4 = t_1t_2qt_3$ and
  - $\delta(q, t_3) = (p, t, R)$ and $b_1b_2b_3b_4 = t_1t_2tp$,
- $a_1a_2a_3a_4 = t_1t_2t_3q$ and
  - $\delta(q, t') = (p, t, R)$ and $b_1b_2b_3b_4 = t_1t_2t_3t$, or
  - $\delta(q, t') = (p, t, N)$ and $b_1b_2b_3b_4 = t_1t_2t_3p$ or
  - $\delta(q, t') = (p, t, L)$ and $b_1b_2b_3b_4 = t_1t_2pt_3$,
- $a_1a_2a_3a_4 = t_1t_2t_3t_4$ and
  - $\delta(q, t') = (p, t, L)$ and $b_1b_2b_3b_4 = t_1t_2t_3p$.

M replaces the last two letters of the first syllable by #, but remembers the last four. In the next cycle it moves to the next syllable and checks for consistency. If the consistency check is positive, then it replaces the last two letters of this syllable by #. This is continued as for L_{copy^*}. After all syllables are shortened another restarting state is entered and all of the ## are rewritten into #. Then again each syllable is shortened by two symbols while consistency is checked.

If one of these checks is negative, M rejects immediately, otherwise M accepts after each syllable has length at most four and all checks are positive. Thus each $x \in \mathrm{VALC}(T)$ is accepted by M, and no other words are accepted.

This completes the proof. □

Thus we have shown that $\mathrm{VALC}(T) \in \mathcal{L}(\mathsf{det\text{-}nf\text{-}RW})$ holds. If every letter a is encoded by $a\#$, then the resulting language is accepted by a det-nf-R automaton. It is known that for language classes that include $\mathrm{VALC}(T)$ many decision problems are undecidable (see e.g. [HU79]). With this we have the following results.

Corollary 3.3.9. *The following problems are in general undecidable:*

INSTANCE	:	*A* det-nf-R *automaton* M.
QUESTION 1	:	*Is the language* $L(M)$ *non-empty?*
QUESTION 2	:	*Is the language* $L(M)$ *finite?*
QUESTION 3	:	*Is the language* $L(M)$ *regular?*
QUESTION 4	:	*Is the language* $L(M)$ *context-free?*

Observe that for a det-R-automaton, emptiness of the accepted language is easily decidable.

3.4 Monotone Restarting Automata

In this section we describe the expressive power of the various types of monotone nonforgetting restarting automata, compare them to monotone forgetting restarting automata, and give example languages. Monotone RRWW-automata recognize the context-free languages, a similar result is shown here for nf-RRWW-automata.

Monotonicity for nonforgetting restarting automata is defined as for forgetting restarting automata (see Definition 2.3.1 on page 20). The prefix mon- is used for monotone restarting automata.

Theorem 3.4.1. [MO06] $\mathcal{L}(\text{mon-nf-RLWW}) = \mathcal{L}(\text{mon-nf-RRWW}) = \mathcal{L}(\text{mon-nf-RWW}) = \text{CFL}$.

Proof. From Theorem 2.3.2 we know that $\text{CFL} = \mathcal{L}(\text{mon-RWW})$, and in Theorem 3.1.4 the equality $\mathcal{L}(\text{nf-RLWW}) = \mathcal{L}(\text{nf-RRWW})$ is shown. As the simulating nf-RRWW-automaton performs exactly the same cycles as the given nf-RLWW-automaton, it follows that the rewrites are done at the same places, and hence the one automaton is monotone if and only if the other is. Thus $\mathcal{L}(\text{mon-nf-RLWW}) = \mathcal{L}(\text{mon-nf-RRWW})$ holds, too.

Together with the trivial inclusions $\mathcal{L}(\text{mon-RWW}) \subseteq \mathcal{L}(\text{mon-nf-RWW}) \subseteq \mathcal{L}(\text{mon-nf-RRWW})$ it follows that $\text{CFL} \subseteq \mathcal{L}(\text{mon-nf-RWW}) \subseteq \mathcal{L}(\text{mon-nf-RRWW}) = \mathcal{L}(\text{mon-nf-RLWW})$ holds. It only remains to show that $\mathcal{L}(\text{mon-nf-RRWW}) \subseteq \text{CFL}$ holds.

Claim $\mathcal{L}(\text{mon-nf-RRWW}) \subseteq \text{CFL}$.

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be a nonforgetting monotone RRWW-automaton and let w be an input string. Further let $C_1, C_2, \ldots, C_m$ be the sequence of cycles that correspond to a computation of M starting on input w. For each cycle C_j the tape contents can be divided into three parts: a prefix x_j, the part u_j within the read/write window during the execution of the actual Rewrite-step, and the remaining suffix y_j.

As M is monotone, the positions where the Rewrite-steps are performed move from left to right across the tape. Accordingly, the prefixes x_j can be stored in a pushdown store. In the forgetting case the states in which M can be after reading the prefix x_j of the restarting configuration $q_0¢x_ju_jy_j\$$ are stored in an extra track on the tape. As M is nondeterministic this is in general a set of states. For nonforgetting RRWW-automata there is an extra track for each restarting state. The states that are derived starting from $q¢x_ju_jy_j\$$ are stored in the track for the restarting state q.

In detail: A PDA A uses a buffer of size k. On simulating M on the input w, A starts with reading the first k symbols into its buffer. Then it simulates one step of the transition relation δ of M that corresponds to the first step of the cycle C_1. The leftmost symbol of w that is no longer in the window of M is stored in the pushdown store together with $|Q|$ many extra tracks. All possible states that M can derive from $q_0¢w\$$ in one step are stored in the first track. As M is nondeterministic, this can be more than the state in the finite control, which corresponds to the first step of the cycle C_1.

In the second track, A stores all states that M can derive from $q_1¢w\$$ in one step. This is done for all states $q_i \in Q$, so that each of the tracks contains a subset of Q that corresponds to the states, in which M can be after performing its first step starting from the configurations $(q_i¢w\$)$.

This simulation is continued until A has stored x_1 on the pushdown store and has u_1 in its finite control. Then A performs the Rewrite-step within its finite control and pops symbols from the pushdown store until it has again k symbols in its finite control. But if there are not enough symbols left, the bottom marker is not popped from the pushdown store, instead A reads symbols from the tape until it has again k symbols in its finite control.

Unfortunately A is unable to read the rest of the tape to check whether M will do a restart, and if it will do one, in which state it will start the next cycle. So A has to guess the behavior of M and check if its guess was correct, when it reaches the right border marker \$ of the tape.

So A guesses the behavior of M for the rest of the tape and the state q_{j_1} in which M restarts after the first cycle. If the pushdown store is empty except for the bottom marker, A starts the simulation with q_{j_1}. Otherwise A chooses one of the states in the track corresponding to q_{j_1}. Remember that these states are derived when reading the prefix present on the pushdown and starting in state q_{j_1}. A chooses the one that corresponds to the first part of the second cycle C_2 of M. As M is monotone and the remaining tape was not touched since the simulation of the first rewrite, the next rewrite cannot be to the left of the topmost symbol of the pushdown store, and A can continue to simulate the second cycle C_2 in the same way as the first.

But A has to remember the state in which M was after the rewrite step of the first cycle and the restarting state with which the second cycle of M started. While A continues to read the tape it must simulate the second part of the first cycle to check if its guess was correct. After the rewrite of the second cycle the simulation of the third cycle starts and A must now remember the information for the second cycle and simulate the second part of the second cycle, too.

This is continued until A simulates the last cycle C_n. Then A simulates the tail of the computation of M, by again choosing one state from the pushdown store. Then it moves to the end of the tape and enters a final state if M accepts, and gets stuck if M rejects.

But there are m cycles and m is arbitrary large. So the simulation of some cycles must be joint, as A can do the simulation of the suffixes y_j only for a finite number of cycles. But fortunately there are at most $|Q|^2$ many different simulations to do.

For each simulation, A remembers an actual state which changes while A reads the rest of the tape, and a restarting state which is fixed. There can be only $|Q|$ many different actual states and to an actual state there can be only $|Q|$ many different restarting states and therefore there are at most $|Q|^2$ many different simulations to do. □

This completes the proof. □

Before we prove that the inclusions for monotone nonforgetting restarting automata are strict we need some technical lemmata and introduce the Kolmogorov complexity (see e.g. [LV97]). $K(x)$ is the Kolmogorov complexity of a word x over a finite alphabet Σ of cardinality at least two. The Kolmogorov complexity is the shortest description of a word.

Definition 3.4.2. *A word $x \in \Sigma^+$ is called incompressible if $K(x) \geq |x|$ holds, it is called $c-$incompressible if $K(x) \geq |x| - c$ holds, and it is called random if $K(x) \geq |x| - 4\log_3 |x|$ holds.*

Proposition 3.4.3. *[JO06] Let A be a deterministic finite-state acceptor with tape alphabet $\Gamma \supseteq \Sigma \neq \emptyset$. Then there exists a constant $n_0 \in \mathbb{N}_+$ such that, for each integer $n > n_0$ and each random word $w \in \Sigma^n$, the following condition is satisfied for each word $v \in \Sigma^+$: Assume A is in state q when it enters the factor wv on its tape from the left, and that it reaches state q' when its head is located inside the factor v. Then A already encounters state q' while its head is still inside the prefix of w of length $\log_s^2 n$ where $s = |\Sigma|$.*

With this proposition the following corollary can be shown.

Corollary 3.4.4. *Let M be a* nf-RRWW*-automaton with tape alphabet Γ. Then there exists a constant $n_0 \in \mathbb{N}_+$ such that, for each integer $n > n_0$ and each random word $w \in \Gamma^n$, the following condition is satisfied for each word $v \in \Gamma^+$: Assume M is in state q when it enters the factor wv*

on its tape from the left, and that it reaches state q' with window content u when its head is located inside the factor v. Then M already encounters state q' and window content u while its head is still inside the prefix x of w of length $\log_s^2 n$ where $s = |\Gamma|$.

If M is able to perform a rewrite step inside v, then it can perform the same rewrite step inside the prefix x of w, that is, it has to perform this rewrite step nondeterministically.

Proof. From M we will construct a deterministic finite-state acceptor A that fulfills the conditions of Proposition 3.4.3. A reads the first k symbols into its finite control, where k is the window size of M, then it simulates the transition relation of M without the rewrites. As the transition relation of M behaves like a finite-state acceptor except for the rewrite and restart steps we can assume without loss of generality that the MVR-steps of M are performed deterministically. Thus it may still be that M can perform a rewrite step nondeterministically, but all MVR-steps are performed deterministically. Therefore whenever M is in state q' and sees the window content u, then A is in state q''. If this happens while the window of M is inside v, then so is A's head and therefore, because of Proposition 3.4.3, A has reached a corresponding state q'' already inside the prefix x of w and so did M.

If M is able to rewrite $u \to v$ while it is in state q', then M was able to rewrite $u \to v$ in the prefix x of w. □

Lemma 3.4.5. *Let $z \in \Sigma^*$, ϕ an injective morphism, $L = \{wz\phi(w^R) | w \in \Sigma^*\}$ a palindrome language and $M = (Q, \Sigma, \Sigma, ¢, \$, q_0, k, \delta)$ a monotone* nf-RRW*-automaton. Then there exists an integer c such that at the start of an accepting computation on input $xz\phi(x^R)$, where x is a large incompressible word over Σ, M can perform at most c rewrites completely in x, rewriting it to x_1 before it rewrites parts of the suffix $z\phi(x^R)$. In addition, the suffix of x_1 of length $|x| - c\log^2(|x|)$ is random.*

Proof idea. Let $z \in \Sigma^*$, ϕ an injective morphism, $L = \{wz\phi(w^R) | w \in \Sigma^*\}$ a palindrome language and $M = (Q, \Sigma, \Sigma, ¢, \$, q_0, k, \delta)$ a monotone nf-RRW-automaton and x a large incompressible word. Assume for a contradiction that M starts an accepting computation with an arbitrary large number of rewrites completely inside the prefix x. Then it follows with counting arguments, as explained in [Ott08], that M will also accept words that do not belong to L. That the suffix of the remaining word is random is shown similar to the proof of Proposition 9 of [Ott08].

Lemma 3.4.6. *Let $z \in \Sigma^*$, ϕ an injective morphism, $L = \{wz\phi(w^R) | w \in \Sigma^*\}$ be a palindrome language, and $M = (Q, \Sigma, \Sigma, ¢, \$, q_0, k, \delta)$ a monotone* nf-RRW*-automaton. If in an accepting computation starting from $xz\phi(x^R)$, where x is a large incompressible word over Σ, M does perform a sequence of cycles $(q_0, xz\phi(x^R)) \vdash_M^{c^*} (q, w_1w_2) = (q, w_1x_1ux_2) \vdash_M^c (p, w_1x_1vx_2)$, where w_2 is the unchanged suffix of $\phi(x^R)$, then $|x_1|$ is bounded by a constant.*

Proof. Assume for a contradiction that x_1 is unbounded. $(q, w_1x_1ux_2)$ and $(p, w_1x_1vx_2)$ are promising configurations, therefore M will accept $(p, w_1x_1vx_2)$. How does the remaining accepting computation look like? M must compare symbols to their counterparts. But before M can again do a rewrite in w_1 it has to delete x_1 almost completely, because of the monotonicity of M. But as x_1 is unbounded it contains a regular pumping factor $x_1 = x_{1,1}yx_{1,2}$, such that for all $l \geq 0$, M can also perform the following sequence of cycles $(q_0, wx_{1,1}y^lx_{1,2}ux_2) \vdash_M^{c^*} (q, w_1x_{1,1}y^lx_{1,2}ux_2) \vdash_M^c (p, w_1x_{1,1}y^lx_{1,2}vx_2)$. And again M must delete $x_{1,1}y^lx_{1,2}$ almost completely, before it can compare symbols to their counterparts.

Thus both computations delete the factor x_1 or $x_{1,1}y^lx_{1,2}$ almost completely. During this process the suffix vx_2 may be rewritten as well, but then these rewrites are performed in both words

identically. As all rewrites during this part of the computation that effect x_1 or $x_{1,1}y^lx_{1,2}$ are rewriting the suffix of x_1 or $x_{1,1}y^lx_{1,2}$, respectively, it follows that M can check only regular conditions for x_1 or $x_{1,1}y^lx_{1,2}$, respectively. This is shown similar as in the proof of Lemma 3.3.3. M is able to rewrite the suffix vx_2 and store information on the tape, but again because of the monotonicity only $k-1$ new symbols can be written on the tape.

Therefore M will end in both cases in the same configuration. Thus there exists a computation such that the initial configuration $(q_0, \phi^{-1}(x_{1,1}yx_{1,2}ux_2)^R)zx_{1,1}y^lx_{1,2}ux_2)$ is transformed into a promising configuration. But if x_1 is not equal to y_1^my for a palindrome y_1, then $wx_{1,1}y^lx_{1,2}ux_2 \notin L$ holds for some $l \geq 0$. And as x is c-incompressible, x and therefore $\phi(x)$ does not contain a factor y_1^my for a large m. This contradicts the error preserving property. □

To show that the inclusions for the different variants of monotone nonforgetting restarting automata are proper we use the following languages:

$$\begin{aligned}
L_{\mathsf{pal}} :=& \{ww^R \mid w \in \{a,b\}^*\} && \in \mathcal{L}(\text{mon-nf-RWW}) && \setminus \mathcal{L}(\text{mon-nf-RW})\\
L_{\mathsf{pal}'} :=& \{ww^R\# \mid w \in \{a,b\}^*\} && \in \mathcal{L}(\text{mon-nf-RW}) && \setminus \mathcal{L}(\text{mon-nf-R})\\
L_{\mathsf{pal}''} :=& \{wc\phi(w^R) \mid w \in \{a,b,c\}^* \\
& \phi(a) = aa,\ \phi(b) = ab,\ \phi(c) = ba,\} && \in \mathcal{L}(\text{det-mon-RL}) && \setminus \mathcal{L}(\text{mon-nf-RW})\\
L_4 :=& \{\, a^nb^m \mid 0 \leq n \leq m \leq 2n \,\} && \in \mathcal{L}(\text{mon-R}) && \setminus \mathcal{L}(\text{det-RLW}).
\end{aligned}$$

In the next lemmata we show that they belong to the given language classes. L_{pal} is known to be a context-free language, so the following lemma shows that the first inclusion is proper.

Lemma 3.4.7. *The language L_{pal} is not accepted by any monotone* nf-RW*-automaton.*

Proof. Let $M = (Q, \Sigma, \Sigma, ¢, \$, q_0, k, \delta)$ be a monotone nf-RW-automaton that accepts L_{pal} and let x be a large incompressible word. In an accepting computation, starting with input xx^R, M can perform only a fixed number of rewrites completely in the prefix x (Lemma 3.4.5) by performing a sequence of cycles $(q_0, xx^R) \vdash_M^{c^n} (q, x_1x^R)$. The suffix of x_1 of length $|x| - n\log^2(|x|)$ is random. The next rewrite is performed in the middle of the tape (Lemma 3.4.6), but from Corollary 3.4.4 it follows that the next rewrite after that one can be performed in a prefix of the random suffix of x_1. This contradicts the assumption that M is monotone.

□

Lemma 3.4.8.

$$L_{\mathsf{pal}'} := \{ww^R\# \mid w \in \{a,b\}^*\} \in \mathcal{L}(\text{mon-nf-RW}) \setminus \mathcal{L}(\text{mon-nf-R})$$

Proof. $L_{\mathsf{pal}'}$ can be recognized by a mon-nf-RW-automaton M with the following meta-instructions:

$$\begin{aligned}
&(1)(q_0, ¢\{a,b\}^*, aa \to \#, q_1)\\
&(2)(q_0, ¢\{a,b\}^*, bb \to \#, q_1)\\
&(3)(q_1, ¢\{a,b\}^*, b\#b \to \#, q_1)\\
&(4)(q_1, ¢\{a,b\}^*, a\#a \to \#, q_1)\\
&(5)(q_0¢\#\$, \mathsf{ACCEPT})\\
&(5)(q_1¢\#\#\$, \mathsf{ACCEPT})
\end{aligned}$$

In the first cycle M guesses the middle and places a # there. Then it enters the state q_1. In this state it deletes the two symbols near the # if they coincide. M does this until it accepts the word ##.

This language cannot be recognized by a mon-nf-R-automaton, because from the previous lemma we know that $L_{\mathsf{pal}} \notin \mathcal{L}(\text{mon-nf-RW})$, and the extra symbol # at the end does not help for a mon-nf-R-automaton. Thus the assertion follows. □

It is a strong conjecture that the language L_{pal} is also not included in $\mathcal{L}(\text{mon-nf-RRW})$, because reading after the rewrite should not help by accepting palindromes.

Conjecture 3.4.9.

$L_{\mathsf{pal}} \notin \mathcal{L}(\text{mon-nf-RRW})$ *and* $L_{\mathsf{pal}'} \notin \mathcal{L}(\text{mon-nf-RR})$.

Lemma 3.4.10.

$L_{\mathsf{pal}''} \in \mathcal{L}(\text{det-mon-RL}) \setminus \mathcal{L}(\text{mon-nf-RW})$.

Proof. $L_{\mathsf{pal}''}$ is recognizable by a det-mon-RL-automaton M. M can perform MVL-steps, so it can determine the last c in the word. Then it deletes the corresponding letters around it. M accepts the word $¢c\$$.

In each cycle, M works in two phases, it first moves to the right border marker \$ and then back to the last occurrence of the letter c. The move right steps are expressed by the first part of each "meta-instruction" that starts with MVR. In the second phase it performs only MVL-steps to find the rightmost occurrence of the letter c and performs a rewrite if the surrounding letters correspond. This is expressed by the second part of each "meta-instruction" that start with MVL.

$$
\begin{array}{ll}
\mathsf{MVR}(¢\{a,b,c\}^*\$) & \mathsf{MVL}(\$\{a,b\}^*, aaca \to c) \\
\mathsf{MVR}(¢\{a,b,c\}^*\$) & \mathsf{MVL}(\$\{a,b\}^*, bacb \to c) \\
\mathsf{MVR}(¢\{a,b,c\}^*\$) & \mathsf{MVL}(\$\{a,b\}^*, abcc \to c) \\
\quad (¢c\$, \mathsf{ACCEPT}) &
\end{array}
$$

Claim: $L_{\mathsf{pal}''} \notin \mathcal{L}(\text{mon-nf-RW})$

Observe that the morphism ϕ is injective. Assume that there exists a nf-RW-automaton M with $L(M) = L_{\mathsf{pal}''}$. M cannot accept all words belonging to $L_{\mathsf{pal}''}$ in tail computations. Thus M must perform some cycles to accept a word $wc\phi(w^R) \in L_{\mathsf{pal}''}$ if w is sufficiently large. It follows from Lemma 3.4.6 that M cannot perform rewrites to the right of the last occurrence of the letter c. If w is a large incompressible word, then it follows from Lemma 3.4.5 that M can only perform finitely many rewrites completely in w and that the suffix of the derived word w' is random.

Assume that an accepting computation of the word $w_1w_2c\phi((w_1w_2)^R) \in L_{\mathsf{pal}''}$ starts with the following sequence of cycle, where w_1 is an incompressible word.

$$q_0¢w_1w_2c\phi((w_1w_2)^R)\$ \vdash_M^{c^*} q¢w_1'w_2c\phi((w_1w_2)^R)\$ \vdash_M^{c} q'¢w_1'w\phi(w_1^R)\$ \vdash_M^{c} q'¢w_1'w'\phi(w_1^R)\$$$

As w_1 is incompressible it follows from Lemma 3.4.5 that a suffix of w_1' is random. Now it follows from Corollary 3.4.4 that instead of the second rewrite inside w M can perform a rewrite in the

prefix of the suffix of w_1, therefore M can perform nonmonotone computations and therefore M is not monotone.

□

The separation language

$$L_4 := \{\, a^n b^m \mid 0 \le n \le m \le 2n \,\} \in \mathcal{L}(\text{mon-R}) \setminus \mathcal{L}(\text{det-RLW}).$$

was already used in the previous chapter and it is well known that it separates these two language classes. With these results we have the same taxonomy for monotone nonforgetting restarting automata as for the forgetting types.

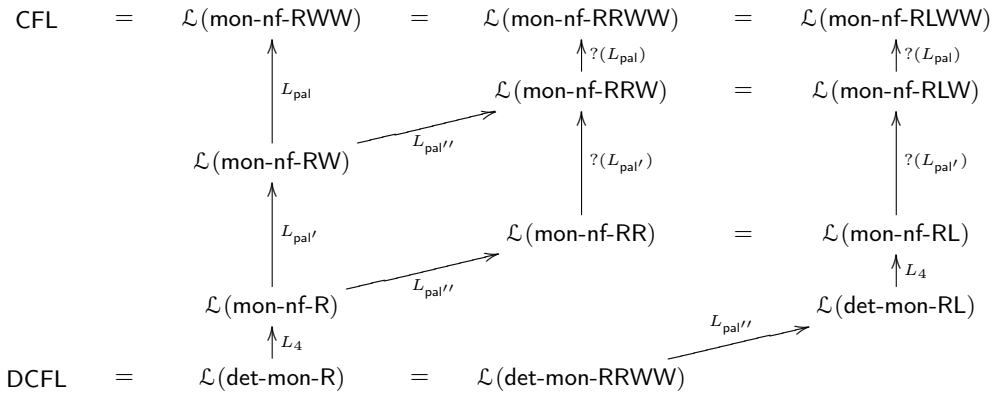

Figure 3.2: The taxonomy of monotone nonforgetting restarting automata

If we compare monotone forgetting with monotone nonforgetting restarting automata, then we see that only the trivial inclusions hold and that they are proper.

Lemma 3.4.11. *The classes* $\mathcal{L}(\text{mon-nf-R})$ *and* $\mathcal{L}(\text{mon-RRW})$ *are incomparable with respect to inclusion.*

Proof. $\mathcal{L}(\text{mon-nf-R}) \not\subset \mathcal{L}(\text{mon-RRW})$:

The example language $L_2 \notin \mathcal{L}(\text{mon-RRW})$ is recognized by the monotone nf-R-automaton M with the following meta-instructions:

$(1)(q_0, ¢a^*, abb \to b, q_1)$	$(5)(q_0, ¢a^*, abbb \to b, q_2)$
$(2)(q_1, ¢a^*, abb \to b, q_1)$	$(6)(q_2, ¢a^*, abbb \to b, q_2)$
$(3)(q_1, ¢ab\$, \text{ACCEPT})$	$(7)(q_2, ¢b^+\$, \text{ACCEPT})$
$(4)(q_0, ¢ab\$, \text{ACCEPT})$	$(8)(q_0, ¢b^*\$, \text{ACCEPT})$

M first guesses to which of the sublanguages the input belongs, enters an according restarting state, and accepts this sublanguage. The empty string and ab are accepted without performing a single cycle.

$\mathcal{L}(\text{mon-RRW}) \not\subset \mathcal{L}(\text{mon-nf-R})$:

The language $\tilde{L}_{\text{pal}'} = L_{\text{pal}'} \cup \{w\#w^R\#|w \in \{a,b\}^*\}$ is accepted by a mon-RW-automaton, but not by a mon-nf-R-automaton, because it cannot accept $L_{\text{pal}'}$. □

Lemma 3.4.12. *The classes* $\mathcal{L}(\text{mon-nf-RW})$ *and* $\mathcal{L}(\text{mon-RR})$ *are incomparable with respect to inclusion.*

Proof. $\mathcal{L}(\text{mon-RR}) \not\subset \mathcal{L}(\text{mon-nf-RW})$:

$L_{\text{pal}''} \notin \mathcal{L}(\text{mon-nf-RW})$ (see Lemma 3.4.10) can be recognized by a mon-RR-automaton with the following meta-instructions, where $d \in \{a,b,c\}$ holds:

$$(1)(\text{¢}\{a,b,c\}^*, dc\phi(d) \to c, \{a,b\}^*\$)$$
$$(2)(\text{¢}c\$, \text{ACCEPT})$$

The other direction was already shown in the proof of the last lemma. □

3.5 Deterministic Monotone Restarting Automata

A restarting automaton is deterministic monotone if it is deterministic and monotone. We have seen that deterministic nonforgetting restarting automata are more powerful than deterministic forgetting ones. This is as well true for monotone restarting automata that are deterministic.

Forgetting deterministic monotone restarting automata recognize exactly DCFL, and the R-model is as powerful as the RRWW-model. This is shown in [JMPV99] by a simulation of a det-mon-RRWW-automaton by a deterministic PDA. This result can be adapted to nonforgetting automata, however only in a weaker form.

Theorem 3.5.1. $\text{DCFL} = \mathcal{L}(\text{det-mon-nf-R}) = \mathcal{L}(\text{det-mon-nf-RWW})$.

Proof. $\text{DCFL} \subseteq \mathcal{L}(\text{det-mon-nf-R}) \subseteq \mathcal{L}(\text{det-mon-nf-RWW})$ is already known, see Theorem 2.3.3 on page 20. We prove now that the inclusion $\mathcal{L}(\text{det-mon-nf-RWW}) \subseteq \text{DCFL}$ is valid.

Claim $\mathcal{L}(\text{det-mon-nf-RWW}) \subseteq \text{DCFL}$

Proof. Let $M = (Q, \Sigma, \Gamma, \text{¢}, \$, q_0, k, \delta)$ be a nonforgetting deterministic monotone RWW-automaton, let w be an input string, and let $C_1, C_2, \ldots, C_m$ be the sequence of cycles that correspond to the computation of M starting with $q_0\text{¢}w\$$. For each cycle C_j the tape contents can be divided into three parts: a prefix x_j, the part u_j within the read/write window during the execution of the actual Rewrite-step, and the remaining suffix y_j.

As M is monotone, the positions where the Rewrite-steps are performed move from left to right across the tape. Accordingly, the prefixes x_j can be stored in a pushdown store. In the forgetting case the state in which M can be after reading the prefix x_j of the restarting configuration $q_0\text{¢}x_ju_jy_j\$$ is stored in an extra track on the tape. For nonforgetting RRWW-automata there is an extra track for each restarting state. The state that is derived starting from $q\text{¢}x_ju_jy_j\$$ is stored in the track for the restarting state q. This is done analogously to the proof of Theorem 3.4.1.

In detail: A DPDA A uses a buffer of size k. On simulating M on the input w, A starts with reading the first k symbols into its buffer. Then it simulates one step of the transition relation δ

of M that corresponds to the first step of the cycle C_1. As M is deterministic there is only one possible step that is applicable to the configuration $q_0 ¢ w \$$. The leftmost symbol of w that is no longer in the window of M is stored in the pushdown store together with $|Q|$ many extra tracks. The state that M derives from $q ¢ w \$$, $q \in Q$, in one step is stored in the track assigned to q. As M is deterministic, there is at most one state in each track.

This simulation is continued until A has stored x_1 on the pushdown store and has u_1 in its finite control. Then A performs the Rewrite-step within its finite control and pops symbols from the pushdown store until it has again k symbols in its finite control. If there are not enough symbols left, the bottom marker is not popped from the pushdown store. Instead A reads symbols from the tape until it has again k symbols in its finite control.

M restarts immediately after the rewrite, so A can continue with the state from the track that corresponds to the restarting state in which M restarts. As M is monotone, the next rewrite is not to the left of the current k symbols in the finite control. So A can continue to simulate the second cycle C_2 in the same way as the first.

This is done until either M halts and accepts, or halts and rejects. Accordingly A enters a final state or a trap state which it never leaves again. Thus, A accepts the same language as M. □

Thus we know that $\mathsf{DCFL} \subseteq \mathcal{L}(\text{det-mon-nf-R}) \subseteq \mathcal{L}(\text{det-mon-nf-RWW}) \subseteq \mathsf{DCFL}$ and therefore $\mathsf{DCFL} = \mathcal{L}(\text{det-mon-nf-R}) = \mathcal{L}(\text{det-mon-nf-RWW})$. □

This theorem only covers restarting automata that restart immediately after executing a rewrite step. In the next paragraphs we show that this theorem is only valid for these restarting automata. Because already det-mon-nf-RR-automata accept languages that are not deterministic context-free.

Example 3.5.2. $L_{\overline{\mathsf{pal}}} = \{wc\bar{w}^R \mid w \in \{a,b,c\}^*, \bar{a} = a, \bar{b} = b, \bar{c} = \varepsilon\} \in \mathcal{L}(\text{det-mon-nf-RR})$*:*

(1)$(q_0, ¢ce \to ¢e, \{a,b,c\}^*\$, q_0)$ $|e \in \{a,b,c\}$
(2)$(q_0, ¢c\$, \mathsf{ACCEPT})$
(3)$(q_0, ¢a \to ¢, (\{a,b\}^*c)^2\{a,b,c\}^*\$, q_a)$
(4)$(q_0, ¢a \to ¢, \{a,b\}^*c\{a,b\}^*\$, q'_a)$
(5)$(q_a, ¢\{a,b\}^*, c \to \varepsilon, (\{a,b\}^*c)^2\{a,b,c\}^*\$, q_a)$
(6)$(q_a, ¢\{a,b\}^*, c \to \varepsilon, \{a,b\}^*c\{a,b\}^*\$, q'_a)$
(7)$(q'_a, ¢\{a,b\}^*, aca \to c, \{a,b\}^*\$, q'_a)$
(8)$(q'_a, ¢\{a,b\}^*, bcb \to c, \{a,b\}^*\$, q'_a)$
(9)$(q'_a, ¢ca\$, \mathsf{ACCEPT})$
(10)$(q_0, ¢b \to ¢, (\{a,b\}^*c)^2\{a,b,c\}^*\$, q_b)$
(11)$(q_0, ¢b \to ¢, \{a,b\}^*c\{a,b\}^*\$, q'_b)$
(12)$(q_b, ¢\{a,b\}^*, c \to \varepsilon, (\{a,b\}^*c)^2\{a,b,c\}^*\$, q_b)$
(13)$(q_b, ¢\{a,b\}^*, c \to \varepsilon, \{a,b\}^*c\{a,b\}^*\$, q'_b)$
(14)$(q'_b, ¢\{a,b\}^*, aca \to c, \{a,b\}^*\$, q'_b)$
(15)$(q'_b, ¢\{a,b\}^*, bcb \to c, \{a,b\}^*\$, q'_b)$
(16)$(q'_b, ¢cb\$, \mathsf{ACCEPT})$

In the initial state M accepts the input c, and if the input is cx, $x \in \{a,b,c\}^+$ it deletes the first c and restarts in q_0. Thus, in the restarting state q_0, M deletes cs at the left end of the tape, provided the tape content is not $¢c\$$. This is achieved by the rewrite step $¢ce \to ¢e$, $e \in \{a,b,c\}$.

If the tape content does not start with c, then M deletes the first letter d, $d \in \{a,b\}$, on the tape and moves on, counting the remaining cs on the tape. If there are at least two cs left it restarts in the restarting state q_d, if there is only one c left on the tape, then M restarts in q'_d.

In $q_a(q_b)$ M deletes the first c and counts the remaining cs. If there are at least two cs left, it restarts in $q_a(q_b)$. If there is only one c left, then M restarts in $q'_a(q'_b)$. In q'_a M simply accepts the language $\{wcw^Ra | w \in \{a,b\}^*\}$. This is done by deleting the letters next to the c if they coincide. In

q'_b M simply accepts the language $\{wcw^Rb|w \in \{a,b\}^*\}$. Again this is done by deleting the letters next to the c if they coincide. Thus, an input x is accepted if it belongs to $L_{\overline{\mathsf{pal}}}$. M is monotone, because it deletes at the left border marker until it finds a letter a or b. Then it deletes all but the last c from left to right, and finally it deletes the letters next to the last remaining c.

To show that $L_{\overline{\mathsf{pal}}} \notin \mathsf{DCFL}$ is valid, we use Ogdens Lemma for deterministic context-free languages.

Lemma 3.5.3 (Ogdens Lemma for DCFL)**.** [Har79, JMPV97b]

Let $L \in \mathsf{DCFL}$, then there is an integer $p(L)$ such that for every $w \in L$ and for every set K of positions in w the following holds: if $|K| \geq p(L)$, then there is a factorization $\varphi = (v_1, ..., v_5)$ of w such that:

1. *$v_2 \neq \varepsilon$;*
2. *For all $n \geq 0$, $v_1v_2^nv_3v_4^nv_5 \in L$;*
3. *if $K/\varphi = \{K_1, ..., K_5\}$, then*
   - *i) $K_1, K_2, K_3 \neq \emptyset$ or $K_3, K_4, K_5 \neq \emptyset$,*
   - *ii) $|K_2 \cup K_3 \cup K_4| \leq p(L)$;*
4. *if $v_5 \neq \varepsilon$, then for each $m, n \geq 0, u \in \Sigma^*$, $v_1v_2^{m+n}v_3v_4^nu \in L$ if and only if $v_1v_2^mv_3u \in L$.*

Lemma 3.5.4. DCFL *is strictly included in* $\mathcal{L}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR})$.

Proof. We have seen that $L_{\overline{\mathsf{pal}}} \in \mathcal{L}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR})$ is valid. Now we show that this language is not deterministic context-free.

We use $w = a^kca^k \in L_{\overline{\mathsf{pal}}}$, where $k = p(L_{\overline{\mathsf{pal}}}) + 1$ holds and the set K contains all positions after the c. Then we get a factorization of w with $v_1 = a^i, v_2 = a^j, v_3 = a^{k-i-j}ca^l, v_4 = a^j, v_5 = a^{k-l-j}$. This is the only way to factor w so that 1. and 2. is complied. The pumping factors v_2 and v_4 must have the same length and must occur on different sides of the c, because this is the only way such that $v_1v_2^nv_3v_4^nv_5 \in L_{\overline{\mathsf{pal}}}$ holds. With this factorization we know that $K_1 = K_2 = \emptyset$ and $K_3, K_4, K_5 \neq \emptyset$ hold.

As $|K_2 \cup K_3 \cup K_4| = l + j \leq p(L_{\overline{\mathsf{pal}}}) < k$ must be valid, $v_5 \neq \varepsilon$ holds. Thus we can use 4. to prove that $L_{\overline{\mathsf{pal}}}$ is not in DCFL.

For $m = 0$ and $u = ca^{k+l-j}$ it follows that $v_1v_3u = a^i\, a^{k-i-j}ca^l\, ca^{k+l-j} = a^{k-j}ca^lca^{k-j+l} \in L_{\overline{\mathsf{pal}}}$ holds. But for every $n > 0$ we get $v_1v_2^nv_3v_4^nu \notin L_{\overline{\mathsf{pal}}}$, because the number of a's before the last c increases, while the number of a's after the last c remains unchanged. Thus $L_{\overline{\mathsf{pal}}} \notin \mathsf{DCFL}$ holds. □

With small changes we can find languages that prove the strictness of the inclusions between $\mathcal{L}(\mathsf{det\text{-}mon\text{-}nf\text{-}RR})$, $\mathcal{L}(\mathsf{det\text{-}mon\text{-}nf\text{-}RRW})$ and $\mathcal{L}(\mathsf{det\text{-}mon\text{-}nf\text{-}RRWW})$. To simplify the description of the restarting automata, we use the additional restriction $|w|_c \geq 1$ for the other languages. Without this restriction the same results hold. Here (EF), for $e, f \in \{a, b, c\}$, are extra symbols that are used to encode the tape content.

$$
\begin{aligned}
L_{\overline{\mathsf{pal}}} &= \{wc\bar{w}^R \mid w \in \{a,b,c\}^*, \bar{a}=a, \bar{b}=b, \bar{c}=\varepsilon\} \in \mathcal{L}(\text{det-mon-nf-RR}) \setminus \mathsf{DCFL} \\
L_{\mathsf{pal}''} &= \{wc\phi(w^R) \mid w \in \{a,b,c\}^*, \phi(a)=aa,\ \phi(b)=ab,\ \phi(c)=ba, |w|_c \geq 1\} \\
&\qquad \in \mathcal{L}(\text{det-mon-nf-RRWW}) \setminus \mathcal{L}(\text{det-mon-nf-RRW}) \\
L_{\overline{\mathsf{pal}}''} &= \{w(AA)(AB)(AC)(BA)(BB)(BC)(CA)(CB)(CC) \mid w \in L_{\mathsf{pal}''}\} \\
&\qquad \in \mathcal{L}(\text{det-mon-nf-RRW}) \setminus \mathcal{L}(\text{det-mon-nf-RR})
\end{aligned}
$$

In $L_{\overline{\mathsf{pal}}}$ the morphism $d \to \overline{d}$ only removes the cs. $L_{\mathsf{pal}''}$ was already used in Lemma 3.4.10. In $L_{\overline{\mathsf{pal}}''}$ each word from $L_{\mathsf{pal}''}$ is followed by nine additional symbols that are used to encode two input symbols of $L_{\mathsf{pal}''}$.

Lemma 3.5.5. $L_{\mathsf{pal}''} \in \mathcal{L}(\text{det-mon-nf-RRWW}) \setminus \mathcal{L}(\text{det-mon-nf-RRW})$.

Proof. With the restriction to subwords w of even length we can use the following deterministic monotone nf-RRWW automaton M'' that accepts the language $L_{\mathsf{pal}''}$. This restriction is not necessary, it only shortens the description of the automaton. M'' uses the auxiliary symbols $\Gamma_0 = \{(AA),(AB),(AC),(BA),(BB),(BC),(CA),(CB),(CC)\}$. Let $d,e \in \{a,b,c\}$, let (DE) be the auxiliary symbol that corresponds to the two input symbols d and e.

(1) $(q_0, ¢\Gamma_0^*, de \to (DE), \{a,b,c\}^2\{a,b\}^*c\{a,b,c\}^*\$, q_0)$
(2) $(q_0, ¢\Gamma_0^*, de \to (DE), c\{a,b\}^*\$, q_1)$
(3) $(q_1, ¢\Gamma_0^*, (DE)c\phi(ed) \to c, \{a,b\}^*\$, q_1)$
(4) $(q_1, ¢c\$, \mathsf{ACCEPT})$

M'' first encodes the word w with its auxiliary symbols. Again the technique from the previous restarting automata is used to deterministically restart in a different state, if only one c remains on the tape. Here it is only restarted in a different state if the suffix starts with the last c. The last remaining c now marks the middle. The symbols around this c are deleted if the encoded symbol (DE) match with $\phi((de)^R) = \phi(ed)$.

Claim: $L_{\mathsf{pal}''} \notin \mathcal{L}(\text{det-mon-nf-RRW})$.

Proof. Assume that there exists a det-mon-nf-RRW-automaton $M = (Q, \Sigma, \Sigma, ¢, \$, q_0, k, \delta)$, with $\Sigma = \{a,b,c\}$, for this language. Let $wc\phi(w^R) \in L_{\mathsf{pal}''}$ and w is a large incompressible word over Σ. Then M is unable to deterministically find the last occurrence of the letter c in one cycle. Thus it needs to process the input from left to right. From Lemma 3.4.5 we know that M can only perform a finite number of rewrites inside of w in an accepting computation and that the remaining word has a long suffix that is random. Thus with Corollary 3.4.4 it follows that M cannot deterministically perform a rewrite in the middle of the word afterwards. Thus M cannot accept $L_{\mathsf{pal}''}$. □

□

Lemma 3.5.6. $L_{\overline{\mathsf{pal}}''} \in \mathcal{L}(\text{det-mon-nf-RRW}) \setminus \mathcal{L}(\text{det-mon-nf-RR})$.

Proof. Again we use the restriction to subwords w of even length to describe a deterministic monotone nf-RRW automaton M that accepts the language $L_{\overline{\mathsf{pal}}''}$. M uses the additional symbols $\{(AA),(AB),(AC),(BA),(BB),(BC),(CA),(CB),(CC)\}$ at the end of each word to encode pairs

of input symbols . Let $d, e \in \{a, b, c\}$, let (DE) be the additional symbol that corresponds to the two input symbols d and e, and let $w_0 = (AA)(AB)(AC)(BA)(BB)(BC)(CA)(CB)(CC)$.

$$(1)(q_0, ¢\Gamma_0^*, de \to (DE), (\{a, b\}^* c)^2 \{a, b, c\}^* w_0 \$, q_0)$$
$$(2)(q_0, ¢\Gamma_0^*, de \to (DE), \{a, b\}^* c \{a, b\}^* w_0 \$, q_1)$$
$$(3)(q_1, ¢\Gamma_0^* \{a, b\}^*, dec\phi(ed) \to c, \{a, b\}^* w_0 \$, q_1)$$
$$(4)(q_1, ¢\Gamma_0^*, (DE)c\phi(ed) \to c, \{a, b\}^* w_0 \$, q_1)$$
$$(5)(q_1, ¢cw_0 \$, \mathsf{ACCEPT})$$

M behaves just like M'' in the proof of Lemma 3.5.5 except that the word w_0 must be at the right end of the tape. w_0 is not altered during the computation.

As $L_{\mathsf{pal}''} \notin \mathcal{L}(\mathsf{det\text{-}nf\text{-}RRW})$ holds and as the additional symbols at the end of the tape do not help an RR-automaton it follows that $L_{\overline{\mathsf{pal}}''} \notin \mathcal{L}(\mathsf{det\text{-}nf\text{-}RR})$ holds.

□

All of these languages are in $\mathcal{L}(\mathsf{det\text{-}mon\text{-}nf\text{-}RL})$, because we always look for the last c, which is trivial for RL-automata. We will now prove a result that goes much further. In this proof shrinking nonforgetting restarting automata are used. To recall the Definition 2.7.1 on page 27: A shrinking restarting automaton does not need to reduce the length of the tape in each rewrite step $u \to v$, it must only reduce, for a given weight function, the weight. That means the weight of v must be smaller than the weight of u.

Theorem 3.5.7. $\mathcal{L}(\mathsf{det\text{-}mon\text{-}nf\text{-}sh\text{-}RLWW}) = \mathcal{L}(\mathsf{det\text{-}mon\text{-}RL})$.

Proof. It is shown in [JMOP05] that $\mathcal{L}(\mathsf{det\text{-}mon\text{-}RL}) = \mathcal{L}(\mathsf{det\text{-}mon\text{-}RLWW})$ holds. The extension to shrinking restarting automata is obtained as follows.

A language L is accepted by a monotone deterministic shrinking RLWW automaton (sh-RLWW) if and only if L^R is accepted by a left-monotone deterministic sh-RLWW automaton (cf. Lemma 1 of [JMOP05]).

From the main results of [OJ03] we see that $\mathcal{L}(\mathsf{det\text{-}left\text{-}mon\text{-}sh\text{-}RLWW}) = \mathcal{L}(\mathsf{det\text{-}left\text{-}mon\text{-}RLWW})$, while Theorem 3 of [JMOP05] yields $\mathcal{L}(\mathsf{det\text{-}left\text{-}mon\text{-}RLWW}) = \mathcal{L}(\mathsf{det\text{-}left\text{-}mon\text{-}RL})$. Hence, we see that $\mathcal{L}(\mathsf{det\text{-}left\text{-}mon\text{-}sh\text{-}RLWW}) = \mathcal{L}(\mathsf{det\text{-}left\text{-}mon\text{-}RL})$. By using Lemma 1 of [JMOP05] again we obtain $\mathcal{L}(\mathsf{det\text{-}mon\text{-}RL}) = \mathcal{L}(\mathsf{det\text{-}mon\text{-}sh\text{-}RLWW})$.

Thus, it remains to show that every monotone deterministic nonforgetting sh-RLWW automaton can be simulated by a monotone deterministic sh-RLWW automaton. This can be done as follows.

Let M by a det-mon-nf-sh-RLWW automaton. It is simulated by a monotone deterministic sh-RLWW automaton M'. We can assume that M restarts immediately after the rewrite (see Lemma 3.3.2). Thus, M' is able to write the restarting state on the tape.

First M' scans the tape from left to right searching for symbols that encode a state and a tape symbol of M. In its internal state, M' remembers the rightmost state found in this way, and if no such symbol is found, then M' remembers the initial state of M. Next M' starts with the simulation of M at the left end of the tape with the remembered restarting state. M' continues with the simulation until it reaches the configuration in which M would perform a rewrite operation $u \to v$. M' executes this rewrite operation, encoding the new restarting state of M in the rightmost symbol of v. To ensure that the right-hand side of each rewrite step is not empty M' has a window of size $k+1$, where k is the window size of M. As M is monotone, the newly written restarting state is the rightmost state encoded in the tape inscription, and therewith it will be chosen correctly in

the next cycle. It is easily seen that M' simulates M cycle by cycle and that it accepts the same language as M. □

But $\mathcal{L}$(det-mon-RL) is strictly included in CRL, CFL and their intersection.

Lemma 3.5.8. $\mathcal{L}$(det-mon-nf-RLWW) = $\mathcal{L}$(det-mon-RL) $\subsetneq$ CRL $\cap$ CFL.

Proof. The inclusion in CFL has been shown in the previous section. The inclusion in CRL was shown in Theorem 2.4.9 on page 24. The strictness of the inclusion follows by the example language $L_2 = \{ a^n b^n \mid n \geq 0 \} \cup \{ a^n b^m \mid m > 2n \geq 0 \}$, which is context-free and Church-Rosser by [Ott06] Lemma 6, but cannot be accepted by a det-mon-RL-automaton, see Lemma 2.3.5 on page 21.

□

The only question left to answer is the exact relationship between $\mathcal{L}$(det-mon-nf-RRWW) and $\mathcal{L}$(det-mon-nf-RL). We first study shrinking deterministic monotone nonforgetting restarting automata. Then we use the obtained results to answer this question.

Theorem 3.5.9. $\mathcal{L}$(det-mon-nf-sh-RRWW) = $\mathcal{L}$(det-mon-RL).

Proof. $\mathcal{L}$(det-mon-nf-sh-RRWW) $\subseteq$ $\mathcal{L}$(det-mon-RL) follows immediately from Theorem 3.5.7.

For the other direction we use the fact that each det-mon-RL-automaton is correctness preserving. With Theorem 2.8.4 it follows that there exists a correctness preserving mon-RR-automaton that accepts the same language. Thus, if a language L belongs to L(det-mon-RL), then there exists a correctness preserving mon-RR-automaton $M = (Q, \Sigma, \Sigma, ¢, \$, q_0, k, \delta)$ with $L(M) = L$. M can be described by meta-instructions $(E_i, u_i \to v_i, E_i')$, E_i and E_i' are called the prefix- and suffix-condition, respectively. As M is correctness preserving a cycle that starts in a promising configuration leads again to a promising configuration.

We are now going to give a construction of a det-mon-nf-sh-RRWW-automaton $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k+1, \delta')$ that is able to check the regular conditions for a cycle before it performs the rewrite. Here $\Sigma' = \{A \mid a \in \Sigma\}$ and $\Gamma = \Sigma \cup \Sigma'$ hold. The weight of the letters in Σ is taken as two and the weight of the auxiliary symbols is one.

M' starts its computation by marking the first k symbols on the tape and moves to the right border marker. The prefix conditions of all meta-instructions are checked for the empty set and the suffix conditions are checked for the unmarked part of the tape. If one meta-instruction I_i is found for which the prefix- and the suffix-conditions are fulfilled and u_i is the marked part of the tape, then M' restarts in a restarting state indicating that the meta-instruction I_i is applicable. Otherwise it restarts in a state indicating that the marking must go on. Whenever M' detects that an accepting meta-instruction is applicable it accepts immediately.

If the marking goes on, then M' marks the next unmarked symbol and checks whether the sequence of all but the last k symbols of the marked part fulfills the prefix condition of I_i, the sequence of the last k symbols of the marked part are equal to u_i, and the unmarked part fulfills the suffix condition of I_i. If one meta-instruction I_i is found that fulfills all these conditions, then M' restarts in a restarting state indicating that the meta-instruction I_i is applicable. Otherwise it restarts in a state indicating that the marking must go on.

If a meta-instruction I_i is applicable, then it is applied in the next cycle. M' moves over the marked part of the tape, rewrites u_i to v_i, the v_i is written as marked symbols on the tape. While it performs this cycle it also checks what meta-instruction is applicable next. As M is monotone the next rewrite must completely contain v_i or it contains symbols from the unmarked part of the

tape. Thus either one meta-instruction I_j is applicable such that v_i is a suffix of u_j or M' can continue to mark letters on the tape.

To find the position of the u_i deterministically, M' needs a window of size $k+1$ to see one unmarked symbol on the tape while it has all of u_i in its window. This additional symbol remains unmarked and is not changed in the rewrite step.

Observe that M can be nondeterministic, thus the place for the rewrite is perhaps not deterministically detectable. But as M' tests at each place of the tape, whether a rewrite is applicable there or not, it can always check the prefix and suffix condition from left to right, because it does not need to find the place of the rewrite.

Claim: $L(M) = L(M')$.

Proof. $L(M) \subseteq L(M')$.

Let $w \in L(M)$, then there exists an accepting sequence of cycles starting with w:

$$w \vdash_M w_1 \vdash_M \ldots \vdash_M w_{n-1} \vdash_M w_n \vdash_M \mathsf{ACCEPT}.$$

M' starts simulating the first cycle $w = w'uw'' \vdash_M w'vw'' = w_1$ by marking the first k letters. Then it checks whether the conditions for one meta-instruction are fulfilled. In this case it restarts in a restarting state indicating that in the next cycle a rewrite of M can be simulated. In all other cases M' marks symbols until $w'u$ is marked. Then it restarts in a restarting state indicating that in the next cycle a rewrite of M can be simulated.

When the cycle of M is performed by M', remark that all conditions are already checked and the rewrite $W' \cdot U \cdot w'' \to W' \cdot V \cdot w''$ is at the last k marked symbols, here $W' = A_1 A_2 \ldots A_m$, if $w' = a_1 a_2 \ldots a_m$ holds. In this cycle, M' checks if the next rewrite has V as a suffix. As M is monotone, the next rewrite is not left of the last marked symbol. Thus either in the next cycle of M' the next cycle of M is performed if it has the same right distance, or M' continues to mark the tape until it detects the place for the next rewrite.

This is done until M' accepts the input Xy after n cycles, where $w_n = xy$ holds. Conversely if $w \in L(M')$, then beside the marking all rewrites simulate cycles of M. So for each accepting sequence of cycles of M' there exists a accepting sequence of cycles of M that starts with the same word. Thus $L(M') \subseteq L(M)$ holds. □

This completes the proof. □

This proof can be adapted for length reducing det-mon-nf-RRWW-automata.

Theorem 3.5.10. $\mathcal{L}$(det-mon-nf-RRWW) = $\mathcal{L}$(det-mon-RL).

Proof. $\mathcal{L}$(det-mon-nf-RRWW) $\subseteq$ $\mathcal{L}$(det-mon-nf-RLWW) is trivial and in Theorem 3.5.7 it was shown that $\mathcal{L}$(det-mon-nf-RLWW) = $\mathcal{L}$(det-mon-RL) holds.

The construction for the other direction is similar to the one in the previous Theorem. Again not the det-mon-RL-automaton, but a correctness preserving mon-RR-automaton that accepts the same language is simulated.

Let $M = (Q, \Sigma, \Sigma, ¢, \$, q_0, k, \delta)$ be a correctness preserving mon-RL-automaton with $(E_i, u_i \to v_i, E_i')$ the meta-instructions of M, where $1 \leq i \leq n$, E_i is again called the prefix-, E_i' the suffix-condition. This automaton can be simulated by a det-mon-nf-RRWW-automaton $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k+1, \delta')$ that is able to check the regular conditions for a cycle before it performs

the rewrite. Here $\Sigma_2' = \{(AB)|a, b \in \Sigma\}$ and $\Gamma = \Sigma \cup \Sigma_2'$ hold. M' works like the shrinking automaton in the proof of the previous Theorem with two differences.

First two input symbols are encoded in one auxiliary symbol, such that the encoding (marking) can be done with length reducing rewrites. This implies the second difference, because if more than one rewrite with the same right distance occurs in successive cycles, and each rewrite reduces the length only by one, then it is not so easy to simulate them one by one.

Thus every time M' detects a place for a rewrite $u_i \to v_i$, it also checks whether the next cycle has a rewrite $u_j \to v_j$ with the same right distance that means whether v_i is a suffix of u_j or not. In the affirmative M' performs both rewrites together in a single rewrite step. This ensures that the length of the tape is reduced by at least two, which means that even the encoded tape is reduced by at least one. In the negative M' can encode the letter d which is right of the rewrite if it is needed to have a length reducing rewrite step. This is possible, because the rewrite of the next cycle has a smaller right distance.

Thus it follows as for shrinking restarting automata that $L(M) = L(M')$ holds. □

This is the first time that some variants of RWW and RRWW-automata are separated. Just recently it was shown that the language class $\mathcal{L}$(det-mon-RL) coincides with the class of left-to-right regular languages (LRR). For a definition of LRR grammars and languages see [CC73].

Proposition 3.5.11. [Ott07] LRR = $\mathcal{L}$(det-mon-RL).

So the classes $\mathcal{L}$(det-mon-nf-RWW) and $\mathcal{L}$(det-mon-nf-RRWW) are not only different, but they both describe well-known language classes. Thus we get the following taxonomy of deterministic monotone nonforgetting restarting automata:

$$
\begin{array}{ccccc}
\text{LRR} & = & \mathcal{L}(\text{det-mon-nf-sh-RLWW}) & = & \mathcal{L}(\text{det-mon-RL}) \\
 & & \uparrow{\scriptstyle L_{\text{pal}''}} & & \\
 & & \mathcal{L}(\text{det-mon-nf-RRW}) & & \\
 & & \uparrow{\scriptstyle L_{\overline{\text{pal}}''}} & & \\
 & & \mathcal{L}(\text{det-mon-nf-RR}) & & \\
 & & \uparrow{\scriptstyle L_{\overline{\text{pal}}}} & & \\
\text{DCFL} & = & \mathcal{L}(\text{det-mon-nf-R}) & = & \mathcal{L}(\text{det-mon-nf-RWW})
\end{array}
$$

Figure 3.3: The taxonomy of deterministic monotone nonforgetting restarting automata

3.6 Shrinking Nonforgetting Restarting Automata

Shrinking automata are an extension to the normal length reducing restarting automata. Length reducing restarting automata reduce the length of the tape in each cycle, while shrinking restarting automata only have to reduce the weight of the tape, or the restarting configuration, in each cycle. For forgetting restarting automata the weight from cycle to cycle can be measured by looking at the rewrite instruction. If the rewrite instruction is weight reducing, then the weight of the tape, or of the restarting configuration, is reduced and vice versa.

For nonforgetting restarting automata this is not true. There are two different methods to define the notion of shrinking for nonforgetting restarting automata. The weight must be reduced in

every cycle but it is a difference whether the weight of the tape or the weight of the restarting configuration is considered. The reduction of the weight of the tape corresponds to a weight reduction of the rewrite step. This is the definition, which is given next. The other notion is studied in Subsection 3.7.

Definition 3.6.1. *A shrinking nonforgetting restarting automata $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ is a restarting automaton without the restriction to shorten the tape in every rewrite step. Instead there must exist a weight function $\varphi : \Gamma \to \mathbb{N}_+$, such that, for every rewrite step $u \to v$, $\varphi(u) > \varphi(v)$ holds. Here φ is extended to words by taking $\varphi(u_1 u_2 \dots u_n) = \varphi(u_1) + \varphi(u_2) + \dots + \varphi(u_n)$, $u_i \in \Gamma$ for all $i = 1, \dots, n$.*

This definition of shrinking is closely related to the notion of shrinking for forgetting automata. Also in [JO07] this kind of shrinking is considered for nonforgetting restarting automata. In this work we will always use this type of (tape) shrinking unless it is especially mentioned that the other kind of shrinking is used. Some results concerning shrinking nonforgetting restarting automata are already obtained in the previous sections.

We have seen in Section 2.7 that $\mathcal{L}(\mathsf{sh\text{-}RRWW}) = \mathrm{L}(\mathsf{nf\text{-}sh\text{-}RRWW})$ holds. Here we show that the same is true for deterministic RLWW-automata.

To establish this result we need Lemma 3.3.2 from page 42, which describes a normalform for deterministic nonforgetting restarting automata.

Theorem 3.6.2. $\mathcal{L}(\mathsf{det\text{-}sh\text{-}RLWW}) = \mathcal{L}(\mathsf{det\text{-}nf\text{-}sh\text{-}RLWW})$.

Proof. We simulate a shrinking nonforgetting restarting automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ with weight function σ by a shrinking forgetting restarting automaton $M' = (Q', \Sigma, \Gamma', ¢, \$, q_0', k+1, \delta)$ with the weight function σ'. This is done by writing the restarting states on the tape. To have at least one symbol on the right-hand side of each rewrite step, the window size is increased by one. The new tape alphabet consists of several copies of the old one $\Gamma' = \{A, A_{new,q}, A_{old,q} | A \in \Gamma, q \in Q\}$. We call the new symbols marked with a state and a label "new"/"old". The weight function is defined as follows:

$$\begin{aligned}
&\sigma'(A) = 3 * \sigma(A),\\
&\sigma'(A_{new,q}) = 3 * \sigma(A) + 2,\\
&\sigma'(A_{old,q}) = 3 * \sigma(A) + 1.
\end{aligned}$$

M' simulates M cycle by cycle. At the beginning of each cycle, M' scans the whole tape for occurrences of marked symbols, and then returns to the left border marker ¢. M' distinguishes four cases: no occurrence of a marked symbol, one marked symbol with the label "new", one marked symbol with the label "old", and two marked symbols, one with label "new" and one with label "old".

(i) *No marked symbol on the tape:*:
M' simulates a cycle of M starting from the initial state q_0. There is only one difference: when M' simulates a rewrite step $u \to v$ of M it marks the leftmost symbol in the new factor v with the next restarting state and the label "new". M' can do this because we can assume that M restarts immediately after the rewrite (see Lemma 3.3.2). The weight of the marked symbol is larger than the weight of the original symbol, but the weight of each rewrite step of M is reduced by at least one, thus in σ' the weight is reduced by at least three and therefore the overall weight is reduced in a rewrite step of M' as well. This case only happens at the start of a computation.

(ii) *One marked symbol with the label "new" on the tape:*:
The symbol of the form $A_{new,q}$ is replaced by $A_{old,q}$.

(ii) *One marked symbol with the label "old" on the tape:*:
M' simulates M starting its simulation in the restarting state q, if $A_{old,q}$ is the marked symbol on the tape. Again the leftmost symbol of the new factor v is marked with the new restarting state and the label "new". If the rewrite occurs in the place of the marked symbol $A_{old,q}$, then this symbol is rewritten into A.

(iv) *Two marked symbols on the tape:*:
If two marked symbols are on the tape, then one must have the marker "old" and one the marker "new". If this is the case, then M' deletes the label "old", that means the symbol $A_{old,q}$ is replaced by A.

The automaton M' works as follows:

At the start of each cycle M' scans the tape completely for occurrences of marked symbols. Only in the initial configuration there is none. In this case M' performs a cycle $(q_0, w_0) \vdash_M^c (q, w_1)$ as described in (i). After this cycle, there is one marked symbol $A_{new,q}$ on the tape. This symbol is replaced by the corresponding symbol $A_{old,q}$ in the next cycle as described in (ii).

In the next cycle M' sees the marked symbol $A_{old,q}$ on the tape. It then moves to the left border marker and starts simulating a cycle of M starting in the restarting state q, say $(q, w_1) \vdash_M^c (p, w_2)$. The new restarting state p is encoded in the leftmost symbol of the rewrite, instead of B the symbol $B_{new,p}$ is written. If there are now two marked symbols on the tape, the symbol with the label "old", $A_{old,q}$, is replaced by the symbol A in the next cycle. Now there is only the marked symbol $B_{new,p}$ on the tape, this is replaced by $B_{old,p}$. This sequence of two or three cycles iterates while simulating M.

As each cycle of M is simulated by a sequence of two or three cycles of M', it follows that $L(M) = L(M')$ holds. □

The same method of marking symbols can be used to simulate restarting automata that are allowed to perform more than one rewrite per cycle.

Theorem 3.6.3. *For every shrinking* det-nf-RLWW*-automaton with c rewrites per cycle M, there exists a shrinking* det-nf-RLWW*-automaton M' such that $L(M) = L(M')$ holds. If M is forgetting, then so is M'.*

Proof. We simulate a shrinking (nonforgetting) restarting automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$, with c rewrites per cycle and weight function σ by a shrinking (nonforgetting) restarting automaton $M' = (Q', \Sigma, \Gamma', ¢, \$, q_0', k+1, \delta)$ with the weight function σ'. This is done by writing the state on the tape. To have at least one symbol on the right-hand side of each rewrite step, the window size is increased by one. The new tape alphabet consists of several copies of the old one $\Gamma' = \{A, A_{q,i}, A_{remove} | A \in \Gamma, q \in Q, 1 \leq i \leq c\}$. We call the new symbols $A_{(q,i)}$ marked with a state q and a number i. The symbols A_{remove} are called "remove" symbols. The weight function is defined as follows:

$$\sigma'(A) = 3 * \sigma(A),$$
$$\sigma'(A_{q,i}) = 3 * \sigma(A) + 2,$$
$$\sigma'(A_{remove}) = 3 * \sigma(A) + 1.$$

M' simulates one cycle of M by $2c+1$ cycles. In each cycle it first scans the tape completely for occurrences of marked or remove symbols. If a marked symbol is found it remembers the state of the marked symbol with the highest number. In the first c cycles the c rewrites are performed, one in each cycle. While doing this c marked symbols $A_{q,i}$, $1 \leq i \leq c, q \in Q$, are written on the tape. The state in the marked symbol is the states in which M continues after the rewrite and the second index is the number of the rewrite. Assume M rewrites a configuration $\mathfrak{c}w_1xpuw_2\$$ into $\mathfrak{c}w_1xvqw_2\$$, $x \in \Gamma$ and $w_1, w_2, u, v \in \Gamma^*$ in its first rewrite. Then M' rewrites the configuration $\mathfrak{c}w_1pxuw_2\$$ into $\mathfrak{c}w_1x_{q,1}qw_2\$$ if $v = \varepsilon$ holds, or into a configuration where the leftmost symbol of v is marked with $q, 1$. After the rewrite M' restarts.

For all other rewrites M' moves to the marked symbol $A_{q,i}$ with the highest second index, enters state q and continues by simulating another rewrite of M. M' again restarts immediately after the rewrite.

When M' finds a marked symbol with the number c on the tape, then the leftmost of the marked symbols $A_{q,j}$ is replaced by A_{remove}. In the following $c-1$ cycles the other marked symbols $A^l_{q,i}$, $1 \leq i \leq c, i \neq j$, are replaced by A^l. In the last cycle the A_{remove} is replaced by A. If M is nonforgetting the restarting state is remembered in the state of M'.

Thus M' can simulate M cycle by cycle (or by $2c+1$ cycles) and therefore $L(M) = L(M')$ holds. □

Together Theorem 3.6.3 and Theorem 3.6.2 lead to the following Corollary.

Corollary 3.6.4. *The language class* $\mathfrak{L}$(det-sh-RLWW) *coincides with the class of languages that are recognized by shrinking deterministic nonforgetting* RLWW*-automata with multiple rewrites.*

Thus for deterministic sh-RLWW as well as for nondeterministic sh-RRWW-automata (see Section 2.7 on page 27), the property of being nonforgetting or the ability to perform many rewrites per cycles do not increase the expressive power. It is an open question whether these two language classes coincide. It is conjectured that they are not equal.

3.7 A Different Kind of Shrinking Nonforgetting Restarting Automata

Here we will study a different kind of shrinking. As it was mentioned in the beginning of the previous section it is possible to define shrinking nonforgetting restarting automata in two different ways.

1. each letter of the tape alphabet has a weight and the weight of the tape is reduced in each rewrite step. This coincides with the case that the weight of the tape is reduced in each cycle.
2. Each letter of the tape alphabet and each (restarting) state has a weight. It is required that the weight of the restarting configurations is reduced in each cycle.

Definition 3.7.1. *A state shrinking nonforgetting restarting automaton* $M = (Q, \Sigma, \Gamma, \mathfrak{c}, \$, q_0, k, \delta)$ *is a nonforgetting restarting automaton without the restriction to shorten the tape in every rewrite step. Instead there must exist a weight function* $\varphi : \Gamma \cup Q \to \mathbb{N}_+$ *such that, if* $(q, w) \vdash^c_M (p, w')$ *holds, then* $\varphi((q, w)) > \varphi((p, w'))$ *holds, too. Here* φ *is extended to configurations* $\varphi : (q, w) \to \mathbb{N}_+$.

The prefix shq− *denotes that the automaton is a state shrinking restarting automaton.*

It is easily seen that shrinking is a special case of state shrinking.

Proposition 3.7.2. $\mathcal{L}(\mathsf{sh\text{-}X}) \subseteq \mathcal{L}(\mathsf{shq\text{-}X})$ *for all types of restarting automata* X.

Proof. For all types of forgetting restarting automata these two types of shrinking are equivalent. Every nonforgetting shrinking restarting automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ with weight function φ can be simulated by a nonforgetting state shrinking restarting automaton $M'(Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ with weight function φ'. φ' is defined as follows: $\varphi'(q) = 1$ for all $q \in Q$ and $\varphi'(A) = \varphi(A)$ for all $A \in \Gamma$. □

These state shrinking automata have some interesting properties. For example deterministic state shrinking nonforgetting restarting automata do not need to read after the rewrite if they are non monotone.

Lemma 3.7.3. $\mathcal{L}(\mathsf{det\text{-}nf\text{-}shq\text{-}RRX}) = \mathcal{L}(\mathsf{det\text{-}nf\text{-}shq\text{-}RX})$ *for* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$.

Proof. $\mathcal{L}(\mathsf{det\text{-}nf\text{-}shq\text{-}RX}) \subseteq \mathcal{L}(\mathsf{det\text{-}nf\text{-}shq\text{-}RRX})$ is trivial. For the other direction let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be a deterministic state shrinking nonforgetting RRX-automaton. Every cycle $(q, w) \vdash_M^c (p, w')$ of M is simulated by a deterministic state shrinking nonforgetting RX-automaton $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k, \delta')$ in two cycles. In the first cycle it is checked which meta-instruction is applicable and the rewrite is performed at the end of the tape rewriting $\$ \to \$$. M' restarts in a restarting state with a smaller weight than q and the information which meta-instruction is applicable. In the second cycle this meta-instruction is simulated until M' performs the rewrite, where the simulation is ended and M' restarts immediately in the restarting state p. Thus in two cycles of M' $(q, w) \vdash_M^{2c} (p, w')$ is obtained as well.

□

This lemma does not hold for monotone restarting automata. Another interesting property is that state shrinking RRWW-automata are closed under intersection.

Theorem 3.7.4. *The language class* $\mathcal{L}(\mathsf{nf\text{-}shq\text{-}RRWW})$ *is closed under intersection.*

Proof. For two languages $L_1, L_2 \in \mathcal{L}(\mathsf{nf\text{-}shq\text{-}RRWW})$ and two nf-shq-RRWW-automata $M_1 = (Q_1, \Sigma, \Gamma_1, ¢, \$, q_{(0,1)}, k_1, \delta_1), M_2 = (Q_2, \Sigma, \Gamma_2, ¢, \$, q_{(0,2)}, k_2, \delta_2,$ with $L(M_1) = L_1$ and $L(M_2) = L_2$ we construct a nf-shq-RRWW-automata $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ with $L(M) = L_1 \cap L_2$.

M first copies the tape content so that it can check membership in L_1 on the first syllable and membership in L_2 on the second. If both checks are positive, then M accepts.

In detail: Let φ_1 be the weight function of M_1 and φ_2 be the weight function of M_2. Then M uses the weight function $\varphi(a) = \varphi_1(a) + \varphi_2(a) + 2$ for all $a \in \Sigma$, $\varphi(A) = \varphi_i(A)$ for $A \in \Gamma_i \setminus \Sigma$ with $i \in \{1, 2\}$. $\Gamma = \Sigma \cup \Sigma^1 \cup \Sigma^2 \cup (\Gamma_1 \setminus \Sigma) \cup (\Gamma_2 \setminus \Sigma) \cup \{\#\}$ where Σ^1 and Σ^2 are copies of Σ with $\varphi(a^i) = \varphi_i(a)$ for $a^i \in \Sigma^i$. $\varphi(\#) = 1$ and for the initial state $\varphi(q_0) = \varphi_1(q_{(0,1)}) + \varphi_2(q_{(0,2)}) + 1$.

M proceeds as follows. The first letter on the tape a_1 is rewritten into a_1^1. This rewrite reduces the weight by $\varphi(a_1) - \varphi(a_1^1) = \varphi(a_1^2) + 2$. $\varphi(a_1^2) + 1$ of this weight is "stored" in the restarting state such that the new restarting state has weight $\varphi_1(q_{(0,1)}) + \varphi_2(q_{(0,2)}) + \varphi(a_1^2) + 2$ and the weight of the restarting configuration is reduced by 1. In the next cycle $\#a_1^2$ is written at the end of the tape. The weight of the new restarting state is $\varphi_1(q_{(0,1)}) + \varphi_2(q_{(0,2)})$. In the next two cycles the second letter a_2 is first rewritten into a_2^1 and then written at the end of the tape as a_2^2. In each cycle the weight is reduced by one and the weight of the restarting state is again $\varphi_1(q_{(0,1)}) + \varphi_2(q_{(0,2)})$. This is repeated until all of the initial tape contents $¢w\$$ is rewritten into $¢w^1\#w^2\$$. Now M simulates

M_1 on the first syllable w_1. Observe that $\varphi(w^1) = \varphi_1(w)$ holds and that the current restarting state has weight $\varphi_1(q_{(0,1)}) + \varphi_2(q_{(0,2)})$. Thus if M_1 accepts the input w then the remaining weight in the restarting state is at least $\varphi_2(q_{(0,2)})$. If M_1 accepts the input, then M simulates M_2 on the second syllable w^2. If this simulation is also accepting, then M accepts. It follows that $L(M) = L_1 \cap L_2$ holds. □

There are still many open questions about state shrinking restarting automata. The most important is perhaps whether $\mathcal{L}(\mathsf{shq\text{-}nf\text{-}RRWW}) = \mathcal{L}(\mathsf{sh\text{-}RRWW}) = \mathcal{L}(\mathsf{FC})$ holds or not.

3.8 Correctness Preserving Restarting Automata

In this and the following section the results about correctness preserving restarting automata from Section 2.8 are extended to nonforgetting restarting automata. For nonforgetting restarting automata the definition of (strong) correctness preservation is different from the one for forgetting restarting automata. Here promising configurations are considered instead of the complete language. So we restate the definitions about promising configurations and strongly correctness preservation from the beginning of this chapter.

Definition 3.8.1. *A restarting configuration $q¢\omega\$$ is called an* accepting configuration*, if $q¢\omega\$ \vdash^c \mathsf{ACCEPT}$ holds.*

A restarting configuration $q¢\omega\$$ is called a promising configuration*, if there exists a computation $q¢\omega\$ \vdash^{c^*} p¢\omega'\$$ and $p¢\omega'\$$ is an accepting configuration.*

The set of all promising configurations of M is called $P_C(M)$.

Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta, Q_R)$ be a nf-RLWW*-automaton. M is called (strongly) correctness preserving, if every successor restarting configuration of a promising configuration is again a promising configuration.*

Remember that, if a nonforgetting restarting automaton $M = (Q, \Sigma, \Gamma, ¢\$, q_0, k, \delta)$ has an accepting computation starting from the restarting configuration $q¢w\$$ where $w \in \Sigma^*(\in \Gamma^*)$, it does not follow necessarily that $w \in L(M)(\in L_C(M))$ holds. It follows only for a promising configuration of the form $q_0¢w\$$ with $w \in \Sigma^*(\in \Gamma^*)$ that $w \in L(M)(\in L_C(M))$ holds.

For forgetting restarting automata there is a separation language between deterministic and correctness preserving RR-automata. For nonforgetting it is open whether this inclusion is proper. Later in this section we show that this question is related to another open problem about deterministic nonforgetting restarting automata.

For nf-R-, nf-RW-, and nf-RWW-automata we obtain the following result relating nondeterministic automata that are strongly correctness preserving to deterministic automata of the same type.

Theorem 3.8.2. *For any $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$, if $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ is a correctness preserving* nf-RX*-automaton, then there exists a deterministic* nf-RX*-automaton M' satisfying $P_C(M') = P_C(M)$ and $L(M') = L(M)$.*

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be a correctness preserving nf-RX-automaton. We describe a deterministic nf-RX-automaton $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k, \delta')$ such that $P_C(M') = P_C(M)$ holds. The automaton M' acts in the same way as M, but when scanning its tape from left to right it applies the first rewrite instruction that becomes applicable. In case several such instructions exist, the tie is broken based on a linear ordering of the rewrite instructions. Then M' is obviously a deterministic nf-RX-automaton.

As M' has the same Accept instructions as M, and as all rewrite steps that M' can execute are also valid rewrite steps of M, we see that $P_C(M') \subseteq P_C(M)$ holds. So it remains to verify the converse inclusion.

Let $w \in \Gamma^*$ and $q \in \mathsf{Q}$. Assume that w is accepted by M when it starts in the restarting state q, that is, $q¢w\$ \in P_C(M)$. If starting from the restarting configuration $q¢w\$$, M can execute a rewrite operation, then M' will definitely execute a rewrite operation in this situation, that is, we obtain a cycle of the form

$$q¢w\$ \vdash_{M'}^c q'¢w'\$$$

for some $w' \in \Gamma^*$ and $q' \in Q$. From the definition of M' it follows that we also have the cycle $w \vdash_M^c w'$. However, as M is correctness preserving, this means that $w' \in P_C(M)$ holds. By induction it now follows that $q¢w\$$ is a promising configuration of M', that is, $P_C(M') = P_C(M)$. As M' and M have the same input alphabet and initial state, we obtain $L(M') = L(M)$, too.

□

In the following we will extend this result to nf-RLWW-automata. In fact, we consider two generalizations of these automata. First of all we consider shrinking nf-RLWW-automata. In fact, it is easily seen from the proof of Theorem 3.8.2 that this theorem even holds for shrinking nf-RWW- and shrinking nf-RW-automata.

Secondly, we admit (shrinking) restarting automata that are allowed to perform up to c rewrite operations per cycle for some constant $c \geq 1$. Next we recall the following useful technical result which is an extension of Theorem 3.1.4.

Theorem 3.8.3. *For any* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$, *if* $M_L = (Q_L, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_L)$ *is a (shrinking)* nf-RLX*-automaton that executes up to* $c \geq 1$ *rewrite steps per cycle, then there exists a (shrinking)* nf-RRX*-automaton* $M_R = (Q_R, \Sigma, \Gamma, ¢, \$, q_0, k, \delta_R)$ *that also executes up to* c *rewrite steps per cycle such that, for all* $u, v \in \Gamma^*$, $q¢u\$ \vdash_{M_L}^c q'¢v\$$ *if and only if* $q¢u\$ \vdash_{M_R}^c q'¢v\$$. *In particular,* $P_C(M_L) = P_C(M_R)$, $L_C(M_L) = L_C(M_R)$ *and* $L(M_L) = L(M_R)$ *hold.*

Hence, correctness preserving nf-RRX-automata are as expressive as correctness preserving nf-RLX-automata. The following result now relates deterministic nf-RLX-automata to correctness preserving nf-RLX- (and therewith nf-RRX-) automata.

Theorem 3.8.4. *For any* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$, *if* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ *is a correctness preserving (shrinking)* nf-RLX*-automaton that performs up to* $c \geq 1$ *rewrite steps per cycle, then there exists a (shrinking) deterministic* nf-RLX*-automaton* M' *performing up to* c *rewrite steps per cycle such that* $P_C(M') = P_C(M)$ *and* $L(M') = L(M)$ *hold.*

The proof is an adaptation of the proof of Theorem 2.8.5. Theorem 3.8.3 enables us to describe any (shrinking) nf-RLX-automaton that executes up to $c \geq 1$ rewrite steps per cycle by accepting meta-instructions and by generalized rewriting meta-instructions of the form

$$I = (q, ¢ \cdot E_0, u_1 \to v_1, E_1, u_2 \to v_2, \ldots, E_{s-1}, u_s \to v_s, E_s \cdot \$, p),$$

where $1 \leq s \leq c$, $E_0, E_1, \ldots, E_s$ are regular languages, $u_1 \to v_1, \ldots, u_s \to v_s$ are the rewrite steps executed by I, and $p, q \in Q$. This meta-instruction can be applied to a restarting configuration $q¢w\$$, where w has the form $w = x_0u_1x_1u_2 \cdots x_{s-1}u_sx_s$ satisfying $x_i \in E_i$ for all $i = 0, \ldots, s$, and it yields the restarting configuration $p¢x_0v_1x_1v_2 \cdots x_{s-1}v_sx_s\$$. In the proof of Theorem 2.8.5 the simulation is done cycle by cycle. Thus, it can easily be adapted for nonforgetting restarting automata if promising configurations instead of words $w \in L_C(M)$ are considered.

It is still open whether Theorem 3.8.4 carries over to nf-RRX-automata or not. From Theorems 3.8.3 and 3.8.4 it follows that $\mathcal{L}$(det-nf-RLX) equals the class of languages accepted by correctness preserving nf-RRX-automata. Thus the following open questions are related. Here the prefix cp- stands for correctness preserving restarting automata.

Corollary 3.8.5. *The following two open questions are equivalent:*

- *Does* $\mathcal{L}$(det-nf-RRX) = $\mathcal{L}$(cp-nf-RRX) *hold?*
- *Does* $\mathcal{L}$(det-nf-RRX) = $\mathcal{L}$(det-nf-RLX) *hold (see Section 3.3)?*

3.9 The Error-Detection Distance

Here we introduce the notion of error-detection distance for nonforgetting restarting automata. It is a generalization of the correctness preserving property.

Definition 3.9.1. *Let* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ *be a* nf-RLWW*-automaton, and let* i *be a nonnegative integer. We say that* M *has* error-detection distance i*, if, for all restarting configurations* $q¢w\$ \in P_C(M)$ *and all partial computations* $q¢w\$ \vdash_M^c q_1¢w_1\$ \vdash_M^c \cdots \vdash_M^c q_m¢w_m\$$*, if* $q_1¢w_1\$ \notin P_C(M)$*, then* $m \leq i$*. That is, if the first cycle* $q¢w\$ \vdash_M^c q_1¢w_1\$$ *transforms the promising configuration* $q\,¢w\,\$$ *into a restarting configuration* $q_1¢w_1\$ \notin P_C(M)$*, which means that* M *has made a mistake, then starting from the restarting configuration* $q_1¢w_1\$$*,* M *can execute at most* $i-1$ *more cycles before it halts and rejects, that is, before it detects its error.*

Obviously, the property of being correctness preserving corresponds to error-detection distance 0.

3.9.1 Bounded Error-Detection Distance

Here it is shown that, for almost all types X of nf-RLWW-automata, an X-automaton of bounded error-detection distance is equivalent in expressive power to an X-automaton that is correctness preserving. Unfortunately up to now it is still just a conjecture that the result for R-, RW-, and RWW-automata can also be shown for nonforgetting restarting automata.

Conjecture 3.9.2. *For any* X $\in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$*, if* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ *is a* nf-RX*-automaton that has error-detection distance* $i \geq 1$*, then there exists a* nf-RX*-automaton* $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k', \delta')$ *with error-detection distance* 0 *such that* $P_C(M') = P_C(M)$ *and* $L(M') = L(M)$*.*

The corresponding result for RRX- and RLX-automata carries over to nf-RRX- and nf-RLX-automata.

Theorem 3.9.3. *For any* X $\in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$*, if* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ *is a* nf-RRX- *or* nf-RLX*-automaton that has error-detection distance* $i \geq 1$*, then there exists a* nf-RRX- *or* nf-RLX*-automaton* $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k, \delta')$ *with error-detection distance at most* $i-1$ *such that* $P_C(M') = P_C(M)$ *and* $L(M') = L(M)$*.*

Proof. If M is a nf-RRX-automaton, then with Theorem 3.8.3 there exists a nf-RLX-automaton $\hat{M}$ that accepts the same language as M, and it is easily seen that it has the same error detection distance as M. $\hat{M}$ is then simulated by a nf-RLX-automaton $\hat{M}'$. By using Theorem 3.8.3 again there exists a nf-RRX-automaton M' that accepts the same language and has the same error detection distance as $\hat{M}'$. Thus, only nf-RLX-automata need to be considered in this proof.

Let $q\¢w\$ \vdash_M^C q_1\¢w_1\$$ be a cycle of M. M' simulates this cycle of M but instead of a restart at the end it moves to the left end of the tape, enters a state q_1', and tries to apply a cycle to $q_1\¢w_1\$$. From Lemma 3.3.2 we can assume without loss of generality that M restarts immediately after the rewrite, and therefore M' can simulate a second cycle. If M would perform a rewrite, then M' simply restarts in the restarting state q_1, if M is unable to perform another cycle, then M' halts and rejects, and if M accepts, then M' accepts as well.

Thus, while simulating a cycle of M, the automaton M' can already verify whether all computations following the current cycle will be tails, or whether another restart step follows. In the former case M' can shorten the simulation by checking the results of all these tail computations without performing the restart step, that is, M' shortens the computation of M being simulated by one cycle. In this way an automaton with error-detection distance $i-1$ is obtained from M.

□

By applying Theorem 3.9.3 repeatedly we obtain the following consequence.

Corollary 3.9.4. *For any* $\mathsf{X} \in \{\mathsf{nf\text{-}RR}, \mathsf{nf\text{-}RL}, \mathsf{nf\text{-}RRW}, \mathsf{nf\text{-}RLW}, \mathsf{nf\text{-}RRWW}, \mathsf{nf\text{-}RLWW}\}$*, if M is an* X*-automaton that has bounded error-detection distance, then there exists a correctness preserving* X*-automaton M' such that $P_C(M') = P_C(M)$, $L_C(M') = L_C(M)$, and $L(M') = L(M)$ hold.*

Thus, for all these types of restarting automata, bounded error-detection distance limits the expressive power to that of correctness preserving automata.

By combining Theorem 3.8.4 with Corollary 3.9.4 we obtain the following result.

Corollary 3.9.5. *For any* $\mathsf{X} \in \{\mathsf{WW}, \mathsf{W}, \varepsilon\}$*, if M is a* $\mathsf{nf\text{-}RRX}$*- or a* $\mathsf{nf\text{-}RLX}$*-automaton that has bounded error-detection distance, then there exists a deterministic* $\mathsf{nf\text{-}RLX}$*-automaton M' satisfying $P_C(M') = P_C(M)$, $L_C(M') = L_C(M)$, and $L(M') = L(M)$.*

Corollary 3.9.5 shows that for the various types of $\mathsf{nf\text{-}RL}$-automata, the deterministic variant is as expressive as the nondeterministic variant with bounded error-detection distance. This clearly shows that for these types of restarting automata, the nondeterministic variants are more expressive than the corresponding deterministic variants only if they have unbounded error-detection distance. Thus, it is the ability of a nondeterministic $\mathsf{nf\text{-}RLWW}$-automaton to make an error that is only detected an unbounded number of cycles later that really contributes to the expressive power of these automata. On the negative side this means that the error-detection distance is not a useful complexity measure for nonforgetting restarting automata, as there are essentially only two cases: error-detection distance 0 (that is, strongly correctness preserving automata) and unbounded error-detection distance.

3.10 Cyclic Restarting Automata

Definition 3.10.1. *A nonforgetting restarting automaton is called* cyclic*, if it always reaches its initial state q_0 after finitely many restart steps.*

A cyclic nf-restarting automaton is called l-cyclic*, if it always reaches the initial state q_0 after exactly l restart steps. It is called* $l_{\leq}$*-cyclic, if it always reaches the initial state q_0 after at most l restart steps.*

The meta-cycle from q_0 to q_0 of an l-cyclic nf-restarting automaton M that contains l cycles of M is called an l-cycle of M.

A cyclic restarting automaton is always a nonforgetting automaton and therefore is called only cyclic restarting automaton to simplify the notation.

Lemma 3.10.2.

a) Every nondeterministic l-cyclic restarting automaton M of type X can be simulated by a nondeterministic restarting automaton M' of the same type with exactly l rewrites per cycle.

b) Every nondeterministic $l_{\leq}$-cyclic restarting automaton M type X can be simulated by a nondeterministic restarting automaton M' of the same type with at most l rewrites per cycle.

Here $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$ *holds.*

Proof. In the proof only $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RWW}, \mathsf{RRWW}\}$ are considered, because from Theorem 3.8.3 the other cases follow. We start with proving the first part. The second follows directly from this proof.

The idea of the proof is that the l-cyclic restarting automaton M cannot remember things from one l-cycle to the next. Because of this we can simulate an l-cycle by one cycle of M' with l rewrite operations. For $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RWW}, \mathsf{RRWW}\}$ let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta, Q_R)$ be an l-cyclic restarting automaton of type X, then the X-automaton $M' = (Q', \Sigma, \Gamma, ¢, \$, q_0, k, \delta')$ with l rewrites per cycle can recognize the same language as M.

First of all, M' guesses the l rewrite steps of M of one l-cycle. If the rewrites occur all in different places, M' can perform these l rewrites without changes. It only has to do them from left to right and remember the original order to check the regular expressions, which have to be checked all at once.

Let $I_1, I_2, ...I_l$, be the sequence of meta-instructions of M that are used in the current l cycles. Then M' does the l rewrites from these meta-instructions from left to right in one single cycle. M' simulates all regular expressions. If a rewrite step $I_i : u_i \to v_i$ occurs, then the regular expressions of the meta-instructions $I_1, ..., I_{i-1}$ use u_i and the meta-instructions $I_{i+1}, ..., I_l$ use v_i to simulate the original order.

If more than one rewrite step occurs at the same place on the tape, then M' puts all rewrites in one area together and then performs these rewrites sequentially from left to right.

In detail: Let m be the number of rewrites that are performed on a subword x of length $k * m$ in one l-cycle. Then M' first combines these m rewrites. In the l-cycle

$$q_0 ¢ \alpha x \beta \$ \vdash_M^{c^l} \alpha_1 y \beta_1$$

x is rewritten into y in m not necessarily successive cycles. Thus $|x| \geq |y| + m$, therefore $x = a_1 a_2 \dots a_{k*m}$ can be rewritten into $y = b_1 b_2 \dots b_n$ in m rewrite steps. In the first cycle $a_1 a_2 \dots a_k$ is rewritten into $b_1 b_2 \dots b_{k-1}$, and in the ith rewrite step $a_{k*i+1} a_{k*i+2} \dots a_{k*i+k}$ is rewritten into $b_{(k-1)*i+1} b_{(k-1)*i+2} \dots b_{(k-1)*i+(k-1)}$. Here $b_i = \varepsilon$ for $i > n$, if X is not an R- or RR-automaton. If X is a R- or RR-automaton, then $a_{k*i+1} a_{k*i+2} \dots a_{k*i+k}$ is rewritten into the remaining scattered subword of $a_{k*i+1} a_{k*i+2} \dots a_{k*i+k}$ in y.

Thus these m overlapping rewrites can be performed sequentially in a subword of length $k * m$. As there are at most l rewrites it is possible to perform overlapping rewrites in a subword of at most length $k * l$.

M' must accept or reject all words up to length $k * l$ directly in one step, to ensure that always a subword of length $k * m$ with $m \leq l$ exists. In every cycle M' makes as many rewrites as M

does in its meta-cycle from q_0 to q_0. So if M is a $l_{\leq}$-cyclic (R)RW(W) automaton then M' is a (R)RW(W) automaton with at most l rewrites per cycle.

□

This proof heavily depends on the nondeterminism of M'. Therefore there is no deterministic variant of this lemma.

The converse is not true, there exist languages that can be recognized by a restarting automaton with l rewrites per cycle, but are not recognizable by an l-cyclic restarting automaton.

Lemma 3.10.3. *The language class of restarting automata with l rewrites per cycle is not included in the class of l-cyclic restarting automata. The language $L_{Dyck^2} = D_2$ shuffle D_2 is easily recognized by an* R*-automaton with 2 rewrites per cycle, but it cannot be recognized by a 2-cyclic R automaton.*

Proof. Let $L_1 \subset \{a, \bar{a}, b, \bar{b}\}^*$ and $L_2 \subset \{c, \bar{c}, d, \bar{d}\}$ be Dyck languages over two symbols and let L_{Dyck^2} be the shuffle of L_1 with L_2. This language is recognized by the following R-automaton M with two rewrites per cycle.

$$
\begin{aligned}
&(1)(\text¢\{a,b,c,d\}^*,\ a \to \varepsilon,\ \{c,d\}^*,\ \bar{a} \to \varepsilon)\\
&(2)(\text¢\{a,b,c,d\}^*,\ b \to \varepsilon,\ \{c,d\}^*,\ \bar{b} \to \varepsilon)\\
&(3)(\text¢\{a,b,c,d\}^*,\ c \to \varepsilon,\ \{a,b\}^*,\ \bar{c} \to \varepsilon)\\
&(4)(\text¢\{a,b,c,d\}^*,\ d \to \varepsilon,\ \{a,b\}^*,\ \bar{d} \to \varepsilon)\\
&(5)(\text¢\$, \mathsf{ACCEPT})
\end{aligned}
$$

M guesses the correct opening bracket to the first closing bracket, deletes it and then it checks, if its guess was correct, that means that the inner part between these brackets contains no brackets from the same Dyck-language and that the first closing bracket corresponds to the deleted opening bracket. Then it deletes the closing bracket. M accepts the empty word.

But it is impossible for a 2-cyclic R-automaton M' to recognize this language. This is proved as follows:

There exist words that can only be recognized by deleting only one type of opening and closing bracket in one 2-cycle. The word ω is one of these:

$$\omega = \alpha_1 \dots \alpha_{N_1} \bar{a}^{N_1'} \alpha_{N_1+1} \dots \alpha_{N_2} \bar{a}^{N_2'} \dots \alpha_{N_E} \bar{a}^{N_E'} \beta_{N_E} \dots \beta_1 \in L$$

$\alpha_i = b^{m_i'} d^{m_i'''} a^{m_i} c^{m_i''}$ consists of opening brackets, $\beta_i = \bar{a}^{n_i} \bar{b}^{m_i'} \bar{c}^{m_i''} \bar{d}^{m_i'''}$ consists of closing brackets and $n_i = m_i$ for all $1 \leq i \leq N_E$ except for $i \in \{N_1 \dots N_E\}$ where $n_i = m_i - N_i'$. All n_i and m_i must be large.

If an automaton M' deletes two types of brackets in one cycle, then there are only four possibilities: bd, da, ac and cb, because of the structure of the first part of the word. bd and ac are not possible, as their closing brackets are not sufficiently close together at the end of the tape. $\bar{d}\bar{a}$ is at the end of the Tape together too, but in different blocks, so that the deleting of both in one 2-cycle is against the Error preserving Property. $\bar{b}\bar{c}$ is at the end in one block, but at the beginning in two different blocks, so it is also impossible to delete these two brackets.

With that we can assume that M' deletes only one type of brackets in an accepting computation in one 2-cycle. There are four cases:

- **Deleting a's**
  It is possible to find the first a, but it is impossible to find the last $\bar{a}$, which corresponds to the first a. It is the same with the second third and so on. It is also possible to find the first $\bar{a}$, but the last a before this $\bar{a}$ cannot be found, and it is the same for all other $\bar{a}$ in the first part of the word, because the distance between two $\bar{a}$ is too large. With the same reason it is impossible to delete the first $\bar{a}$ in the second part of the word, because it is indistinguishable from the $\bar{a}$s in the first part.

- **Deleting b's**
  Again the first b can be found, but not the last $\bar{b}$. The first $\bar{b}$ can be found, but it is impossible to find the associated b. There are no other parts of ω where b's and associated $\bar{b}$s can be detected.

- **Deleting c's**
  With the c's it is more or less the same as with the b's, the beginning and end of the word does not help and the border between the first and the second part looks the same as the $\bar{a}$ in the first part.

- **Deleting d's**
  The first d and the last $\bar{d}$ can both be found and therefore there is a possible 2-cycle for M'. But there are no other places, where associated brackets of type d can be deleted with the same reasons than in case b.

Thus, except for the d's at the border, it is impossible to find associated brackets of any type in ω.

We now construct a new word ω' that contains many words of the same type as ω:

$$\omega' = (\alpha_1 \dots \alpha_{N_1} \bar{a}^{N_1'} \alpha_{N_1+1} \dots \alpha_{N_2} \bar{a}^{N_2'} \dots \alpha_{N_E} \bar{a}^{N_E'} \beta_{N_E} T_{N_E} \beta_{N_E-1} T_{N_E-1} \dots T_1 \beta_1)^2 \in L.$$

Here the words T_j have the same structure as ω:

$$T_j = \alpha_1 \dots \alpha_{N_1} \bar{a}^{N_1'} \alpha_{N_1+1} \dots \alpha_{N_2} \bar{a}^{N_2'} \dots \alpha_{N_E} \bar{a}^{N_E'} \beta_{N_E} \dots \beta_1 \in L.$$

Where $\alpha_i = b^{m_i'} d^{m_i'''} a^{m_i} c^{m_i''}$ consists of opening brackets, $\beta_i = \bar{a}^{n_i} \bar{b}^{m_i'} \bar{c}^{m_i''} \bar{d}^{m_i'''}$ consists of closing brackets and $n_i = m_i$ for all $1 \leq i \leq N_E$ except for $i \in \{N_1 \dots N_E\}$ where $n_i = m_i - N_i'$. All n_i and m_i must be large.

In ω' it is no longer possible to detect the borders of the words T_j of type ω, because after the end of T_j another block of the same structure as the end of T_j begins. Therefore it is impossible to delete d's at the borders of one of the T_js. ω' contains two main parts and the border is not detectable for an R-automaton because of the structure of the words T_j. The prefix of each T_j equals the prefix of the second main part of ω'. That is why the deletion of the brackets from type d at the borders of ω does not work for ω'.

And the T_j does not help by finding associated brackets because they appear in the part of the closing brackets and except for the last blocks of closing brackets it is the problem to find the associated opening ones.

We have seen that there is no possibility to find two associated brackets of any type in ω' by an 2-cyclic R-automaton, therefore the language L_{Dyck^2} cannot be recognized by an 2-cyclic R-automaton.

□

In contrast to this, some results about shrinking restarting automata were obtained in previous sections. We know from the sections 2.7 and 3.6 that many types of shrinking restarting automata with c rewrites per cycle are equal in expressive power to shrinking forgetting automata.

Chapter 4

CD Systems of Restarting Automata

In this chapter we consider another extension of restarting automata: cooperating distributed (CD-) systems of restarting automata. They consist of a finite collection of restarting automata that work together to analyze a sentence. The cooperation is done in a similar way as for CD-grammar systems (see e.g. [CVD90, DP97]). One automaton is chosen to start the computation and this automaton performs some cycles on the given input. A mode of operation controls how long the chosen automaton works on the input. A successor relation determines which automaton takes over, after the active automaton has finished its part of the computation.

An input is accepted, if one automaton accepts the input and an input is rejected, if an automaton is unable to perform the number of cycles that is required by the mode of operation.

In this chapter the expressive power of these systems is studied and they are compared to nonforgetting restarting automata. Also different types of determinism for CD-systems are introduced and compared.

4.1 Definitions and Examples

Definition 4.1.1. *A cooperating distributed system of* RLWW*-automata,* CD-RLWW*-system for short, consists of*

- *a finite collection* $\mathcal{M} := ((M_i, \sigma_i)_{i \in I}, I_0)$ *of* RLWW*-automata* $M_i = (Q_i, \Sigma, \Gamma, ¢, \$, q_0^{(i)}, k, \delta_i)$ *$(i \in I)$,*
- successor relations $\sigma_i \subseteq I$ *$(i \in I)$ and*
- *a subset* $I_0 \subseteq I$ *of* initial indices.

Here it is required that $Q_i \cap Q_j = \emptyset$ *for all* $i, j \in I$, $i \neq j$, *that* $I_0 \neq \emptyset$, *that* $\sigma_i \neq \emptyset$ *for all* $i \in I$ *and that* $i \notin \sigma_i$. *Further, let* m *be one of the following* modes of operation*, where* $j \in \mathbb{N}$*:*

$= j$	:	*execute exactly j cycles;*
$\leq j$	:	*execute up to j cycles;*
$\geq j$	:	*execute at least j cycles;*
t	:	*continue until no more cycle can be executed.*

Note: In ([MO07a, MO07c]) we defined CD-systems of restarting automata such that each component automaton M_i had its own tape alphabet Γ_i. However, all example systems given had a single tape alphabet only. Having different tape alphabets looks like a generalization, but in fact it is not. With $\Gamma = \bigcup \Gamma_i$ and δ_i designed accordingly it is always possible to construct a CD-system of restarting automata with one tape alphabet for all components that accepts the same language. On the other hand, if each component is allowed to have its own tape alphabet, there must be a mechanism that checks that, when another component M_i becomes active, only symbols from Γ_i are on the tape. Only if we consider the number of auxiliary symbols for the components as a complexity measure, it may make a difference whether each automaton has its own tape alphabet or not.

One could also think about systems, where only some of the components use auxiliary symbols. Then the tape alphabets are different as well.

Definition 4.1.2. *A configuration of a* CD*-system* $\mathcal{M}$ *is given by a tuple* $(M_i, \alpha \cdot q\beta)$*, where* M_i *is a component of* $\mathcal{M}$*,* q *is a state of* M_i*, and either* $\alpha = ¢w_1, \beta = w_2\$$ *with* $w_1, w_2 \in \Gamma^*$ *or* $\alpha = \varepsilon, \beta = ¢w\$$ *with* $w \in \Gamma^*$*.*

A configuration $(M_i, q_0^{(i)}¢w\$)$ *with* $w \in \Gamma^*$ *and* $q_0^{(i)}$ *the initial state of* M_i *is called a* restarting configuration*. If* $\omega \in \Sigma^*$ *and* $i \in I_0$ *hold, it is called an* initial configuration*.*

For restarting configurations it would not be necessary to give the initial state of the active component explicitly. A cycle of a CD-RLWW-system $\mathcal{M}$ from one restarting configuration to the next can be seen as a relation over tuples of component automata and words. $(M_i, w_1) \vdash^c_{\mathcal{M}} (M_j, w_2)$ means that the component M_i transforms w_1 in one cycle into w_2 and that the component M_j becomes active after this cycle.

A shorter form to describe a cycle is $w_1 \vdash^c_{M_i} w_2$. But here the active component after the cycle is not stated. However for an accepting sequence of cycles, the following sequence $w_1 \vdash^c_{M_1} w_2 \vdash^c_{M_2} w_3 \ldots \vdash^c_{M_l} w_l \vdash^c_{M_{l+1}} \mathsf{ACCEPT}$ contains all relevant information and is succinct.

Let $x \in \Sigma^*$ be an input word, and let m be one of the above modes of operation. The computation of $\mathcal{M}$ in mode m on input x proceeds as follows: First an index $i_0 \in I_0$ is chosen nondeterministically. Then the RLWW-automaton M_{i_0} starts the computation with the initial configuration $q_0^{(i_0)}¢x\$$, executing one or more cycles according to mode m. Then an index $i_1 \in \sigma_{i_0}$ is chosen nondeterministically, and M_{i_1} continues the computation by executing one or more cycles according to mode m. After that an index $i_2 \in \sigma_{i_1}$ is chosen nondeterministically, and M_{i_2} continues the computation. The computation halts successfully, if, for some $l \geq 0$, the machine M_{i_l} accepts. Should at some stage the chosen machine M_{i_j} be unable to execute the required number of cycles, then the computation fails.

In mode t the chosen restarting automaton M_{i_j} continues with the computation until it either accepts, in which case $\mathcal{M}$ accepts, or until it can neither execute another cycle nor an accepting tail, in which case an automaton $M_{i_{j+1}}$ with $i_{j+1} \in \sigma_{i_j}$ takes over. Should at any time the chosen machine be unable to execute a single cycle or an accepting tail, then the computation of $\mathcal{M}$ fails. This ensures that each component performs at least one cycle (or an accepting tail).

If no component of a CD system $\mathcal{M}$ performs MVL-operations, then each component automaton can be described by meta-instructions.

Definition 4.1.3. *By* $L_{\mathsf{m}}(\mathcal{M})$ *we denote the language that the* CD-RLWW*-system* $\mathcal{M}$ *accepts in mode* m*. It consists of all words* $x \in \Sigma^*$ *that are accepted by* $\mathcal{M}$ *in mode* m *as described above. If* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, then a* CD-X*-system is a* CD-RLWW*-system for which all restarting automata are of type* X*. By* $\mathcal{L}_{\mathsf{m}}$(CD-X) *we will denote the class of languages that are accepted by* CD-X*-systems in mode* m*.*

The complete language of $\mathcal{M}$ *in mode* m *consists of all words* $x \in \Gamma^*$ *that are accepted by* $\mathcal{M}$ *in mode* m *as described above.*

Now we explain how a CD-system works in a small example that also shows the expressive power of CD-systems.

Example 4.1.4. *We describe a* CD-R*-system* $\mathcal{M} := ((M_1, \{2\}), (M_2, \{1\})), \{1\})$ *for the language* $L_{\text{copy}} := \{ w\#w \mid w \in \{a,b\}^* \}$. *Here* $(M_1, \{2\})$ *means that the component* M_1 *has the successor relation* $\sigma_1 = \{2\}$. $(M_2, \{1\})$ *means that the component* M_2 *has the successor relation* $\sigma_2 = \{1\}$ *and the set* $\{1\}$ *is the set of initial indices, that is, here the only initial component is* M_1. *The starting machine* M_1 *is described by the following two meta-instructions:*

$$\begin{array}{ll} (1.) & (\mathrm{¢} \cdot \{a,b\}^* \cdot c\# \cdot \{a,b\}^*, c\$ \to \$) \quad \textit{for } c \in \{a,b\}; \\ (2.) & (\mathrm{¢} \cdot \# \cdot \$, \mathsf{Accept}), \end{array}$$

and the machine M_2 *is given through the following meta-instruction:*

$$(3.) \quad (\mathrm{¢} \cdot \{a,b\}^*, c\# \to \#) \quad \textit{for } c \in \{a,b\}.$$

In mode $=1$ *the machines* M_1 *and* M_2 *alternate, with* M_1 *starting the computation.* M_1 *always deletes the last letter of the second factor, provided it coincides with the last letter of the first factor, and* M_2 *simply deletes the last letter of the first factor. It follows easily that* $L_{=1}(\mathcal{M}) = L_{\text{copy}}$.

In the previous chapter new error and correctness preserving properties were introduced. These properties also hold for CD-systems of restarting automata.

Definition 4.1.5. *A restarting configuration* $(M_i, q\mathrm{¢}w\$)$ *is called an* accepting configuration *for mode* m*, if it is accepted in a tail computation.*

A restarting configuration $(M_i, q\mathrm{¢}w\$)$ *is called a* promising configuration *for mode* m*, if there exists a computation of* $\mathcal{M}$ *in mode* m*, which leads to an accepting configuration, when starting from this configuration.*

An initial promising configuration is a promising configuration $(M_i, q\mathrm{¢}w\$)$ *with* $i \in I_0$ *and* $w \in \Sigma^*$.

For each initial promising configuration $(M_i, q\mathrm{¢}w\$)$ of $\mathcal{M}$ in mode m, it follows that there exists an accepting computation for $w \in \Sigma^*$, that is, w is accepted by $\mathcal{M}$ in mode m and therefore belongs to $L_{\mathsf{m}}(\mathcal{M})$.

For each promising configuration $(M_i, q\mathrm{¢}w\$)$ of $\mathcal{M}$ in mode m with $i \in I_0$, it follows that there exists an accepting computation for $w \in \Gamma^*$, that is, w is accepted by $\mathcal{M}$ in mode m and therefore belongs to the complete language of $\mathcal{M}$ in mode m.

With this definition we can achieve the Error Preserving and the Correctness Preserving Property for CD-systems of restarting automata.

Lemma 4.1.6 (Error Preserving Property)**.**

Let $\mathcal{M} = ((M_i, \sigma_i)_{i \in I}, I_0)$ *be a* CD*-system of restarting automata, and let* u, v *be words over its tape alphabet* Γ. *If* $(M_i, q\mathrm{¢}u\$) \vdash_{\mathcal{M}}^{c} (M_j, p\mathrm{¢}v\$)$ *is part of a computation of* $\mathcal{M}$ *in mode* m *and* $(M_i, q\mathrm{¢}u\$)$ *is not a promising configuration of* $\mathcal{M}$ *in mode* m*, then* $(M_j, p\mathrm{¢}v\$)$ *is neither.*

Remark that for a word x that does not belong to the language of $\mathcal{M}$, there exists no computation of $\mathcal{M}$, which reaches a promising configuration starting from an initial configuration $(M_i, q_0^{(i)}\mathrm{¢}x\$)$. Thus this Error Preserving Property is an extension of the Error Preserving Property for restarting automata. There is also a strong Correctness Preserving Property for CD-systems.

Definition 4.1.7. *A* CD*-system of restarting automata is called (strongly) correctness preserving, if each successor restarting configuration of a promising configuration is again a promising configuration.*

Lemma 4.1.8 (Correctness Preserving Property)**.** *Let* $\mathcal{M} = ((M_i, \sigma_i)_{i \in I}, I_0)$ *be a* CD*-system of restarting automata. For every promising configuration of* $\mathcal{M}$ *in mode* m *it follows that either it is an accepting configuration of* $\mathcal{M}$*, or there exists a successor restarting configuration that is again a promising configuration of* $\mathcal{M}$ *in mode* m*.*

The strong Correctness Preserving Property does not hold for all kinds of determinism. Different notions of determinism are introduced in the section about deterministic CD-systems of restarting automata, and there the correctness preserving property for deterministic CD-systems of restarting automata is given.

These two lemmata allow the usage of the Error and the Correctness Preserving Properties for cooperating distributed systems of restarting automata.

4.2 Nonforgetting Restarting Automata versus CD-Systems of Restarting Automata

In this section CD-systems of restarting automata are compared to nonforgetting restarting automata. It is shown that nonforgetting restarting automata can be simulated by certain CD-systems of restarting automata and that almost all kinds of CD-systems can be simulated by nonforgetting restarting automata.

Theorem 4.2.1. *If* M *is a nonforgetting restarting automaton of type* X *for any type* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, then there exists a CD-system* $\mathcal{M}$ *of restarting automata of type* X *such that* $L_{=1}(\mathcal{M}) = L(M)$ *holds.*

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ be a non-forgetting restarting automaton of type X, where we assume that $Q = \{q_0, q_1, \ldots, q_m\}$. We define a simulating CD-system $\mathcal{M}$ of X-automata as follows.

Let $I := \{\, (i,j) \mid 0 \leq i, j \leq m \,\} \cup \{\, (i) \mid 0 \leq i \leq m \,\}$ be the set of indices, and let $I_0 := \{\, (0,j) \mid 0 \leq j \leq m \,\}$ be the set of initial indices.

For each pair of indices $i, j \in \{0, 1, \ldots, m\}$, $i \neq j$, we take a copy $M_{(i,j)}$ of M that simulates those computations of M that start with a restarting configuration of the form $q_i ¢ w \$$, and that either accept, reject, or perform a restart that takes M into state q_j. In addition, we introduce two X-machines $M_{(i,i)}$ and $M_{(i)}$ for each $i \in \{0, 1, \ldots, m\}$ that simulate those computations of M that start with a restarting configuration of the form $q_i ¢ w \$$, and that either accept, reject, or perform a restart that takes M back into state q_i. The reason for having both these machines, $M_{(i,i)}$ and $M_{(i)}$, will become clear when we define the successor relations. Thus,

$$\begin{aligned} M_{(i,j)} &:= (Q^{(i,j)}, \Sigma, \Gamma, ¢, \$, q_i^{(i,j)}, k, \delta^{(i,j)}), \\ M_{(i)} &:= (Q^{(i)}, \Sigma, \Gamma, ¢, \$, q_i^{(i)}, k, \delta^{(i)}), \end{aligned}$$

where $Q^{(i,j)} := \{q_0^{(i,j)}, \ldots, q_m^{(i,j)}\}$ and $Q^{(i)} := \{q_0^{(i)}, \ldots, q_m^{(i)}\}$ are copies of Q such that, for all $\alpha, \beta \in I$, if $\alpha \neq \beta$, then $Q^\alpha \cap Q^\beta = \emptyset$. Further, for all $i, j \in \{0, 1, \ldots, m\}$, $\delta^{(i,j)}$ is obtained from δ by replacing each occurrence of a state $q_l \in Q$ by an occurrence of the corresponding state $q_l^{(i,j)}$,

by replacing each occurrence of the restart operation $(q_j, \mathsf{Restart})$ by the simple restart operation $\mathsf{Restart}$, and by deleting all occurrences of restart operations of the form $(q_l, \mathsf{Restart})$, where $l \neq j$. Further, $\delta^{(i)}$ is obtained from $\delta^{(i,i)}$ by replacing each occurrence of a state $q_l^{(i,i)}$ by an occurrence of the corresponding state $q_l^{(i)}$. Finally, the successor relations are defined as follows:

$$\begin{array}{lcl} \sigma_{(i,j)} & := & \{\, (j,l) \mid 0 \leq l \leq m \,\} \text{ for } i \neq j, \\ \sigma_{(i,i)} & := & \{\, (i,l) \mid 0 \leq l \leq m, i \neq l \,\} \cup \{(i)\}, \\ \sigma_{(i)} & := & \{\, (i,l) \mid 0 \leq l \leq m \,\}. \end{array}$$

Observe that according to the definition of CD-systems of restarting automata, index (i,i) must not be a member of $\sigma_{(i,i)}$. That is the reason for introducing the machine $M_{(i)}$.

As each component automaton of the system $\mathcal{M}$ is essentially a forgetting copy of the nonforgetting restarting automaton M, only the initial states are different, we see that $\mathcal{M}$ consists of restarting automata of type X only. It remains to establish the following result.

Claim. $L(M) = L_{=1}(\mathcal{M})$.

Proof. Let $x \in L(M)$, that is, there exists a computation of M of the form

$$(q_0, x) \vdash_M^c (q_{i_1}, x_1) \vdash_M^c (q_{i_2}, x_2) \vdash_M^c \cdots \vdash_M^c (q_{i_l}, x_l) \vdash_M^c \mathsf{Accept}.$$

The CD-system $\mathcal{M}$ can simulate this computation as follows. As starting machine $M_{(0,i_1)}$ is chosen, and the computation of $\mathcal{M}$ starts with the cycle

$$(M_{(0,i_1)}, x) \vdash_{\mathcal{M}}^c (M_{(i_1,i_2)}, x_1),$$

as the restart operation $(q_{i_1}, \mathsf{Restart})$ that M applies in the first cycle above is replaced by the simple restart operation $\mathsf{Restart}$ of $M_{(0,i_1)}$ and as $(i_1, i_2) \in \sigma_{(0,i_1)}$holds, it follows that the automaton $M_{(i_1,i_2)}$ can become active and the initial state of this automaton is $q_{i_1}^{(i_1,i_2)}$, so the next cycle begins in the correct state.

If $i_1 \neq i_2$ or $i_2 \neq i_3$, then $\mathcal{M}$ continues with the cycle

$$(M_{(i_1,i_2)}, x_1) \vdash_{\mathcal{M}}^c (M_{(i_2,i_3)}, x_2),$$

as the restart operation $(q_{i_2}, \mathsf{Restart})$ that M applies in the second cycle above is replaced by the simple restart operation $\mathsf{Restart}$ of $M_{(i_1,i_2)}$, and as $(i_2, i_3) \in \sigma_{(i_1,i_2)}$holds, the initial state of $M_{(i_2,i_3)}$ is again a copy of the correct state needed to simulate the computation. If, however, $i_1 = i_2 = i_3$, then $\mathcal{M}$ continues with the cycle

$$(M_{(i_1,i_2)}, x_1) \vdash_{\mathcal{M}}^c (M_{(i_1)}, x_2),$$

as the restart operation $(q_{i_1}, \mathsf{Restart})$ that M applies in the second cycle above is replaced by the simple restart operation $\mathsf{Restart}$ of $M_{(i_1,i_1)}$, the automaton $M_{(i_1)}$ is chosen. Observe that $(i_1) \in \sigma_{(i_1,i_1)}$ is chosen, because a component must not be its own successor. The initial state of $M_{(i_1)}$ is another copy of q_{i_1}.

Inductively, it follows that $\mathcal{M}$ reaches the restarting configuration $(M_{(i_l,i_{l+1})}, ¢x_l\$)$ or $(M_{(i_l)}, ¢x_l\$)$, and that $M_{(i_l,i_{l+1})}$ or $M_{(i_l)}$, respectively, will accept in a tail computation when starting from this restarting configuration. Thus, we see that $L(M) \subseteq L_{=1}(\mathcal{M})$ holds.

Conversely, if $x \in L_{=1}(\mathcal{M})$, then there exist $(0, i_1) \in I_0$ and a computation of $\mathcal{M}$ of the following form:

$$x \vdash_{M_{(0,i_1)}}^c x_1 \vdash_{M_{(i_1,i_2)}}^c \cdots \vdash_{M_{(i_{l-1},i_l)}}^c x_l \vdash_{M_{(i_l,i_{l+1})}}^c \mathsf{Accept},$$

where $(i_j, i_{j+1}) \in \sigma_{(i_{j-1},i_j)}$ for all $j = 1, \ldots, l$. Observe that if some consecutive indices coincide, then instead of $M_{(i_j,i_j)}$ the automaton $M_{(i_j)}$ may occur. From the definition of $\mathcal{M}$ we see that M can execute the following computation:

$$(q_0, x) \vdash_M^c (q_{i_1}, x_1) \vdash_M^c (q_{i_2}, x_2) \vdash_M^c \cdots \vdash_M^c (q_{i_l}, x_l) \vdash_M^c \mathsf{Accept}.$$

Hence, $L(M) = L_{=1}(\mathcal{M})$ follows.

□

□

The converse holds for almost all modes of operations.

Theorem 4.2.2. *Let $j \geq 1$, and let $\mathsf{m} \in \{=j, \leq j, \geq j\}$ be a mode of operation. Then, for each* CD-X*-system $\mathcal{M}$, where $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$, there exists a non-forgetting* X*-automaton M such that $L(M) = L_{\mathsf{m}}(\mathcal{M})$ holds.*

Proof. Let $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$, let $j \geq 1$, let $\mathsf{m} \in \{=j, \leq j, \geq j\}$, and let $\mathcal{M} = ((M_i)_{i \in I}, I_0)$ be a CD-X-system, where $M_i = (Q_i, \Sigma, \Gamma, ¢, \$, q_0^{(i)}, k, \delta_i)$. We define a non-forgetting X-automaton $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, \delta)$ as follows:

Assume that, for each $i \in I$, the X-automaton M_i has state set $Q_i = \{ q_l^{(i)} \mid l = 0, 1, \ldots, n_i \}$ with initial state $q_0^{(i)}$. Then we take

$$Q := \bigcup_{i \in I} \{ q_l^{(i,l)} \mid l = 0, 1, \ldots, n_i, 1 \leq l \leq j \} \cup \{q_0\},$$

that is, Q consists of the disjoint union of j copies of each of the state sets of the automata M_i, $i \in I$, plus an additional new initial state q_0. Essentially the j copies of each state set Q_i will be used to count the number of iterations an automaton M_i is being simulated. The transition relation of M is defined as follows:

(i) *Transitions from the initial state q_0*:
For each possible content of the read/write window of the form $¢ \cdot u$,

$$\begin{aligned} \delta(q_0, ¢ \cdot u) \quad := \quad & \textstyle\bigcup_{i \in I_0} \{ (q_l^{(i,1)}, \mathsf{MVR}) \mid (q_l^{(i)}, \mathsf{MVR}) \in \delta_i(q_0^{(i)}, ¢ \cdot u) \} \\ & \cup \textstyle\bigcup_{i \in I_0} \{ (q_l^{(i,1)}, ¢ \cdot v) \mid (q_l^{(i)}, ¢ \cdot v) \in \delta_i(q_0^{(i)}, ¢ \cdot u) \} \\ & \cup (\{\mathsf{Accept}\} \cap \textstyle\bigcup_{i \in I_0} \delta(q_0^{(i)}, ¢ \cdot u)). \end{aligned}$$

Starting from an initial configuration these transitions allow M to pick an index $i \in I_0$ and to simulate the first step of the computation of M_i from the given initial configuration. In the first line all MVR-operations of all initial components with the tape content $¢ \cdot u$ are taken. In the second line all REWRITE-operations for the given tape content are given, and the tape content $¢ \cdot u$ is accepted, if one of the initial components accepts immediately.

(ii) *Simulating the automata M_i ($i \in I$)*:
For $i \in I$ and $1 \leq l \leq j$, the states from the set $\{ q_l^{(i,l)} \mid l = 0, 1, \ldots, n_i \}$ are used to simulate the automaton M_i step by step. Only the restart operations are replaced as described below.

(iii) *General restart transitions*:
For each index $i \in I$ and each $1 \leq n < j$, we introduce, for each restart operation of M_i of the form $\mathsf{Restart} \in \delta_i(q_l^{(i)}, u)$, a restart operation of the form $(q_0^{(i,n+1)}, \mathsf{Restart}) \in \delta(q_l^{(i,n)}, u)$, that is, the

second index is used to count the number of cycles of M_i that have been simulated consecutively. Further, we need some additional restart operations for M that depend on the mode of operation m.

(iv) *Restart transitions that depend on the mode* m *of* $\mathcal{M}$:
If m is mode $= j$, then, for each $i \in I$, we introduce restart operations $(q_0^{(s,1)}, \mathsf{Restart}) \in \delta(q_l^{(i,j)}, u)$, $s \in \sigma_i$, for each restart operation of M_i of the form $\mathsf{Restart} \in \delta_i(q_l^{(i)}, u)$. Thus, after executing the j-th cycle of M_i, an automaton M_s, $s \in \sigma_i$, is chosen.

If m is mode $\geq j$, then, for each $i \in I$, we introduce the restart operation $(q_0^{(i,j)}, \mathsf{Restart}) \in \delta(q_l^{(i,j)}, u)$ and the restart operations $(q_0^{(s,1)}, \mathsf{Restart}) \in \delta(q_l^{(i,j)}, u)$, $s \in \sigma_i$, for each restart operation of M_i of the form $\mathsf{Restart} \in \delta_i(q_l^{(i)}, u)$. Thus, after executing the j-th cycle of M_i, either an automaton M_s, $s \in \sigma_i$, is chosen, or another cycle of M_i is simulated, again with j as the second index.

Finally, if m is mode $\leq j$, then, for each $i \in I$ and each $1 \leq n \leq j$, we introduce restart operations $(q_0^{(s,1)}, \mathsf{Restart}) \in \delta(q_l^{(i,n)}, u)$, $s \in \sigma_i$, for each restart operation of M_i of the form $\mathsf{Restart} \in \delta_i(q_l^{(i)}, u)$. Thus, after executing the n-th cycle of M_i, either an automaton M_s, $s \in \sigma_i$, is chosen, or, if $n < j$, it is also possible to execute another cycle of M_i using a restart transition from (iii).

From the given construction it now follows easily that M is a non-forgetting X-automaton.

Claim $L(M) = L_{\mathsf{m}}(\mathcal{M})$

Proof. First we show, how M uses the second upper index to the conditions for the mode m, then we show, how M simulates a sequence of cycles of $\mathcal{M}$.

As described in (*iii*) M counts the number of cycles that the active automaton has already performed within the second upper index of the state and simulates a change of the component by changing the first upper index and resetting the second upper index to 1. This change of the component is done according to the mode of operation as described in (*iv*).

Let $x \in L_{\mathsf{m}}(\mathcal{M})$, that is, there exists a computation of M of the form

$$x \vdash_{M_1}^{c^{n_1}} x_1 \vdash_{M_2}^{c^{n_2}} \cdots \vdash_{M_l}^{c^{n_l}} x_l \vdash_{M_{l+1}}^{c^{n_{l+1}}} \mathsf{Accept}.$$

How long a component is active depends on the mode of operation. The nonforgetting restarting automaton M can simulate this computation as follows: M chooses in its first transition step to simulate the component M_1 and starts with its first step. It then simulates one cycle of the chosen component and restarts either in $q_0^{(1,2)}$ or in $q_0^{(2,1)}$ according to the mode m. So M can simulate n_i cycles of each component M_i, $1 \leq i \leq l$ by increasing the second upper index up to n_i and then "change" the component by restarting in $q_0^{(i+1,1)}$. Then it simulates a number of cycles of that component and so on until M accepts from a restarting configuration $(q_0^{(l+1,n_{l+1})}$.

Conversely if $x \in L(M)$ then there exists a sequence of cycles

$$(q_0, x) \vdash_M^c (q_0^{(1,2)}, x_{(1,2)}) \ldots \vdash_M^c (q_0^{(1,n_1)}, x_{(1,n_1)}) \vdash_M^c (q_0^{(2,1)}, x_{(2,1)}) \ldots$$
$$\ldots \vdash_M^c (q_0^{(l,1)}, x_{(l,1)}) \ldots \vdash_M^c (q_0^{(l,n_l)}, x_{(l,n_l)}) \vdash_M^c \mathsf{ACCEPT}$$

and the first upper index is changed due to the conditions described in (iv) which corresponds to the mode of operation m. Hence, $L(M) = L_{\mathsf{m}}(\mathcal{M})$ follows. □

□

Thus, we see that CD-systems of restarting automata in mode $= 1$ are just as powerful as non-forgetting restarting automata. Also we see that a CD-system working in mode $= j$, $\leq j$, or $\geq j$ can be simulated by a CD-system of the same type that works in mode $= 1$.

It remains to study CD-systems working in mode t. Here we have the following result.

Theorem 4.2.3. *Let* $\mathsf{X} \in \{\mathsf{RR}, \mathsf{RL}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, and let* $\mathcal{M}$ *be a* CD-X*-system. Then there exists a non-forgetting* X*-automaton* M *such that* $L(M) = L_{\mathsf{t}}(\mathcal{M})$ *holds.*

Proof. Let $\mathsf{X} \in \{\mathsf{RR}, \mathsf{RRW}, \mathsf{RRWW}\}$ and $\mathcal{M} = ((M_i, \sigma_i)_{i \in I}, I_0)$ be a CD-X-system. Nondeterministic RL- RLW- and RLWW-automata can be simulated by RR- RRW- and RRWW-automata, therefore only CD-systems of restarting automata without MVL-operations need to be regarded in this proof.

The problem in simulating the mode t computations of $\mathcal{M}$ by a non-forgetting X-automaton M is caused by the way in which the computation of $\mathcal{M}$ switches from one component to the next. If a component M_i gets stuck after having performed a positive number of cycles, that is, it can neither execute another cycle nor an accepting tail, then a component M_j with $j \in \sigma_i$ takes over. However, if M_i gets stuck immediately, that is, without having executed any cycle, then the overall computation of $\mathcal{M}$ fails, that is, in this case $\mathcal{M}$ halts and rejects. The non-forgetting X-automaton M tries to simulate the computations of $\mathcal{M}$ cycle by cycle. However, when simulating the component M_i of $\mathcal{M}$, then M must decide at the end of each cycle, that is, when executing the corresponding restart operation, whether to start the simulation of another cycle of M_i or whether to start the simulation of a cycle of a component M_j for some $j \in \sigma_i$. To solve this problem we proceed as follows.

The non-forgetting X-automaton M is essentially obtained as the disjoint union of the components of the CD-X-system $\mathcal{M}$ as described in the proof of the previous theorem. However, we have to adjust each component to solve the above-mentioned problem about the restart operations.

M_i reads its tape completely within each cycle, that is, it executes a restart operation only at the right delimiter \$. Hence, while simulating a cycle of M_i, the automaton M can already check whether all computations of M_i that follow the current cycle will be rejecting tails, or whether another Restart or an accepting tail follows. This is best described in terms of meta-instructions.

Let $x \in L_{\mathsf{t}}(\mathcal{M})$, that is, there exists a computation of M of the form

$$x \vdash_{M_{i_1}}^{c^{n_1}} x_1 \vdash_{M_{i_2}}^{c^{n_2}} \cdots \vdash_{M_{i_l}}^{c^{n_l}} x_l \vdash_{M_{i_{l+1}}}^{c^{n_{l+1}}} \mathsf{Accept}.$$

This means that the component M_{i_1} performs n_1 cycles before it rejects and then the component M_{i_2}, with $i_2 \in \sigma_{i_1}$, become active. When M_{i_2} is unable to perform another cycle after n_2 many cycles, M_{i_3} becomes active. This is done until $M_{i_{l+1}}$ accepts after n_{l+1} many cycles.

M simulates the computation of $\mathcal{M}$ cycle by cycle. M starts simulating the first cycle of M_{i_1} from the initial configuration $q_0 \text{¢} x \$$ as described in the previous proof. While M simulates a cycle $x \vdash_{M_{i_1}}^{c} y$ of M_{i_1} it also checks all meta-instructions $I_j = (E_j, u_j \to v_j, {E_j}')$ of M_{i_1} within its finite control to determine, whether M_{i_1} is able to perform another cycle afterwards. It does this by checking if y has a factorization $w_j u_j w_j'$ with $w_j \in E_j, w_j' \in E_j'$. If y has such a factorization, that means that the meta-instruction I_j is applicable, then M restarts in q_0^i simulating another cycle of M_{i_1}. Otherwise none of the meta-instructions of M_{i_1} is applicable, and therefore M_{i_1} is unable to perform another cycle, starting with y. Accordingly, at the end of the current cycle M executes a restart operation that transfers control to the initial state of the component M_{i_2}, $i_2 \in \sigma_{i_1}$.

It is easily seen that $L_{\mathsf{t}}(\mathcal{M}) \subseteq \mathcal{L}(M)$ follows.

The other direction is straightforward, because if $x \in L(M)$, then there exists a sequence of cycles

$$(q_0, x) \vdash_M^{c^{n_1}} (q_0^{i_1}, \hat{x}) \vdash_M^{c^{n_1-1}} (q_0^{i_2}, x_1) \vdash_M^{c^{n_2}} \cdots \vdash_M^{c^{n_l}} (q_0^{i_l}, x_l) \vdash_M^{c^{n_{l+1}}} \mathsf{Accept},$$

and q_0^i corresponds to the initial state of M_i. This describes exactly a computation of $\mathcal{M}$ in mode t. □

It is an open question whether the latter result extends to CD-systems of R(W)(W)-automata. The problem with these types of restarting automata stems from the fact that within a cycle such an automaton will in general not see the complete tape content. Therefore it is not clear how a nonforgetting restarting automaton can determine whether another cycle is applicable in the same component or not.

4.3 Deterministic CD-Systems of Restarting Automata

In general it is very easy and natural to define determinism: an automaton is deterministic, if all its computations are deterministic. For CD-systems it is a bit more difficult, because a system consists of several components. First of all, each component of a CD-system has to be deterministic: this is the first notion of determinism in this work. But there are two other notions, which are also introduced here, because the first notion of determinism leads to CD-systems which are not necessarily completely deterministic.

CD-grammar systems are called deterministic, if each component is deterministic (see, e.g., [DP97, CVDKP94]). This is the motivation for the first notion of determinism. A CD-system of restarting automata is called *locally deterministic*, if all its components M_i are deterministic restarting automata. It is called locally deterministic, because only the components are deterministic, but in general the whole system is not. Observe that the starting automaton is chosen nondeterministically from the set of initial components M_i, $i \in I_0$, and also a successor automaton for it is chosen nondeterministically from among the successor components M_j , $j \in \sigma_i$, and so on.

The above definition can be strengthened by removing this nondeterminism. A CD-system of restarting automata is called *strictly deterministic*, if each component is deterministic, if I_0 is a singleton and each component has exactly one successor in its successor relation. The CD-R-system of Example 4.1.4 is strictly deterministic. This ensures that all computations of a strictly deterministic CD-system of restarting automata are deterministic.

We will see below that having only one successor is a rather serious restriction. Thus we introduce a third kind of determinism, which allows more than one successor component, but ensures that it is chosen deterministically by changing the restart operation. A CD-system of restarting automata is called globally deterministic, if I_0 is a singleton, each component is a deterministic restarting automaton and the successor is chosen deterministically from among all possible successors of this component. This is achieved by combining each restart operation from M_i with an index from the set σ_i. As long as M_i is not finished with the computation the index is ignored, but if according to the given mode of operation M_i has finished its last cycle, the component, whose index is assigned to the final restart operation, takes over.

For example, when working in mode $= j$ for some $j > 1$, then the first $j - 1$ applications of restart steps within the computation of a component M_i just restart M_i itself, but at the j-th application of a restart step, the component $l \in \sigma_i$ becomes active, if l is the index associated with this particular restart operation. In this way it is guaranteed that all computations of a globally deterministic CD-system are deterministic. However, for a component M_i there can still be several

possible successor components. This is reminiscent of the way in which nonforgetting restarting automata work.

We use the prefix det-local to denote locally deterministic CD-systems, the prefix det-global to denote globally deterministic CD-systems, and the prefix det-strict to denote strictly deterministic CD-systems. For each type of restarting automaton $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$ and any mode m, it is easily seen that the following inclusions hold:

$$\mathcal{L}(\text{det-}\mathsf{X}) \subseteq \mathcal{L}_{\mathsf{m}}(\text{det-strict-CD-}\mathsf{X}) \subseteq \mathcal{L}_{\mathsf{m}}(\text{det-global-CD-}\mathsf{X}).$$

The missing inclusions between globally and locally deterministic CD-systems are not so easily seen. They are treated in the subsection about locally deterministic CD-systems.

The modes $\leq j$ and $\geq j$ are nondeterministic modes, so even strictly or globally deterministic CD-systems working in these modes have some nondeterminism. In the subsections about globally and strictly deterministic CD-systems we only consider deterministic modes. In the subsection about strictly deterministic CD-systems it will be shown that the above inclusions are proper, when no auxiliary symbols are available.

As mentioned in the beginning of this chapter, the strong correctness preserving property does not hold for all kinds of deterministic CD-systems of restarting automata. Strictly and globally deterministic CD-systems of restarting automata are completely deterministic. Locally deterministic CD-systems are not necessarily completely deterministic, therefore they are not strongly correctness preserving. For globally or strictly deterministic CD-systems of restarting automata working in a deterministic mode (t or $= j$), every successor restarting configuration of a promising configuration is again a promising configuration. Thus the following proposition holds.

Proposition 4.3.1. *Each strictly or globally deterministic* CD*-system* $\mathcal{M}$ *operating in mode* $= j$ *or* t *is strongly correctness preserving.*

4.3.1 Globally Deterministic CD-systems

Concerning the globally deterministic CD-systems, we have the following results, which correspond to the results for nondeterministic CD-systems.

Theorem 4.3.2. *If* M *is a nonforgetting deterministic restarting automaton of type* X *for some* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, then there exists a globally deterministic CD-system* $\mathcal{M}$ *of restarting automata of type* X *such that* $L_{=1}(\mathcal{M}) = L(M)$ *holds.*

Proof. Let $M = (Q, \Sigma, \Gamma, \mathrm{¢}, \$, q_0, k, \delta)$ be a nonforgetting deterministic restarting automaton of type X, where $Q = \{q_0, q_1, \ldots, q_m\}$. We define a simulating globally deterministic CD-system $\mathcal{M}$ of X-automata as follows.

Let $I := \{\, i \mid 0 \leq i \leq m \,\} \cup \{\, \hat{i} \mid 0 \leq i \leq m \,\}$ be the set of indices, and let 0 be the initial index. For each index $i \in \{0, 1, \ldots, m\}$, we take two copies M_i and $M_{\hat{i}}$ of M that simulate those computations of M that start with a restarting configuration of the form $q_i \mathrm{¢} w \$$. Thus,

$$\begin{aligned} M_i &:= (Q^{(i)}, \Sigma, \Gamma, \mathrm{¢}, \$, q_i^{(i)}, k, \delta^{(i)}), \\ M_{\hat{i}} &:= (Q^{(\hat{i})}, \Sigma, \Gamma, \mathrm{¢}, \$, q_i^{(\hat{i})}, k, \delta^{(\hat{i})}), \end{aligned}$$

where $Q^{(i)} := \{q_0^{(i)}, \ldots, q_m^{(i)}\}$ and $Q^{(\hat{i})} := \{q_0^{(\hat{i})}, \ldots, q_m^{(i)}\}$ are copies of Q such that, for all $\alpha, \beta \in I$, if $\alpha \neq \beta$, then $Q^\alpha \cap Q^\beta = \emptyset$. Further, for all $i \in \{0, 1, \ldots, m\}$, $\delta^{(i)}$ is obtained from δ by replacing

each occurrence of a state $q_l \in Q$ by an occurrence of the corresponding state $q_l^{(i)}$, and by replacing each occurrence of a restart operation $(q_j, \mathsf{Restart})$ $(0 \leq j \leq m)$ by the restart operation $\mathsf{Restart}$, combined with the successor $\hat{j}$. Further, $\delta^{(\hat{i})}$ is obtained from δ by replacing each occurrence of a state $q_l \in Q$ by an occurrence of the corresponding state $q_l^{(\hat{i})}$, and by replacing each occurrence of a restart operation $(q_j, \mathsf{Restart})$ $(0 \leq j \leq m)$ by the restart operation $\mathsf{Restart}$, combined with the successor j. Finally, the successor relations are defined as follows:

$$\begin{array}{rcl} \sigma_i & := & \{\, \hat{j} \mid 0 \leq j \leq m \,\} \text{ for all } 0 \leq i \leq m, \\ \sigma_{\hat{i}} & := & \{\, j \mid 0 \leq j \leq m \,\} \text{ for all } 0 \leq i \leq m. \end{array}$$

Observe that according to the definition of CD-systems of restarting automata, index i must not be a member of σ_i. That is the reason for introducing the automata $M_{\hat{i}}$.

As each component automaton of the system $\mathcal{M}$ is essentially a copy of the restarting automaton M, we see that $\mathcal{M}$ consists of deterministic restarting automata of type X only. The initial index is 0 and each restart is assigned with a successor, thus $\mathcal{M}$ is a globally deterministic CD-X-system. It remains to establish the following result.

Claim. $L(M) = L_{=1}(\mathcal{M})$.

Proof. Let $x \in L(M)$, that is, there exists a computation of M of the form

$$(q_0, x) \vdash_M^c (q_1, x_1) \vdash_M^c (q_2, x_2) \vdash_M^c \cdots \vdash_M^c (q_l, x_l) \vdash_M^c \mathsf{Accept}.$$

The CD-system $\mathcal{M}$ can simulate this computation as follows: The computation of $\mathcal{M}$ starts with the cycle

$$(M_0, x) \vdash_{\mathcal{M}}^c (M_{\hat{1}}, x_1),$$

as the restart operation $(q_{i_1}, \mathsf{Restart})$ that M applies in the first cycle above is replaced by the simple restart operation $\mathsf{Restart}$ of M_0 combined with the successor $\hat{i}_1$. The component $M_{\hat{i}_1}$ becomes active. This component has as initial state $q_{i_1}^{(\hat{i}_1)}$. Therefore $\mathcal{M}$ starts its next cycle in the correct state:

$$(M_{\hat{1}}, x_1) \vdash_{\mathcal{M}}^c (M_2, x_2).$$

The restart operation $(q_{i_2}, \mathsf{Restart})$ that M applies in the second cycle above is replaced by the simple restart operation $\mathsf{Restart}$ of $M_{\hat{i}_1}$ combined with the successor i_2. Inductively, it follows that $\mathcal{M}$ reaches the restarting configuration $q_{i_l}^{(i_l)}\mathbin{¢} x_l \$$ or $q_{i_l}^{(\hat{i}_l)}\mathbin{¢} x_l \$$, and that M_{i_l} or $M_{\hat{i}_l}$, respectively, will accept in a tail computation when starting from this restarting configuration. Thus, we see that $L(M) \subseteq L_{=1}(\mathcal{M})$ holds.

Conversely, if $x \in L_{=1}(\mathcal{M})$, then there exists a computation of $\mathcal{M}$ of the form

$$x \vdash_{M_0}^c x_1 \vdash_{M_{\hat{i}_1}}^c \cdots \vdash_{M_{i_{l-1}}}^c x_l \vdash_{M_{\hat{i}_l}}^* \mathsf{Accept},$$

or

$$x \vdash_{M_0}^c x_1 \vdash_{M_{\hat{i}_1}}^c \cdots \vdash_{M_{\hat{i}_{l-1}}}^c x_l \vdash_{M_{i_l}}^* \mathsf{Accept}.$$

From the definition of $\mathcal{M}$ we see that in either case M can execute the following computation:

$$(q_0, x) \vdash_M^c (q_{i_1}, x_1) \vdash_M^c (q_{i_2}, x_2) \vdash_M^c \cdots \vdash_M^c (q_{i_l}, x_l) \vdash_M^* \mathsf{Accept}.$$

Hence, $L(M) = L_{=1}(\mathcal{M})$ follows. □

□

For the converse we again have the following stronger result, but remark that only the deterministic modes $= j$ are considered.

Theorem 4.3.3. *For each* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, if* $\mathcal{M}$ *is a globally deterministic* CD-X*-system, and if* j *is a positive integer, then there exists a nonforgetting deterministic* X*-automaton* M *such that* $L(M) = L_{=j}(\mathcal{M})$ *holds.*

Proof. Let $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$, let $j \geq 1$, and let $\mathcal{M} = ((M_i)\ i \in I,\ \{i_0\})$ be a globally deterministic CD-X-system, where $M_i = (Q_i, \Sigma, \Gamma, ¢, \$, q_0^{(i)}, k, \delta_i)$. We define a nonforgetting deterministic X-automaton $M = (Q, \Sigma, \Gamma,\ ¢, \$, q_0, k, \delta)$ as follows:

Assume that, for each $i \in I$, the X-automaton M_i has the state set $Q_i = \{\, q_l^{(i)} \mid l = 0, 1, \ldots, n_i \,\}$ with initial state $q_0^{(i)}$. Then we take

$$Q := \bigcup_{i \in I} \{\, q_l^{(i,r)} \mid l = 0, 1, \ldots, n_i, 1 \leq r \leq j \,\},$$

that is, Q consists of the disjoint union of j copies of each of the state sets of the automata M_i, $i \in I$. Essentially the j copies of each state set Q_i will be used to count the number of iterations an automaton M_i is being simulated. As initial state we choose $q_0 := q_0^{(i_0,1)}$, as each computation of $\mathcal{M}$ starts with the automaton M_{i_0}. The transition relation of M is defined as follows:

(i) *Simulating the automata* M_i $(i \in I)$:
For $i \in I$ and $1 \leq r \leq j$, the states from the set $\{\, q_l^{(i,r)} \mid l = 0, 1, \ldots, n_i \,\}$ are used to simulate the automaton M_i step by step. However, the restart operations are replaced as described below.

(ii) *Restart steps*:
For each index $i \in I$, for each $1 \leq r < j$, and for each restart operation of M_i of the form $\delta_i(q_l^{(i)}, u) = \mathsf{Restart}$, we introduce a restart operation of the form $\delta(q_l^{(i,r)}, u) = (q_0^{(i,r+1)}, \mathsf{Restart})$, that is, the second index is used to count the number of cycles of M_i that have been simulated consecutively.

In addition, for each $i \in I$, we introduce a restart operation $\delta(q_l^{(i,j)}, u) = (q_0^{(s,1)}, \mathsf{Restart})$ for each restart operation of M_i of the form $\delta_i(q_l^{(i)}, u) = \mathsf{Restart}$, where $s \in \sigma_i$ is the successor that is associated to this restart operation of M_i. Thus, after executing the j-th cycle of M_i, the computation continues with the automaton M_s.

From the given construction it follows similar to the proof of Theorem 4.2.2 that M is a nonforgetting deterministic X-automaton, and that $L(M) = L_{=j}(\mathcal{M})$ holds. □

Thus, we see that globally deterministic CD-systems of restarting automata working in mode $= 1$ are just as powerful as nonforgetting deterministic restarting automata. It remains to study CD-systems that work in mode t.

In the nondeterministic case, RL-, RLW- and RLWW-automata needed no special treatment, in the deterministic case they are strictly more powerful than the corresponding RR-, RRW- and RRWW-automata and must be handled separately. For this we first restate the normalform for det-RL-, det-RLW- and det-RLWW-automata. This normalform corresponds to the one established in Lemma 3.3.2 on page 42.

Let $\mathsf{X} \in \{\varepsilon, \mathsf{W}, \mathsf{WW}\}$. For each det-RLX-automaton M, there exist a det-RLX-automaton M' that performs its restart operation immediately after its rewrite operation, such that $L(M) = L(M')$ holds.

With this normalform we can prove the following theorem for all kinds of restarting automata.

Theorem 4.3.4. *Let* $\mathsf{X} \in \{\mathsf{RR}, \mathsf{RL}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, and let* $\mathcal{M}$ *be a globally deterministic* CD-X*-system. Then there exists a nonforgetting deterministic* X*-automaton* M *such that* $L(M) = L_{\mathsf{t}}(\mathcal{M})$ *holds.*

Proof. Let $\mathsf{X} \in \{\mathsf{RR}, \mathsf{RL}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RRWW}, \mathsf{RLWW}\}$, and let $\mathcal{M} = ((M_i, \sigma_i)_{i \in I}, \{i_0\})$ be a globally deterministic CD-X-system. Again, as in the proof of Theorem 4.2.3, the problem in simulating the mode t computations of $\mathcal{M}$ by a nonforgetting deterministic X-automaton M is caused by the way in which the computation of $\mathcal{M}$ switches from one component to the next. If a component M_i gets stuck after having performed a positive number of cycles, that is, it can neither execute another cycle nor an accepting tail, then a component M_j with $j \in \sigma_i$ takes over. As $\mathcal{M}$ is globally deterministic, the index $j \in \sigma_i$ is determined by the last restart operation executed by M_i in the current computation. However, if M_i gets stuck immediately, that is, without having executed a single cycle, then the overall computation of $\mathcal{M}$ fails, that is, in this case $\mathcal{M}$ halts and rejects. The nonforgetting deterministic X-automaton M will simulate the computations of $\mathcal{M}$ cycle by cycle.

However, when simulating the component M_i of $\mathcal{M}$, then M must decide at the end of each cycle, that is, when executing the corresponding restart operation, whether to start the simulation of another cycle of M_i or whether to start the simulation of a cycle of the component M_j, where $j \in \sigma_i$ is the index assigned to the last restart operation. To solve this problem we proceed as follows.

For RL-, RLW- and RLWW-automata we can assume that M_i restarts immediately after executing a rewrite (Lemma 3.3.2). This property is needed, when M tries to find a succeeding cycle. A nonforgetting deterministic restarting automaton M with MVL-operations acts as the component automaton M_i it wants to simulate.

The only difference is that when M_i would perform a restart, M moves to the left end of the tape and tries to execute another cycle. If it reaches a configuration such that M_i can perform a rewrite, and because of the normalform also a restart, then M knows that M_i can perform another cycle afterwards. In this case M restarts again in the same "component". If it is unable to reach another rewrite, it restarts in the "component", which is assigned to the most resent restart of M_i.

Thus after it has simulated one cycle of M_i, M can check whether this component is able to perform another cycle or not and behave accordingly.

For RR-, RRW- and RRWW-automata the simulation is done similar to the nondeterministic case (Theorem 4.2.3).

Again, we can assume that M_i reads its tape completely within each cycle, that is, it executes a restart operation only at the right delimiter \$. Hence, while simulating a cycle of M_i, the automaton M can already check whether all computations of M_i that follow the current cycle will be rejecting tails, or whether another Restart or an accepting tail follows. This is best described in terms of meta-instructions.

Let $x \in L_{\mathsf{t}}(\mathcal{M})$, that is, there exists a computation of M of the form

$$x \vdash^{c^{n_1}}_{M_{i_1}} x_1 \vdash^{c^{n_2}}_{M_{i_2}} \cdots \vdash^{c^{n_l}}_{M_{i_l}} x_l \vdash^{c^{n_{l+1}}}_{M_{i_{l+1}}} \mathsf{Accept},$$

This means that the component M_{i_1} performs n_1 cycles before it rejects and then the component M_{i_2}, with $i_2 \in \sigma_1$ and M_{i_2} is assigned to the last restart operation of M_{i_1}, becomes active. When M_{i_2} is unable to perform another cycle after n_2 many cycles, M_{i_3} becomes active. This is done until $M_{i_{l+1}}$ accepts after n_{l+1} many cycles.

M simulates the computation of $\mathcal{M}$ cycle by cycle. It starts simulating the first cycle of M_{i_1} starting from the initial configuration $q_0 ¢ x \$$. While M simulates a cycle $x \vdash_{M_i}^{c} y$ of M_i it also checks all meta-instructions $I_n = (E_n, u_n \to v_n, {E_n}')$ of M_i within its finite control, to determine if M_i is able to perform another cycle afterwards. It does this, by checking, if y has a factorization $w_j u_j w_j'$ with $w_j \in E_j, w_j' \in E_j'$. If y has such a factorization, that means that the meta-instruction I_j is applicable after the current cycle, then M restarts in q_0^i simulating another cycle of M_i.

Otherwise none of the meta-instructions of M_i are applicable, and therefore M_i is unable to perform another cycle, starting with y. Accordingly, at the end of the current cycle M restarts in the restarting state that corresponds to the initial state of the components M_j, $j \in \sigma_i$, which is assigned to the corresponding restart of $\mathcal{M}$.

It is easily seen that $L_{\mathsf{t}}(\mathcal{M}) \subseteq L(M)$ follows.

The other direction is straight forward, because if $x \in L(M)$, then there exists a sequence of cycles

$$(q_0, x) \vdash_M^{c^{n_1}} (q_0^1, \hat{x}) \vdash_M^{c^{n_1-1}} (q_0^2, x_1) \vdash_M^{c^{n_2}} \cdots \vdash_M^{c^{n_l}} (q_0^l, x_l) \vdash_M^{c^{n_{l+1}}} \mathsf{Accept},$$

and q_0^l corresponds to the initial state of M_l. This describes exactly a computation of $\mathcal{M}$.

□

Again it is not clear whether the latter result extends to CD-systems of R(W)(W)-automata. The problem with these types of restarting automata stems from the fact that within a cycle such an automaton will in general not see the complete tape content.

4.3.2 Strictly Deterministic CD-Systems

Here we study the expressive power of strictly deterministic CD-systems of restarting automata. As seen in Example 4.1.4 the copy language L_{copy} is accepted by a strictly deterministic CD-R-system with two components. This language is not growing context-sensitive [Bun96, Lau88]. As deterministic RRWW-automata only accept Church-Rosser languages, which are a proper subclass of the growing content-sensitive languages, this yields the following separation result.

Proposition 4.3.5. *For all* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RWW}, \mathsf{RRWW}\}$,

$$\mathcal{L}(\text{det-X}) \subsetneq \mathcal{L}_{=1}(\text{det-strict-CD-X}).$$

Let $L_{\text{copy}^m} := \{\, w(\#w)^{m-1} \mid w \in \Sigma_0^+ \,\}$ be the m-fold copy language. Analogously to Example 4.1.4 it can be shown that this language is accepted by a strictly deterministic CD-R-system with m components that works in mode = 1. The next example deals with a generalization of these languages.

Example 4.3.6. Let $L_{\text{copy}^*} := \{\, w(\#w)^n \mid w \in (\Sigma_0^2)^+,\, n \geq 1 \,\}$ be the copy-star language, where $\Sigma_0 := \{a, b\}$. Here the restriction to non-empty factors $w \in \Sigma_0^*$ of even length is introduced just to simplify the description of a CD-system for the language L_{copy^*}.

Let $\mathcal{M} := ((M_1, \{2\}), (M_2, \{1\}), \{1\})$ be the following CD-RWW-system. The input alphabet is $\Sigma := \Sigma_0 \cup \{\#\}$, the tape alphabet is $\Gamma := \Sigma \cup \Gamma_0$, where $\Gamma_0 := \{A_{a,a}, A_{a,b}, A_{b,a}, A_{b,b}\}$, and the RWW-automata M_1 and M_2 are given through the following meta-instructions, where $c, d, e, f \in \Sigma_0$:

$$
\begin{array}{ll}
M_1: & (\mathrm{¢} \cdot (\Sigma_0^2)^* \cdot cd \cdot \# \cdot (\Sigma_0^2)^*, cd \cdot \# \to A_{c,d} \cdot \#),\\
& (\mathrm{¢} \cdot (\Sigma_0^2)^* \cdot cd \cdot \# \cdot (\Sigma_0^2)^*, cd \cdot \$ \to A_{c,d} \cdot \$),\\
& (\mathrm{¢} \cdot (\Sigma_0^2)^* \cdot cd \cdot \# \cdot (\Sigma_0^2)^*, cdA_{e,f} \to A_{c,d}A_{e,f}),\\
& (\mathrm{¢} \cdot \Gamma_0^* \cdot A_{c,d} \cdot \# \cdot (\Sigma_0^2)^*, cd \cdot \# \to A_{c,d} \cdot \#),\\
& (\mathrm{¢} \cdot \Gamma_0^* \cdot A_{c,d} \cdot \# \cdot (\Sigma_0^2)^*, cd \cdot \$ \to A_{c,d} \cdot \$),\\
& (\mathrm{¢} \cdot \Gamma_0^* \cdot A_{c,d} \cdot \# \cdot (\Sigma_0^2)^*, cdA_{e,f} \to A_{c,d}A_{e,f}),\\
& (\mathrm{¢} \cdot \Gamma_0^+ \cdot \$, \mathsf{Accept}),\\
M_2: & (\mathrm{¢} \cdot (\Sigma_0^2)^+, cd \cdot \# \to \#),\\
& (\mathrm{¢}, cd \cdot \# \to \varepsilon),\\
& (\mathrm{¢} \cdot \Gamma_0^+, A_{c,d} \cdot \# \to \#),\\
& (\mathrm{¢}, A_{c,d} \cdot \# \to \varepsilon).
\end{array}
$$

It is easily verified that M_1 and M_2 are deterministic RWW-automata, and hence, $\mathcal{M}$ is a strictly deterministic CD-RWW-system. In mode $= 1$, the two components M_1 and M_2 are used alternatingly, with M_1 starting the computation. Let $x := w_1 \# w_2 \# \ldots \# w_m$ be the given input, where $w_1, w_2, \ldots, w_m \in (\Sigma_0^2)^+$ and $m \geq 2$. First w_1 is compared to w_2 by processing these strings from right to left, two letters in each round. During this process w_1 is erased, while w_2 is encoded using the letters from Γ_0. Next the encoded version of w_2 is used to compare w_2 to w_3, again from right to left. This time the encoded version of w_2 is erased, while w_3 is encoded. This continues until all syllables w_i have been considered. It follows that $L_{=1}(\mathcal{M}) = L_{\mathrm{copy}^*}$ holds.

For accepting the language L_{copy^*} without using auxiliary symbols we have a CD-system of restarting automata that is globally deterministic.

Lemma 4.3.7. *The language L_{copy^*} is accepted by a globally deterministic* CD-R*-system working in mode $= 1$.*

Proof. Let $\mathcal{M} := ((M_i, \sigma_i)_{i \in I}, I_0)$ be the CD-R-system that is specified by $I := \{0, 1, 2, 3, 4, 5, 6\}$, $I_0 := \{0\}$, $\sigma_0 := \{5\}$, $\sigma_1 := \{2, 6\}$, $\sigma_2 := \{1, 6\}$, $\sigma_3 := \{4, 5\}$, $\sigma_4 := \{3, 5\}$, $\sigma_5 := \{1\}$, $\sigma_6 := \{3\}$, and M_0 to M_6 are given through the following meta-instructions, where $c, d \in \Sigma_0$:

$$
\begin{array}{lll}
M_0 & : & (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot \Sigma_0 \cdot c \cdot \#)^+ \cdot (\Sigma_0^2)^+ \cdot \Sigma_0, c \cdot \$ \to \$, \mathsf{Restart}(5)),\\
& & (\mathrm{¢} \cdot (u \cdot \#)^+ \cdot u \cdot \$, \mathsf{Accept}) \text{ for all } u \in \Sigma_0^2,\\
M_1 & : & (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot c \cdot \#)^+ \cdot (\Sigma_0^2)^+, c \cdot \$ \to \$, \mathsf{Restart}(6)),\\
& & (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot c \cdot \#)^+ \cdot (\Sigma_0^2)^+ \cdot c, d \cdot \# \to \#, \mathsf{Restart}(2)),\\
M_2 & : & (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot c \cdot \#)^+ \cdot (\Sigma_0^2)^+, c \cdot \$ \to \$, \mathsf{Restart}(6)),\\
& & (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot c \cdot \#)^+ \cdot (\Sigma_0^2)^+ \cdot c, d \cdot \# \to \#, \mathsf{Restart}(1)),\\
M_3 & : & (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot \Sigma_0 \cdot c \cdot \#)^+ \cdot (\Sigma_0^2)^+ \cdot \Sigma_0, c \cdot \$ \to \$, \mathsf{Restart}(5)),\\
& & (\mathrm{¢} \cdot ((\Sigma_0^2)^* \cdot \Sigma_0 \cdot c \cdot \#)^+ \cdot (\Sigma_0^2)^* \cdot \Sigma_0 \cdot c, d \cdot \# \to \#, \mathsf{Restart}(4)),\\
& & (\mathrm{¢} \cdot (u \cdot \#)^+ \cdot u \cdot \$, \mathsf{Accept}) \text{ for all } u \in \Sigma_0^2,\\
M_4 & : & (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot \Sigma_0 \cdot c \cdot \#)^+ \cdot (\Sigma_0^2)^+ \cdot \Sigma_0, c \cdot \$ \to \$, \mathsf{Restart}(5)),\\
& & (\mathrm{¢} \cdot ((\Sigma_0^2)^* \cdot \Sigma_0 \cdot c \cdot \#)^+ \cdot (\Sigma_0^2)^* \cdot \Sigma_0 \cdot c, d \cdot \# \to \#, \mathsf{Restart}(3)),\\
& & (\mathrm{¢} \cdot (u \cdot \#)^+ \cdot u \cdot \$, \mathsf{Accept}) \text{ for all } u \in \Sigma_0^2,\\
M_5 & : & (\mathrm{¢} \cdot \Sigma_0^+, c \cdot \# \to \#, \mathsf{Restart}(1)),\\
M_6 & : & (\mathrm{¢} \cdot \Sigma_0^+, c \cdot \# \to \#, \mathsf{Restart}(3)).
\end{array}
$$

Clearly M_0 to M_6 are deterministic R-automata, and hence, $\mathcal{M}$ is a globally deterministic CD-R-system. Given an input of the form $w\#w\#\cdots\#w$, where $|w| = 2m > 2$, M_0 verifies that all syllables are of even length, and that they all end in the same letter, say c. This letter c is deleted from the last syllable, and M_5 is called, which simply deletes the last letter (that is, c) from the first syllable. Now M_1 is called, which in cooperation with M_2, removes the last letter from all the other syllables. Finally the tape content $w_1\#w_1\#\cdots\#w_1$ is reached, where $w = w_1c$. In this situation M_1 (or M_2) notices that all syllables are of odd length, and that they all end with the same letter, say d, which it then removes from the last syllable. Now using M_6, M_3, and M_4 this letter is removed from all other syllables. This process continues until either an error is detected, in which case $\mathcal{M}$ rejects, or until a tape content of the form $u\#u\#\cdots\#u$ is reached for a word $u \in \Sigma_0^2$, in which case $\mathcal{M}$ accepts. Thus, we see that $\mathcal{M}$ accepts the language L_{copy^*} working in mode = 1. □

The following negative result contrasts the positive results above.

Theorem 4.3.8. *The language L_{copy^*} is not accepted by any strictly deterministic* CD-RR*-system that is working in mode = 1.*

Proof. Let $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$ be a strictly deterministic CD-RR-system that accepts the language L_{copy^*} in mode = 1. We can assume that $I = \{0, 1, \ldots, m\}$, that $I_0 = \{0\}$, that $\sigma_i = \{i+1\}$ for all $i = 0, 1, \ldots, m-1$, and that $\sigma_m = \{s\}$ for some $s \in I$. Thus, each computation of $\mathcal{M}$ has the following structure:

$$w_0 \vdash_{\mathcal{M}}^{c^s} w_s \vdash_{M_s}^{c} w_{s+1} \vdash_{\mathcal{M}}^{c^{m-s-1}} w_m \vdash_{M_m}^{c} w_{m+1} \vdash_{M_s}^{c} w_{m+2} \vdash_{\mathcal{M}}^{c^{m-s-1}} \cdots,$$

that is, it is composed of a head $w_0 \vdash_{\mathcal{M}}^{c^s} w_s$ that consists of s cycles and of a sequence of meta-cycles of the form $w_s \vdash_{M_s}^{c} w_{s+1} \vdash_{\mathcal{M}}^{c^{m-s-1}} w_m \vdash_{M_m}^{c} w_{m+1}$ that consist of $m-s+1$ cycles each.

Let $x := w\#w(\#w)^n$ be an input word with $w \in (\Sigma_0^2)^*$, where $|w|$ and the exponent n are sufficiently large. Then $x \in L_{\text{copy}^*}$, and hence, the computation of $\mathcal{M}$ that begins with the restarting configuration $(M_0, q_0^{(0)}\text{¢}x\$)$ is accepting. We will now analyze this computation. The factors w of x and their descendants in this computation will be denoted as *syllables*. To simplify the discussion we use indices to distinguish between different syllables.

$\mathcal{M}$ must compare each syllable w_i to all the other syllables. As $|w_i| = |w|$ is large, it can compare w_i to w_j for some $j \neq i$ only piecewise. However, during this process it needs to distinguish the parts that have already been compared from those parts that have not. This can only be achieved by rewriting w_i and w_j accordingly. Since no auxiliary symbols are available, this must be done by rewrite operations that delete those parts of w_i and w_j that have already been compared. Hence, up to a finite part that remains of a syllable, each syllable can only be compared to other syllables once. Therefore $\mathcal{M}$ has to compare the syllables all at once. But there are only $m-s+1$ cycles in a meta-cycle, thus, $\mathcal{M}$ cannot use the information provided by a special component to compare the syllables.

On the other hand, if $\mathcal{M}$ uses some regular condition, i.e. even or odd length of the syllables to rewrite all syllables at a time, then all components behave equal during a sequence of cycles that shortens all syllables. Therefore it can only shorten each syllable once. Because if any component rewrites the last syllable, then the other syllables remain unchanged and therefore the next component will move to the last syllable as well. And so does the next and so on until information is moved over one syllable, but this means that this syllable is deleted up to a finite

part. As this syllable cannot be compared to the other syllables anymore it follows that this is not a successful way to accept L_{copy^*} either.

Thus, $\mathcal{M}$ cannot accept L_{copy^*}. □

Lemma 4.3.7 and Theorem 4.3.8 yield the following proper inclusions.

Corollary 4.3.9. *For all types* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}\}$,

$$\mathcal{L}_{=1}(\text{det-strict-CD-X}) \subsetneq \mathcal{L}_{=1}(\text{det-global-CD-X}).$$

It remains to show that this inclusion is also valid for CD-X- systems, $\mathsf{X} \in \{\mathsf{RW}, \mathsf{RRW}\}$. This is proven in Theorem 4.3.27 on page 99.

However, if we consider the language of only finitely many copies L_{copy^m}, then this language is accepted by a strictly deterministic CD-RW-system with only three components. We describe the method in the following lemma for L_{copy^3}.

Lemma 4.3.10. *The language* L_{copy^3} *is accepted by a strictly deterministic* CD-RW*-system with three components that executes meta-cycles of length 2.*

Proof. For simplicity we consider the sublanguage $L^{(6)}_{\text{copy}^3} := \{\, w\#w\#w\# \mid w \in (\Sigma_0^6)^n,\, n \geq 2 \,\}$ only. Let $\mathcal{M} := ((M_0, \{1\}), (M_1, \{2\}), (M_2, \{1\}), \{0\})$ be the CD-RW-system that is given through the following meta-instructions. Here $\Sigma_0 := \{a, b\}$ and $c, d, e, f, g, h, a' \in \Sigma_0$:

$$\begin{aligned}
M_0 \;:\;& (\text{¢} \cdot (\Sigma_0^6)^+ \cdot \# \cdot (\Sigma_0^6)^+ \cdot \# \cdot (\Sigma_0^6)^+, \#\$ \to \$),\\
M_1 \;:\;& (\text{¢} \cdot (\Sigma_0^6)^* \cdot cdefgh \cdot \# \cdot (\Sigma_0^6)^*, cdefgh \cdot \# \cdot a' \to \#\# \cdot fgh \cdot \# \cdot a'),\\
& (\text{¢} \cdot (\Sigma_0^6)^* \cdot cdefgh \cdot \#\# \cdot (\Sigma_0^3)^+ \cdot \# \cdot (\Sigma_0^6)^*, cdefgh \cdot \#\# \to \#\# \cdot fgh),\\
& (\text{¢} \cdot \#\# \cdot (\Sigma_0^3)^* \cdot cde \cdot \#\#\# \cdot (\Sigma_0^3)^+ \cdot \# \cdot (\Sigma_0^6)^+, cdefgh \cdot \$ \to \#\# \cdot fgh \cdot \$),\\
& (\text{¢} \cdot \#\# \cdot (\Sigma_0^3)^* \cdot cde \cdot \#\#\# \cdot (\Sigma_0^3)^+ \cdot \# \cdot (\Sigma_0^6)^*, cdefgh \cdot \#\# \to \#\# \cdot fgh),\\
& (\text{¢} \cdot \#\#\#\#\# \cdot (\Sigma_0^3)^* \cdot fgh \cdot \#\#\# \cdot (\Sigma_0^3)^*, fgh \cdot \$ \to \$),\\
& (\text{¢} \cdot \#^8 \cdot \$, \mathsf{Accept}),\\
M_2 \;:\;& (\text{¢} \cdot (\Sigma_0^6)^*, cdefgh \cdot \# \cdot a' \to \#\# \cdot cde \cdot \# \cdot a'),\\
& (\text{¢} \cdot (\Sigma_0^6)^*, cdefgh \cdot \#\# \to \#\# \cdot cde),\\
& (\text{¢} \cdot \#\# \cdot (\Sigma_0^3)^*, cde \cdot \#\#\# \to \#\#\#),\\
& (\text{¢} \cdot \#\#\#\#\# \cdot (\Sigma_0^3)^*, fgh \cdot \#\#\# \to \#\#\#).
\end{aligned}$$

Then M_0, M_1, and M_2 are deterministic RW-automata, and therewith $\mathcal{M}$ is a strictly deterministic CD-RW-system with three components that executes meta-cycles of length 2.

Claim. $L_{=1}(\mathcal{M}) = L^{(6)}_{\text{copy}^3}$.

Proof. Given an input of the form $w\#w\#w\#$, where $w = u_1v_1u_2v_2 \dots u_mv_m$ for some words $u_1, \dots, u_m, v_1, \dots, v_m \in \Sigma_0^3$ and an integer $m \geq 1$, $\mathcal{M}$ proceeds as follows:

1. First M_0 verifies that the given input is of the form $w_1\#w_2\#w_3\#$ for some $w_1, w_2, w_3 \in (\Sigma_0^6)^+$. In the negative it halts and rejects. In the affirmative it removes the last occurrence of # and restarts.

2. Then w_1 and w_2 are compared to each other, proceeding from right to left. In the i-th meta-cycle the factors $u_{m-i+1}v_{m-i+1}$ of both syllables are compared. In w_2 the factor u_{m-i+1} is erased, while in w_1 the factor v_{m-i+1} is erased. Further, the string ## is used in both syllables as a marker to distinguish the prefix that has not been processed yet from the suffix that has already been processed.

3. Next the descendant $w_1' := u_1 \dots u_m$ of w_1 is compared to the corresponding sequence of factors of w_3. This is again done from right to left, in the i-th meta-cycle deleting the factor u_{m-i+1} from both w_1' and w_3. Here again the string ## is used as a marker within the descendants of w_3.

4. Finally the descendant $w_2' := v_1 \dots v_m$ of w_2 is compared to the remaining descendant w_3' of w_3. This is again done from right to left, in the i-th meta-cycle deleting v_{m-i+1} from both w_2' and from w_3'. Here no markers are needed, because both words are deleted.

It follows that $\mathcal{M}$ accepts, that is, $L^{(6)}_{\mathrm{copy}^3} \subseteq L_{=1}(\mathcal{M})$.

Conversely, if $x \in (\Sigma_0 \cup \{\#\})^+$ is accepted by $\mathcal{M}$, then it follows from the form of the meta-instruction of M_0 that x is of the form $x = w_1\#w_2\#w_3\#$ for some words $w_1, w_2, w_3 \in (\Sigma_0^6)^+$. Further, the above description of how $\mathcal{M}$ works implies that $w_1 = w_2 = w_3$ must hold, that is, we see that $L^{(6)}_{\mathrm{copy}^3} = L_{=1}(\mathcal{M})$ □

This completes the proof of Lemma 4.3.10. □

Observe that in the above construction the component M_0 is used only once in each computation. It performs a kind of *head computation* that is used to ensure that the words accepted belong to the regular language $(\Sigma_0^6)^+ \cdot \# \cdot (\Sigma_0^6)^+ \cdot \# \cdot (\Sigma_0^6)^+ \cdot \#$. The necessary comparisons are all done by the components M_1 and M_2 that alternate. The same technique can be used to show the following result.

Proposition 4.3.11. *For each integer $m \geq 3$, the language L_{copy^m} is accepted by a strictly deterministic* CD-RW*-system with three components that executes meta-cycles of length 2.*

Here we just need to divide the syllables into $m-1$ factors such that each of them can be used in $m-1$ comparisons. This method also works for other variants of the copy language. For example even the prefix language L_{pre^m} from the next definition is accepted by a strictly deterministic CD-RW-system with five components and meta-cycles of length four.

Definition 4.3.12. *For $m \geq 2$, L_{pre^m} is the following variant of the m-fold copy language:*

$$L_{\mathrm{pre}^m} := \{\, w_1\#w_2\#\dots\#w_m \mid w_1, \dots, w_m \in \Sigma_0^*,\ w_m \leq_{\mathrm{pre}} w_i \text{ for all } 1 \leq i \leq m-1 \,\},$$

where $\leq_{\mathrm{pre}}$ is the prefix relation.

Up to now, we were unable to find a separation language for CD-RW-systems based on the number of components. And even for CD-RR-systems the language L_{copy^m} does not separate the class of CD-systems with m components from the one with $m-1$ components as Martin Plátek proved just recently.

Proposition 4.3.13. [Plá07]
For each integer $m \geq 6$, the language L_{copy^m} is accepted by a strictly deterministic CD-RR*-system with less than m components.*

However, strictly deterministic CD-systems working in mode t are even more expressive.

Proposition 4.3.14. *The language L_{copy^*} is accepted by a strictly deterministic* CD-R*-system working in mode* t.

Proof. Let $\mathcal{M} := ((M_i, \sigma_i)_{i\in I}, I_0)$ be the CD-R-system that is specified by $I := \{0,1,2\}$, $I_0 := \{0\}$, $\sigma_0 := \{1\}$, $\sigma_1 := \{2\}$, $\sigma_2 := \{0\}$, where the R-automata M_0, M_1, and M_2 are given through the following meta-instructions. Here $\Sigma_0 := \{a, b\}$ and $c, d, e \in \Sigma_0$:

$$\begin{aligned}
M_0 \quad &: \quad (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot cd \cdot \#)^+ \cdot (\Sigma_0^2)^+, cd \cdot \$ \to c\$),\\
&\qquad (\mathrm{¢} \cdot cd \cdot (\# \cdot cd)^+ \cdot \$, \mathsf{Accept}),\\
M_1 \quad &: \quad (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot \Sigma_0\#)^* \cdot (\Sigma_0^2)^+, cd \cdot \# \to c \cdot \#),\\
M_2 \quad &: \quad (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot \#)^* \cdot (\Sigma_0^2)^+, c \cdot \# \to \cdot\#),\\
&: \quad (\mathrm{¢} \cdot ((\Sigma_0^2)^+ \cdot \#)^* \cdot (\Sigma_0^2)^+, c \cdot \$ \to \cdot\$).
\end{aligned}$$

Obviously, the RW-automata M_0, M_1, and M_2 are deterministic, and hence, $\mathcal{M}$ is a strictly deterministic CD-RW-system. Given an input of the form $w(\#w)^n$, where $w \in (\Sigma_0^2)^+$ and $n \geq 1$, the automaton M_0 checks that all syllables end with the same suffix cd of length 2, and in the affirmative it rewrites the suffix cd of the last syllable into c. Now no meta-instruction of M_0 is applicable anymore, and therefore, M_1 takes over. It rewrites the suffix $cd\cdot\#$ of the first n syllables into $c\#$, that is, it yields the tape content $¢(w_1c\#))^n w_1 c\$$, where $w = w_1cd$. Then M_2 takes over, which deletes the last letter of each syllable, producing the tape content $¢w_1(\#w_1)^n\$$ within $n+1$ cycles. Then M_0 takes over again. It follows that $L_{\mathsf{t}}(\mathcal{M}) = L_{\text{copy}^*}$. □

4.3.3 Locally Deterministic CD-Systems

Here we will show that the expressive power of locally deterministic CD-systems of restarting automata differs from that of nondeterministic CD-systems on the one hand, and from that of globally deterministic CD-systems on the other hand. First, however, we establish the following inclusion result.

Theorem 4.3.15. *For each type* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, and each integer* $j \geq 1$*,* $\mathcal{L}_{=j}(\mathsf{det\text{-}global\text{-}CD\text{-}X}) \subseteq \mathcal{L}_{=1}(\mathsf{det\text{-}local\text{-}CD\text{-}X})$.

Proof. Let $j \geq 1$, and let $\mathcal{M} := ((M_i, \sigma_i)_{i\in I}, \{i_0\})$ be a globally deterministic CD-system of X-automata, where, for each $i \in I$, the set of states of M_i is $Q_i = \{q_{i,0}, q_{i,1}, \ldots, q_{i,n_i}\}$. Then, for each index $i \in I$, M_i is a deterministic X-automaton, and each restart operation $\delta_i(q_{i,r}, u) = \mathsf{Restart}$ of M_i is combined with an index $l \in \sigma_i$. This means that the automaton M_l takes over, when M_i finishes a part of a computation of $\mathcal{M}$ according to the actual mode of operation $= j$ by executing this particular restart operation. To express this combination, we will denote the above restart operation as $\delta_i(q_{i,r}, u) = (\mathsf{Restart}, l)$. We have to construct a locally deterministic CD-system $\mathcal{M}'$ of X- automata such that $L_{=1}(\mathcal{M}') = L_{=j}(\mathcal{M})$ holds.

We first consider the case $j = 1$. Then $\mathcal{M}$ switches to the next component system whenever a restart operation is executed. The CD-system $\mathcal{M}'$ is defined as $\mathcal{M}' := ((M_i^{(l)}, \sigma_i^{(l)})_{(i,l)\in I'}, I_0')$, where $I' := \bigcup_{i\in I}(\{i\} \times \sigma_i)$, $I_0' := \{i_0\} \times \sigma_{i_0}$, $\sigma_i^{(l)} := \{l\} \times \sigma_l$, and $M_i^{(l)}$ is essentially a copy of the

X-automaton M_i for each $i \in I$ and each $l \in \sigma_i$. However, all restart operations of M_i are deleted from $M_i^{(l)}$ but those of the form $\delta_i(q,u) = (\mathsf{Restart}, l)$, which are changed into $\delta_i^{(l)}(q,u) = \mathsf{Restart}$. Of course, the states of $M_i^{(l)}$ are renamed in such a way that all these automata have disjoint sets of states.

It is obvious that each automaton $M_i^{(l)}$ is a deterministic X-automaton. Thus, $\mathcal{M}'$ is a locally deterministic CD-X-system. It remains to show that $L_{=1}(\mathcal{M}') = L_{=1}(\mathcal{M})$ holds.

Claim 1. $L_{=1}(\mathcal{M}) \subseteq L_{=1}(\mathcal{M}')$.

Proof. Let $w \in \Sigma^*$ be an input word that belongs to the language $L_{=1}(\mathcal{M})$. Then the computation of $\mathcal{M}$ on input w has the following form:

$$w \vdash^c_{M_{l_0}} w_1 \vdash^c_{M_{l_1}} \cdots \vdash^c_{M_{l_{m-1}}} w_m \vdash^c_{M_{l_m}} \mathsf{Accept},$$

where, for all $i = 1, \ldots, m$, the successor $l_i \in \sigma_{l_{i-1}}$ is combined with the restart step that $M_{l_{i-1}}$ executes in the cycle $w_{i-1} \vdash^c_{M_{l_{i-1}}} w_i$. This computation can be simulated by $\mathcal{M}'$ as follows:

$$w \vdash^c_{M_{l_0}^{(l_1)}} w_1 \vdash^c_{M_{l_1}^{(l_2)}} \cdots \vdash^c_{M_{l_{m-1}}^{(l_m)}} w_m \vdash^c_{M_{l_m}^{(l')}} \mathsf{Accept},$$

where $l' \in \sigma_{l_m}$. Observe that $M_{l_0}^{(l_1)}$ contains the restart operation that corresponds to the restart operation $(\mathsf{Restart}, l_1)$ of M_{i_0} executed in the first cycle of the $\mathcal{M}$-computation above, and analogously for the other components. Thus, it follows that $L_{=1}(\mathcal{M}) \subseteq L_{=1}(\mathcal{M}')$. □

Claim 2. $L_{=1}(\mathcal{M}') \subseteq L_{=1}(\mathcal{M})$.

Proof. Let $w \in \Sigma^*$ be an input word that belongs to the language $L_{=1}(\mathcal{M}')$. Then on input w, $\mathcal{M}'$ can execute an accepting computation of the following form:

$$w \vdash^c_{M_{l_0}^{(l_1)}} w_1 \vdash^c_{M_{l_1}^{(l_2)}} \cdots \vdash^c_{M_{l_{m-1}}^{(l_m)}} w_m \vdash^c_{M_{l_m}^{(l_{m+1})}} \mathsf{Accept},$$

where, for all $i = 1, \ldots, m+1$, $l_i \in \sigma_{l_{i-1}}$. It follows immediately from the construction of $\mathcal{M}'$ that this computation corresponds to an accepting computation of $\mathcal{M}$. Hence, it follows that $L_{=1}(\mathcal{M}') \subseteq L_{=1}(\mathcal{M})$. □

Together Claims 1 and 2 prove that $L_{=1}(\mathcal{M}') = L_{=1}(\mathcal{M})$ holds.

For $j > 1$, $\mathcal{M}$ can be simulated by a locally deterministic CD-X-system that uses j copies of each component of $\mathcal{M}$ to count the number of cycles that each component executes (compare the proof of Theorem 4.3.3). The first $j-1$ copies are true copies of the respective component of $\mathcal{M}$, while the j-th copy is modified as in the proof above for mode = 1. Thus only the restart operations that are assigned to the chosen successor remain in the component. □

This result together with Theorem 4.3.2 and Theorem 4.3.4 leads to the following Corollary.

Corollary 4.3.16. *For each type* $\mathsf{X} \in \{\mathsf{RR}, \mathsf{RL}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RRWW}, \mathsf{RLWW}\}$ *the following inclusions hold:* $\mathcal{L}_{\mathsf{t}}(\mathsf{det\text{-}global\text{-}CD\text{-}X}) \subseteq \mathcal{L}_{=1}(\mathsf{det\text{-}local\text{-}CD\text{-}X})$.

Proof. From Theorem 4.3.4 we know that $\mathcal{L}_{\mathsf{t}}(\mathsf{det\text{-}global\text{-}CD\text{-}X}) \subseteq \mathcal{L}(\mathsf{det\text{-}nf\text{-}X})$, and in Theorem 4.3.2 it is shown that $\mathcal{L}(\mathsf{det\text{-}nf\text{-}X}) \subseteq \mathcal{L}_{=1}(\mathsf{det\text{-}global\text{-}CD\text{-}X})$.

Thus, $\mathcal{L}_{\mathsf{t}}(\mathsf{det\text{-}global\text{-}CD\text{-}X}) \subseteq \mathcal{L}_{=1}(\mathsf{det\text{-}global\text{-}CD\text{-}X})$ holds and with Theorem 4.3.15 the assertion follows. □

It remains open whether a corresponding result holds for locally deterministic CD-systems working in mode t or =j. The problem in both cases is that for globally deterministic CD-systems each restart is assigned with a successor. This successor is only used, if the component is changed. But a component in a locally deterministic CD-system does not know, when it is executing its last cycle.

Next we turn to the separation results that were announced at the beginning of this subsection. To establish the first of them we consider the following example language.

Definition 4.3.17. *Let $\Sigma_0 := \{a, b\}$ and $\Sigma_1 := \{a, b, c\}$. The language L_{cc} is a variant of the copy language: $L_{\mathsf{cc}} := \{\, wc\phi_1(w)c\phi_2(w) \mid w \in \Sigma_1^* \,\}$, where $\phi_1 : \Sigma_1^* \to \Sigma_0^*$ is the morphism that is defined by $a \mapsto a$, $b \mapsto b$, and $c \mapsto a$, and $\phi_2 : \Sigma_1^* \to \Sigma_0^*$ is the morphism that is defined by $a \mapsto a$, $b \mapsto b$, and $c \mapsto b$.*

Lemma 4.3.18. *The language L_{cc} is accepted by a* CD-RR*-system consisting of three components that is working in mode $= 1$.*

Proof. Let $\mathcal{M} := ((M_1, \{2\}), (M_2, \{3\}), (M_3, \{1\}), \{1\})$ be the following CD-RR-system with input alphabet Σ_1, where the RR-automata M_1, M_2, and M_3 are given through the following meta-instructions:

$$\begin{aligned}
M_1 \quad &: \quad (\mathrm{¢} \cdot \Sigma_1^* \cdot ac \cdot \Sigma_0^* \cdot ac \cdot \Sigma_0^*, a \cdot \$ \to \$, \varepsilon),\\
&\qquad (\mathrm{¢} \cdot \Sigma_1^* \cdot bc \cdot \Sigma_0^* \cdot bc \cdot \Sigma_0^*, b \cdot \$ \to \$, \varepsilon),\\
&\qquad (\mathrm{¢} \cdot \Sigma_1^* \cdot cc \cdot \Sigma_0^* \cdot ac \cdot \Sigma_0^*, b \cdot \$ \to \$, \varepsilon),\\
&\qquad (\mathrm{¢} \cdot cc \cdot \$, \mathsf{Accept}),\\
M_2 \quad &: \quad (\mathrm{¢} \cdot \Sigma_1^+ \cdot c \cdot \Sigma_0^*, xc \to c, \Sigma_0^* \cdot \$) \text{ for all } x \in \Sigma_0,\\
M_3 \quad &: \quad (\mathrm{¢} \cdot \Sigma_1^*, yc \to c, \Sigma_0^* \cdot c \cdot \Sigma_0^* \cdot \$) \text{ for all } y \in \Sigma_1.
\end{aligned}$$

Given an input $z \in \Sigma_1^*$, it is obvious that M_1 will immediately reject, if $|z|_c \leq 1$. Thus, assume that w has the form $z = ucvcw$, where $u \in \Sigma_1^*$ and $v, w \in \Sigma_0^*$. Observe that $|u| = |v| = |w|$ must hold if $w = ucvcw$ belongs to the language L_{cc}. If $u = v = w = \varepsilon$, then M_1 accepts immediately. Observe that $w = cc$ belongs to the language L_{cc}. If only one or two of the factors u, v, or w are empty, then M_1 rejects immediately. Otherwise, M_1 compares the last letter of u, say d, to the last letter of v, say e, and the last letter of w, say f. If $\phi_1(d) = e$ and $\phi_2(d) = f$, then f is deleted, and M_2 becomes active; otherwise, M_1 halts and rejects. In the latter case $w = ucvcw$ does not belong to L_{cc}, while in the former case M_2 simply deletes the letter e, and then M_3 deletes the letter d. Thus, $\mathcal{M}$ has executed the sequence of cycles $z = ucvcw = u_1dcv_1ecw_1f \vdash_{\mathcal{M}}^{c^3} u_1cv_1cw_1$. Now $z \in L_{cc}$ if and only if $u_1cv_1cw_1 \in L_c$, and hence, it follows inductively that $L_{=1}(\mathcal{M}) = L_{cc}$. □

In contrast to this positive result the following negative result was achieved in [Ott08].

Proposition 4.3.19. *The language L_{cc} is not accepted by any locally deterministic* CD-RRW*-system that is working in mode $= 1$.*

Combined Lemma 4.3.18 and Proposition 4.3.19 yield the following consequence.

Corollary 4.3.20. *For the types* $\mathsf{X} \in \{\mathsf{RR}, \mathsf{RRW}\}$,

$$\mathcal{L}_{=1}(\mathsf{det\text{-}local\text{-}CD\text{-}X}) \subsetneq \mathcal{L}_{=1}(\mathsf{CD\text{-}X}) = \mathcal{L}(\mathsf{nf\text{-}X}).$$

Next we consider the language $L_{(\mathsf{expo,pal})} := L_{\mathrm{pal}} \cup L_{\mathrm{expo}}$, where

$$\begin{array}{lcll} L_{\mathrm{pal}} & := & \{\, w \mid w \in \Sigma_0^*,\, w = w^R \,\}, \text{ and} & \\ L_{\mathrm{expo}} & := & \{\, a^{i_0} b a^{i_1} b \cdots a^{i_{n-1}} b a^{i_n} & \mid \; n \geq 0,\, i_0, \ldots, i_n \geq 0, \text{ and} \\ & & & \exists m \geq 0 : \sum_{j=0}^{n} 2^j \cdot i_j = 2^m \,\} \cup b^*. \end{array}$$

Lemma 4.3.21. *The language* $L_{(\mathsf{expo,pal})}$ *is accepted by a locally deterministic* CD-RW*-system working in mode* $= 1$.

Proof. Let $\mathcal{M} = ((M_i, \sigma_i)_{i \in I}, I_0)$ be the CD-RW-system that is specified by $I := \{1,2,3,4\}$, $I_0 := \{1,3\}$, $\sigma_1 := \{2\}$, $\sigma_2 := \{1\}$, $\sigma_3 := \{4\}$, and $\sigma_4 := \{3\}$, where the components M_i, $1 \leq i \leq 3$, are given through the following meta-instructions:

$$\begin{array}{lcll} M_1 & : & (\mathrm{¢} \cdot d \cdot \Sigma_0^*, d \cdot \$ \to \$) & \text{for all } d \in \Sigma_0, \\ & & (\mathrm{¢} \cdot \{\varepsilon, a, b\} \cdot \$, \mathsf{Accept}), & \\ M_2 & : & (\mathrm{¢}, d \to \varepsilon) & \text{for all } d \in \Sigma_0, \\ M_3 & : & (\mathrm{¢} \cdot a^*, aab \to ba), & \\ & & (\mathrm{¢}, b \to \varepsilon), & \\ & & (\mathrm{¢} \cdot a^*, a^4 \cdot \$ \to baa \cdot \$), & \\ & & (\mathrm{¢} \cdot \{\varepsilon, a, aa, ab^+\} \cdot \$, \mathsf{Accept}), & \end{array}$$

and M_4 is simply a copy of M_3. Obviously, these RW-automata are all deterministic, that is, $\mathcal{M}$ is indeed a locally deterministic CD-RW-system. The subsystem consisting of M_1 and M_2 is easily seen to accept the language L_{pal}. Thus, $L_{=1}(\mathcal{M}) = L_{(\mathsf{expo,pal})}$ is an immediate consequence of the following claim:

Claim. M_3 and M_4 together accept the language L_{expo}.

Proof. M_3 and M_4 are identical, and they simply alternate. If $w = b^m$ for some $m \geq 0$, then obviously w is accepted by M_3 and M_4. If $w = a^{i_0} b a^{i_1} b \cdots a^{i_{n-1}} b a^{i_n}$ is given as input to M_3, where $n, i_0, \ldots, i_n \geq 0$, and $\sum_{j=0}^{n} 2^j \cdot i_j = 2^m$ for some $m \geq 0$, then the first occurrence of b is first shifted to the left end of the word. As $\sum_{j=0}^{n} 2^j \cdot i_j = 2^m$, we see that i_0 is an even number. Thus, this particular occurrence of b is then deleted. This results in the word $w_1 := a^{i_0/2+i_1} b \cdots a^{i_{n-1}} b a^{i_n}$. As

$$i_0/2 + i_1 + \sum_{j=2}^{n} 2^{j-1} \cdot i_j = (\sum_{j=0}^{n} 2^j \cdot i_j)/2 = 2^{m-1},$$

we see that this word belongs to the language L_{expo}. This continues until all occurrences of the letter b have been deleted, or until a word of the form ab^m is reached. The resulting word is of the form a^{2^l} for some $l \geq 0$. If $l \leq 1$, then the word is accepted, otherwise an occurrence of b is generated at the right end of the word and shifted through the word as described above, which results in the word $a^{2^{l-1}}$. Now the claim follows easily. □

□

In contrast to this positive result we have the following.

Theorem 4.3.22. *The language $L_{(\mathsf{expo},\mathsf{pal})}$ is not accepted by any globally deterministic* CD-RRW*-system that is working in mode $= 1$.*

Proof. For a contradiction let $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$ be a globally deterministic CD-RRW-system with window size k that accepts the language $L_{(\mathsf{expo},\mathsf{pal})}$ in mode $= 1$, and let $w := u_1{u_1}^R u u_1{u_1}^R \in L_{(\mathsf{expo},\mathsf{pal})}, c \geq 0$ be an input word for which the words u, u_1 have sufficient length and let u and u_1 both contain every string of length k over Σ_0 arbitrary often and in any ordering. Then the computation of $\mathcal{M}$ on input w is accepting. As $\mathcal{M}$ has no auxiliary symbols and the input symbols are only a and b, it is impossible for it to find and mark the middle, because there is no string left, which could be used as a marker. As each component M_i is deterministic, it can only distinguish between $|Q_i| * |\Sigma|^k$ many configurations. Thus u_1 can be chosen in a way that each M_i must perform a rewrite in a prefix of u_1, if it wants to perform a rewrite before it sees the right sentinel \$. So during the first m cycles of this computation each of the M_i and therefore $\mathcal{M}$, can only perform rewrites in a prefix of restricted length of w and therefore of u_1 or while seeing the right border marker \$.

Assume that $w \vdash_{\mathcal{M}}^{c^m} v$, and assume that m is sufficiently large such that the information erased from the tape during these m cycles cannot be remembered by selecting an appropriate component system of $\mathcal{M}$. There exists such a m, because $\mathcal{M}$ is globally deterministic, and information can only be remembered by choosing a component, but $\mathcal{M}$ has only a finite number of components. Information cannot be stored on the tape, if the word u_1 has a high Kolmogorov complexity,

By executing these m cycles, it is in general not possible to transform a word w belonging to L_{pal} into a word v from L_{expo} or vice versa. More important there exist subwords u such that $w = u_1{u_1}^R u u_1{u_1}^R$ is such a word. If u is not a palindrome and it is large enough, then w cannot be transformed into a palindrome either, as u is not altered during these cycles. If $w \notin L_{\mathsf{expo}}$ holds, then there also exist subwords $u \in \{b^n a \Sigma_0^* | n \geq 0\}$ such that w cannot be transformed into a word belonging to L_{expo} in these m cycles. Because the factor u_1 needs to contain at least 2^n a's, if $u_1 u u_1^R$ can be rewritten into a word belonging to L_{expo}.

Also, based on the information gathered during this process, $\mathcal{M}$ cannot decide whether it has to verify that w belongs to the sublanguage L_{pal} or whether it has to verify that w belongs to the sublanguage L_{expo}. This is a property that also depends on the middle part u. Thus, modulo the finite information that $\mathcal{M}$ can store by choosing a component, v must essentially belong to the same sublanguage as the original input w. This means that the first m cycles must preserve this property. Therefore $\mathcal{M}$ must check $w \in L_{\text{pal}}$ and $w \in L_{\text{expo}}$ simultaneously.

To retain the property of being a palindrome for the tape content, symbols in the prefix are compared to their counterparts in the suffix, and both are deleted in order to distinguish those parts that have been compared from those parts that have not yet been compared.

There is no other way than deleting or rewriting, and thereby shortening, the compared parts. The rewritten parts cannot be an encoding of the original tape content, as there are no markers. However, this process will in general destroy the property of the tape content to belong to the sublanguage L_{expo}.

If $\mathcal{M}$ does not remember how many letters it has deleted, then together with a word of L_{expo} it also accepts some words not belonging to this language. This contradicts the error preserving property. On the other hand, if rewrites are performed that are oriented towards checking membership in L_{expo}, then $\mathcal{M}$ has to move b's from left to right over the tape, while halving the a's. But this destroys the symmetry of palindromes.

In the following paragraphs we give details, about what $\mathcal{M}$ can do in every rewrite and discuss, how this effects the property of belonging to one of the sublanguages. However a finite number of these steps, which do not preserve this property, are always possible, as long as $\mathcal{M}$ remembers where these steps were made. But as only a finite number of steps are possible, we will show that $\mathcal{M}$ is unable to check the belonging to both sublanguages simultaneously for an arbitrary large number of cycles.

In every rewrite there are four options what to do with the b's: creating new b's, deleting b's, moving b's and doing nothing with the b's.

Doing nothing with the b's means that some a's are deleted, this can be helpful for checking membership in L_{pal} (if it is done in a position near the border markers and if the a's are deleted together with their counterparts), but this destroys the property of belonging to L_{expo}.

Deleting b's is allowed at the border markers. Deletes in any other place destroy the property of belonging to L_{expo}. For L_{pal} it is allowed, if they are deleted together with their counterparts. To find the counterparts they must be in a position near the border markers.

b's can only be created, if some a's are deleted, because each restarting automaton is length reducing. Therefore, while checking for L_{expo}, b's can only be created near the right border marker. In this process the a's that are rewritten are halved and written to the right of the b. An example is the rewrite operation $a^4\$ \to ba^2\$$. For L_{pal} this is allowed, if $\¢a^4 \to \¢a^2b$ is rewritten in another cycle at the beginning of the tape, but this is not allowed while checking for L_{expo}.

The last thing is moving b's. If a b is moved from right to left while halving the a's, which the b moves over, this is no problem, but as each rewrite is length reducing, movements from left to right destroy the property of belonging to L_{expo}. But while checking for L_{pal}, movements are only allowed, if they have different directions in the prefix and in the suffix.

Hence, in none of these cases, except for deleting b's at the borders, it is possible to check for membership in both sublanguages simultaneously. It follows that with $L_{(\mathsf{expo},\mathsf{pal})}$ the system $\mathcal{M}$ will also accept some words that do not belong to this language. □

By applying the encoding $\phi : \Sigma_0^* \to \Sigma_0^*$ from above we obtain the language $L'_{(\mathsf{expo},\mathsf{pal})} := \phi(L_{(\mathsf{expo},\mathsf{pal})})$.

Lemma 4.3.23. *The language $L'_{(\mathsf{expo},\mathsf{pal})}$ is accepted by a locally deterministic* CD-R*-system working in mode* $= 1$.

Proof. Let $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$ be the CD-R-system that is specified by $I := \{1,2,3,4\}$, $I_0 := \{1,3\}$, $\sigma_1 := \{2\}$, $\sigma_2 := \{1\}$, $\sigma_3 := \{4\}$, and $\sigma_4 := \{3\}$, where the components M_i, $1 \le i \le 3$, are given through the following meta-instructions:

$$
\begin{array}{lcll}
M_1 & : & (\¢ \cdot d \cdot \phi(\Sigma_0)^*, d \cdot \$ \to \$) & \text{for all } d \in \phi(\Sigma_0), \\
 & & (\¢ \cdot \{\varepsilon, ab, b\} \cdot \$, \mathsf{Accept}), & \\
M_2 & : & (\¢, d \to \varepsilon) & \text{for all } d \in \phi(\Sigma_0), \\
M_3 & : & (\¢ \cdot (ab)^*, ababb \to bab), & \\
 & & (\¢, b \to \varepsilon), & \\
 & & (\¢ \cdot (ab)^*, (ab)^4 \cdot \$ \to babab \cdot \$), & \\
 & & (\¢ \cdot \{\varepsilon, ab, abab, ab \cdot b^+\} \cdot \$, \mathsf{Accept}), &
\end{array}
$$

and M_4 is simply a copy of M_3. Obviously, these R-automata are all deterministic, that is, $\mathcal{M}$ is indeed a locally deterministic CD-R-system. The subsystem consisting of M_1 and M_2 is easily seen to accept the language $\phi(L_{\text{pal}})$, and it can be shown that M_3 and M_4 together accept the

language $\phi(L_{\text{expo}})$. This follows from the claim in the proof of Lemma 4.3.21 and the fact that the R-automata M_3 and M_4 above are obtained from the corresponding RW-automata from that proof by applying the encoding ϕ to all meta-instructions. □

Theorem 4.3.24. *The language $L'_{(\mathsf{expo},\mathsf{pal})}$ is not accepted by any globally deterministic* CD-RR*-system that is working in mode = 1.*

Proof. Let $\mathcal{M} = ((M_i, \sigma_i)_{i \in I}, I_0)$ be a globally deterministic CD-RR-system that accepts the language $L'_{(\mathsf{expo},\mathsf{pal})}$ in mode = 1, and let $w := (ab)^{i_1} b^{l_1} \cdots (ab)^{i_j} b^{l_j} u b^{l_j} (ab)^{i_j} \cdots b^{l_1} (ab)^{i_1} \in L'_{(\mathsf{expo},\mathsf{pal})}$ be an input word for which the numbers $j, i_1, \ldots, i_j, l_1, \ldots, l_j$, and $|u|$ are sufficiently large.

Then the computation of $\mathcal{M}$ on input w is accepting. As $\mathcal{M}$ is deterministic, it can only process prefixes and suffixes of restricted length of w during the first m cycles of this computation. Assume that $w \vdash_{\mathcal{M}}^{c^m} v$, and assume that m is sufficiently large such that the information erased from the tape during these m cycles cannot be remembered by selecting an appropriate component system of $\mathcal{M}$. By executing these m cycles, it is in general not possible to transform a word w belonging to $\phi(L_{\text{pal}})$ into a word v from $\phi(L_{\text{expo}})$ or vice versa. Also, based on the information gathered during this process, $\mathcal{M}$ cannot decide whether it has to verify that w belongs to the sublanguage $\phi(L_{\text{pal}})$, or whether it has to verify that w belongs to the sublanguage $\phi(L_{\text{expo}})$. Thus, modulo the finite information that $\mathcal{M}$ can store by choosing a component, v must essentially belong to the same sublanguage as the original input w. This means that the first m cycles must preserve this property.

To preserve the property of the tape content of being a palindrome, symbols in the prefix are compared to their counterparts in the suffix. In the palindrome language L_{pal}, symbols must be deleted to distinguish those parts that have already been checked from those parts that have not. For the language $\phi(L_{\text{pal}})$, however, the situation is somewhat more involved. Here it helps that $\phi(b^i) = b^i$ for all $i \geq 1$, which implies that the only way to compare large blocks of b's consists in deleting some b's from the prefix and from the suffix, as all components of $\mathcal{M}$ are just deterministic RR-automata. However, this process will in general destroy the property of the tape content to belong to the sublanguage $\phi(L_{\text{expo}})$. Further, it is not possible either to first check all a's and then compare the blocks of b's, since $\mathcal{M}$ cannot move a marker from right to left across the blocks of b's.

On the other hand, if rewrites are performed that are oriented towards checking membership in $\phi(L_{\text{expo}})$, then the symmetry of palindromes is destroyed. This follows from the fact that the process of deleting at the left and at the right border is no sensible strategy for accepting the sublanguage $\phi(L_{\text{expo}})$ (with the exception explained in the proof of Theorem 4.3.22). Here again the encoding does not help, since the only possible markers are sequences with aa as prefix and as suffix, and these sequences cannot be created within large blocks of b's. Therefore, $\mathcal{M}$ cannot keep all information on a possible palindrome (that is, only deleting symbols in combination with their counterparts) and check at the same time whether the tape content belongs to the sublanguage $\phi(L_{\text{expo}})$.

Thus, in either case information is lost that is necessary to verify membership in the corresponding sublanguage. It follows that with $L'_{(\mathsf{expo},\mathsf{pal})}$ the system $\mathcal{M}$ will also accept some words that do not belong to this language. □

Thus, we have the following separation result.

Corollary 4.3.25. *For all types* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}\}$,

$$\mathcal{L}(\mathsf{nf\text{-}det\text{-}X}) = \mathcal{L}_{=1}(\mathsf{det\text{-}global\text{-}CD\text{-}X}) \subsetneq \mathcal{L}_{=1}(\mathsf{det\text{-}local\text{-}CD\text{-}X}).$$

Now we consider the following variation of the language $L_{(\mathsf{expo,pal})}$:

$\tilde{L}_{(\mathsf{expo,pal})} := L_{\mathrm{pal}'} \cup L_{\mathrm{expo}'}$, where

$$\begin{array}{lcll} L_{\mathrm{pal}'} & := & \{\, awa \mid w \in \Sigma_0^*,\, w = w^R \,\}, \text{ and} & \\ L_{\mathrm{expo}'} & := & \{\, a^{i_0}ba^{i_1}b\cdots a^{i_{n-1}}ba^{i_n}\cdot b & \mid \quad n \geq 0,\, i_0, \ldots, i_n \geq 0, \text{ and} \\ & & & \quad \exists m \geq 0 : \sum_{j=0}^{n} 2^j \cdot i_j = 2^m \,\} \cup b^+. \end{array}$$

The difference to the original language is that a word belonging to $L_{\mathrm{pal}'}$ always end with an a and that a word belonging to $L_{\mathrm{expo}'}$ always end with a b. Thus a globally deterministic CD-RW-system can move to the right border to determine which sublanguage to check.

Lemma 4.3.26. *The language* $\tilde{L}_{(\mathsf{expo,pal})}$ *is accepted by a globally deterministic* CD-RW*-system working in mode* $= 1$.

Proof. Let $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$ be the CD-RW-system that is specified by $I := \{0,1,2,3,4\}$, $I_0 := \{0\}$, $\sigma_0 := \{2,3\}$, $\sigma_1 := \{2\}$, $\sigma_2 := \{1\}$, $\sigma_3 := \{4\}$, and $\sigma_4 := \{3\}$, where the components M_i, $0 \leq i \leq 3$, are given through the following meta-instructions:

$$\begin{array}{lcll} M_0 & : & (\mathrm{¢} \cdot a \cdot \Sigma_0^*, a \cdot \$ \to \$, \mathsf{Restart}(2)), & \\ & & (\mathrm{¢} \cdot \Sigma_0^+, b \cdot \$ \to \$, \mathsf{Restart}(3)), & \\ M_1 & : & (\mathrm{¢} \cdot d \cdot \Sigma_0^*, d \cdot \$ \to \$, \mathsf{Restart}(2)) & \text{for all } d \in \Sigma_0, \\ & & (\mathrm{¢} \cdot \{\varepsilon, a, b\} \cdot \$, \mathsf{Accept}), & \\ M_2 & : & (\mathrm{¢}, d \to \varepsilon, \mathsf{Restart}(1)) & \text{for all } d \in \Sigma_0, \\ M_3 & : & (\mathrm{¢} \cdot a^*, aab \to ba, \mathsf{Restart}(4)), & \\ & & (\mathrm{¢}, b \to \varepsilon, \mathsf{Restart}(4)), & \\ & & (\mathrm{¢} \cdot a^*, a^4 \cdot \$ \to baa \cdot \$, \mathsf{Restart}(4)), & \\ & & (\mathrm{¢} \cdot \{\varepsilon, a, aa, ab^+\} \cdot \$, \mathsf{Accept}), & \end{array}$$

and M_4 is a copy of M_3 which always assigns 3 with its restarts. Obviously, these RW-automata are all deterministic, that is, $\mathcal{M}$ is indeed a globally deterministic CD-RW-system. In fact only the component M_0 has more than one successor. This component is only used once in a computation to look which sublanguage to check. A computation starting with M_0 and then continuing in the subsystem consisting of M_1 and M_2 is easily seen to accept only words that belong to the language L_{pal}. Thus, $L_{=1}(\mathcal{M}) = \tilde{L}_{(\mathsf{expo,pal})}$ as starting with M_0, M_3 and M_4 together accept the language $L_{\mathrm{expo}'}$.

M_0 transfers a word belonging to $L_{\mathrm{expo}'}$ into a word belonging to L_{expo} by deleting the last b. In the proof of Lemma 4.3.21 on page 95 it is shown that M_3 and M_4 together accept the language L_{expo}. □

Theorem 4.3.27. *The language* $\tilde{L}_{(\mathsf{expo,pal})}$ *is not accepted by any strictly deterministic* CD-RRW*-system that is working in mode* $= 1$.

Proof. Let $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$ be a strictly deterministic CD-RRW-system that accepts the language $\tilde{L}_{(\mathsf{expo,pal})}$ in mode $= 1$,

In the proof of Theorem 4.3.22 it was shown, that there are words in $L_{(\mathsf{expo,pal})}$ such that an arbitrary large prefix does not allow to determine to which sublanguage the word belongs. For $w \in \tilde{L}_{(\mathsf{expo,pal})}$ the same is true, we only need to guarantee that w starts with the letter a.

In the proof of Theorem 4.3.22 is was also shown that it is impossible to check for the belonging in L_{pal} and L_{expo} simultaneously. Therefore it is impossible for $\mathcal{M}$ to check for the belonging in $L_{\mathrm{pal}'}$ and $L_{\mathrm{expo}'}$ simultaneously in a prefix of w. Thus $\mathcal{M}$ has to move to the right border of the tape in order to determine which sublanguage to check. But at the right border $\mathcal{M}$ cannot move back and as there are only a's and b's on the tape $\mathcal{M}$ cannot move the information from right to left over the tape. As $\mathcal{M}$ is strictly deterministic each component has a fixed successor and therefore it is also impossible to transfer information in that way. Therefore the next component has the same problem, it can either perform a rewrite in the prefix and does not know which sublanguage to test, or it can perform a rewrite in the suffix, while seeing the right border marker in its window.

In the prefix $\mathcal{M}$ can perform as many rewrites as it can remember and only at places on the tape it can remember, but this does not help for accepting w. $\mathcal{M}$ cannot create markers, because it has no auxiliary symbols and the only tape symbols are a and b.

Thus, $\mathcal{M}$ has to know which sublanguage to check, but this means, that it performs its rewrites at the right border marker. In Theorem 4.2.2 it is shown that a CD-RRWW-system working in mode $= 1$ can be simulated cycle by cycle by a nonforgetting RRWW-automaton. From Lemma 3.3.3 we know, that a nonforgetting RRWW-automaton that performs all its rewrites at the right border marker only accepts regular languages. Thus even nondeterministic CD-RRWW-systems working in mode $= 1$ only accept regular languages, when they perform all of their rewrites at the right border of the tape.

Thus $\mathcal{M}$ cannot accept $\tilde{L}_{(\mathsf{expo,pal})}$ in mode $= 1$. □

Proposition 4.3.5 together with the Corollaries 4.3.20, 4.3.9 and 4.3.25 and Theorem 4.3.27 show that, for each type $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RW}, \mathsf{RRW}\}$, we have the following chain of proper inclusions:

$$\mathcal{L}(\mathsf{det\text{-}X}) \subsetneq \mathcal{L}_{=1}(\mathsf{det\text{-}strict\text{-}CD\text{-}X}) \subsetneq \mathcal{L}_{=1}(\mathsf{det\text{-}global\text{-}CD\text{-}X}) \subsetneq \mathcal{L}_{=1}(\mathsf{det\text{-}local\text{-}CD\text{-}X})$$

and

$$\mathcal{L}_{=1}(\mathsf{det\text{-}local\text{-}CD\text{-}RR(W)}) \subsetneq \mathcal{L}_{=1}(\mathsf{CD\text{-}RR(W)}).$$

Thus it remains open, whether the inclusions $\mathcal{L}_{=1}(\mathsf{det\text{-}local\text{-}CD\text{-}R(W)}) \subset \mathcal{L}_{=1}(\mathsf{CD\text{-}R(W)})$ are strict or not.

4.4 Shrinking CD-Systems of Restarting Automata

Shrinking restarting automata form a very robust language class. In Section 2.7 it is shown that $\mathcal{L}(\mathsf{nf\text{-}sh\text{-}RLWW})$ equals $\mathcal{L}(\mathsf{sh\text{-}RRWW})$, and that multiple rewrites per cycle do not add expressive power. Here we first show that all simulation results between CD-systems and nonforgetting restarting automata also hold for CD-systems of shrinking restarting automata. With these results we can extend the results from Chapter 3 to CD-systems of restarting automata.

Definition 4.4.1. *A* CD-system of shrinking RLWW-automata*,* CD-sh-RLWW*-system for short, is a* CD*-system, where each component is a shrinking* RLWW*-automaton and the weight functions of the component automata coincides.*

It is important that the weight functions of the component automata in a CD-system are coordinated. If each component had its own weight function, then computations of the system would in general not be terminating.

With the restriction to only one weight function in a CD-system, the comparison between nonforgetting restarting automata and CD-systems carry over to the shrinking case.

Theorem 4.4.2. *If M is a (deterministic) shrinking nonforgetting restarting automaton of type* X *for any type* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, then there exists a (globally deterministic) CD-system $\mathcal{M}$ of shrinking restarting automata of type* X *such that $L_{=1}(\mathcal{M}) = L(M)$ holds.*

Proof. $\mathcal{M}$ is constructed, as in the proof of Theorem 4.2.1. The weight function of M is taken as weight function of $\mathcal{M}$. In the proof of Theorem 4.2.1 the nonforgetting restarting automaton M is simulated cycle by cycle, thus each rewrite of the CD-system equals the rewrite of the nonforgetting restarting automaton. Thus if M is a shrinking nonforgetting restarting automaton, then $\mathcal{M}$ is a CD-system of shrinking restarting automata. □

The converse is also valid for shrinking restarting automata. The proof is done similarly. Remark that the system has only one weight function and that the determinism of the mode must be considered as well.

Theorem 4.4.3. *Let $j \geq 1$, and let $\mathsf{m} \in \{=j, \leq j, \geq j\}$ be a mode of operation. Then, for each* CD-sh-X*-system $\mathcal{M}$, where* $\mathsf{X} \in \{\mathsf{R}, \mathsf{RR}, \mathsf{RL}, \mathsf{RW}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RWW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, there exists a shrinking non-forgetting* X*-automaton M such that $L(M) = L_{\mathsf{m}}(\mathcal{M})$ holds. M is deterministic, if $\mathcal{M}$ is globally deterministic and the mode of operation is $= j$.*

Theorem 4.4.4. *Let* $\mathsf{X} \in \{\mathsf{RR}, \mathsf{RL}, \mathsf{RRW}, \mathsf{RLW}, \mathsf{RRWW}, \mathsf{RLWW}\}$*, and let $\mathcal{M}$ be a (globally deterministic)* CD-sh-X*-system. Then there exists a (deterministic) shrinking nonforgetting* X*-automaton M such that $L(M) = L_{\mathsf{t}}(\mathcal{M})$ holds.*

Theorem 4.4.5. $\mathcal{L}_{\mathsf{m}}(\mathsf{CD\text{-}sh\text{-}RRWW}) = \mathcal{L}(\mathsf{sh\text{-}RRWW})$ *for all modes of operation* m*.*

Proof. $\mathcal{L}(\mathsf{sh\text{-}RRWW}) \subseteq \mathcal{L}_{\mathsf{m}}(\mathsf{CD\text{-}sh\text{-}RRWW})$ is obvious. The CD-system just consists of two copies of M which alternate according to the mode of operation.

$\mathcal{L}_{\mathsf{m}}(\mathsf{CD\text{-}sh\text{-}RRWW}) \subseteq \mathcal{L}(\mathsf{sh\text{-}RRWW})$:

The theorems in this section show that $\mathcal{L}_{\mathsf{m}}(\mathsf{CD\text{-}sh\text{-}RRWW}) \subseteq \mathcal{L}(\mathsf{nf\text{-}sh\text{-}RRWW})$, from section 2.7 we know that $\mathcal{L}(\mathsf{sh\text{-}RRWW}) = \mathcal{L}(\mathsf{nf\text{-}sh\text{-}RRWW})$, thus the inclusion holds.

This completes the proof. □

There is another possible way to define the notion of shrinking for CD-systems:

Extending CD-systems of restarting automata to shrinking CD-systems of restarting automata can be done similar to state shrinking for nonforgetting restarting automata (see Section 3.7). There each tape symbol and each state has a weight and the shrinking restarting automaton has to reduce its weight in every cycle. Again, as for state shrinking nonforgetting restarting automata we can conjecture that shrinking CD-systems of restarting automata with this notion of shrinking are closed under intersection. It remains for future work to study this notion of shrinking.

4.5 Concluding Remarks

We have seen that CD-systems of restarting automata and nonforgetting restarting automata are closely related. Further, for restarting automata without auxiliary symbols the locally deterministic CD-systems yield language classes that lie strictly in between those classes that are defined by nonforgetting deterministic restarting automata and those classes that are defined by nonforgetting nondeterministic restarting automata. However, the following related questions remain open.

1. To what extend do these results carry over to restarting automata with auxiliary symbols?
2. A nondeterministic CD-system of restarting automata could be called *strict* if there is only a single initial system (that is, $|I_0| = 1$), and if the set of successors is a singleton for each component. It is easily seen that strict CD-systems without auxiliary symbols are more expressive than nondeterministic restarting automata of the same type. However, is there a proper hierarchy of language classes defined by strict CD-systems, based on the number of component systems that lies in between the class of languages defined by nondeterministic restarting automata and the class of languages defined by nonforgetting nondeterministic restarting automata?
3. Also monotone CD-systems of restarting automata are not considered in this work. Again it is possible to define monotonicity in a global or a local manner. A conjecture is that globally monotone CD-RRWW-systems accept only context-free languages, while local monotonicity seems to be related to the notion of j-monotonicity defined in [Plá01].

Bibliography

[AHU69] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. A general theory of translation. *Math. Systems Theory*, 3:193–221, 1969.

[BO93] R.V. Book and F. Otto. *String-Rewriting Systems*. Springer-Verlag, New York, 1993.

[BO98] G. Buntrock and F. Otto. Growing context-sensitive languages and Church-Rosser languages. Information and Computation, 141:1–36, 1998.

[Bun96] G. Buntrock. *Wachsende kontext-sensitive Sprachen*. Habilitationsschrift, Fakultät für Mathematik und Informatik, Universität Würzburg, July 1996.

[CC73] K. Culic, II and R. Cohen. LR-regular grammars - an extension of LR(k) grammars. J. Computer System Sciences, 7:66–96, 1973.

[CVD90] E. Csuhaj-Varju and J. Dassow. On cooperating distributed grammar systems. *Journal of Information Processing and Cybernetics, EIK*, 26:49–63, 1990.

[CVDKP94] E. Csuhaj-Varju, J. Dassow, J. Kelemen, and G. Păun. *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*. Gordon and Breach, London, 1994.

[DP97] J. Dassow and G. Păun. Grammar systems. In Rozenberg G. and Salomaa A., editors, *Handbook of Formal Languages*, volume 2 of *Lecture Notes in Computer Science 3572*, pages 155–213. Springer, Berlin, 1997.

[Har79] M. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, Reading, M.A., 1979.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, M.A., 1979.

[JL02] T. Jurdziński and K. Loryś. Church-Rosser languages vs. UCFL. In P. Widmayer, F. Triguero, R. Morales, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *ICALP 29rd*, *Lecture Notes in Computer Science 2380*, pages 147–158. Springer-Verlag, Berlin, 2002.

[JLNO04] T. Jurdziński, K. Loryś, G. Niemann, and F. Otto. Some results on RWW- and RRWW-automata and their relation to the class of growing context-sensitive languages. J. Automata, Languages and Combinatorics, 9:407–437, 2004.

[JMOP05] T. Jurdziński, F. Mráz, F. Otto, and M. Plátek. Monotone deterministic RL-automata don't need auxiliary symbols. In C. De Felice and A. Restivo, editors, *Developments in Language Theory, DLT 2005, Proc.*, Lecture Notes in Computer Science 3572, pages 284–295. Springer-Verlag, Berlin, 2005.

[JMPV95] P. Jančar, F. Mráz, M. Plátek, and J. Vogel. Restarting automata. In H. Reichel, editor, *Fundamentals of Computation Theory, Proceedings FCT'95*, *Lecture Notes in Computer Science 965*, pages 283–292. Springer-Verlag, Berlin, 1995.

[JMPV97a] P. Jančar, F. Mráz, M. Plátek, and J. Vogel. Monotonic rewriting automata with a restart operation. In F. Plášil and K.G. Jeffery, editors, *SOFSEM'97: Theory and Practise of Informatics*, *Lecture Notes in Computer Science 1338*, pages 505–512. Springer-Verlag, Berlin, 1997.

[JMPV97b] P. Jančar, F. Mráz, M. Plátek, and J. Vogel. On restarting automata with rewriting. In G. Păun and A. Salomaa, editors, *New Trends in Formal Languages*, *Lecture Notes in Computer Science 1218*, pages 119–136. Springer-Verlag, Berlin, 1997.

[JMPV98] P. Jančar, F. Mráz, M. Plátek, and J. Vogel. Different types of monotonicity for restarting automata. In V. Arvind and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science, 18th Conference, Proceedings*, *Lecture Notes in Computer Science 1530*, pages 343–354. Springer-Verlag, Berlin, 1998.

[JMPV99] P. Jančar, F. Mráz, M. Plátek, and J. Vogel. On monotonic automata with a restart operation. J. Automata, Languages and Combinatorics, 4:287–311, 1999.

[JMPV07] P. Jančar, F. Mráz, M. Plátek, and J. Vogel. Monotonicity of restarting automata. J. Automata, Languages and Combinatorics, 12:353–371, 2007.

[JO06] T. Jurdziński and F. Otto. Restricting the use of auxiliary symbols for restarting automata. In J. Farré, I. Litovsky, and S. Schmitz, editors, *Tenth International Conference on Implementation and Application of Automata, CIAA 2005, Proc.*, Lecture Notes in Computer Science 3845, pages 176–187. Springer-Verlag, Berlin, 2006.

[JO07] T. Jurdziński and F. Otto. Shrinking restarting automata. *International Journal of Foundations of Computer Science*, 18:361–385, 2007.

[JOMP04] T. Jurdziński, F. Otto, F. Mráz, and M. Plátek. On the complexity of 2-monotone restarting automata. Mathematische Schriften Kassel 4/04, Universität Kassel, June 2004.

[Lau88] C. Lautemann. One pushdown and a small tape. In Wagner K.W., editor, *Dirk Siefkes zum 50. Geburtstag*, pages 42–47. Technische Universität Berlin and Universität Augsburg, 1988.

[LV97] Ming Li and Paul Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer Press, New York, 1997.

[MMOP06] H. Messerschmidt, F. Mráz, F. Otto, and M. Plátek. Correctness preservation and complexity of simple RL-automata. In O. Ibarra and H.-C. Yen, editors, *CIAA 2006, Proc.*, Lecture Notes in Computer Science 4094, pages 162–172. Springer-Verlag, Berlin, 2006.

[MNO88] R. McNaughton, P. Narendran, and F. Otto. Church-Rosser Thue systems and formal languages. J. Assoc. Comput. Mach. , 35:324–344, 1988.

[MO05] F. Mráz and F. Otto. Hierarchies of weakly monotone restarting automata. RAIRO - Theoretical Informatics and Applications, 39:325–342, 2005.

[MO06] H. Messerschmidt and F. Otto. On nonforgetting restarting automata that are deterministic and/or monotone. In D. Grigoriev, J. Harrison, and E.A. Hirsch, editors, *CSR 2006, Proc.*, Lecture Notes in Computer Science 3967, pages 247–258. Springer, Berlin, 2006.

[MO07a] H. Messerschmidt and F. Otto. Cooperating distributed systems of restarting automata. *International Journal of Foundations of Computer Science*, 18:1333–1342, 2007.

[MO07b] H. Messerschmidt and F. Otto. On determinism versus nondeterminism for restarting automata. In R. Loos, S.Z. Fazekas, and C. Martin-Vide, editors, *LATA 2007, Preproc.*, Report 35/07, pages 413–424. Research Group on Mathematical Linguistics, Universitat Rovira i Virgili, Tarragona, 2007.

[MO07c] H. Messerschmidt and F. Otto. Strictly deterministic CD-systems of restarting automata. In E. Csuhaj-Varjú and Z. Ésik, editors, *Fundamentals of Computation Theory, FCT 2007, Proc.*, *Lecture Notes in Computer Science 4639*, pages 424–434. Springer, Berlin, 2007.

[MO08] H. Messerschmidt and F. Otto. On determinism versus nondeterminism for restarting automata. Information and Computation, 206:1204–1218, 2008. A conference version of this article is published as [MO07b].

[MS04] H. Messerschmidt and H. Stamer. Restart-automaten mit mehreren restartzuständen. In H. Bordihn, editor, *Workshop "Formale Methoden in der Linguistik" und 14. Theorietag "Automaten und Formale Sprachen", Proc.*, pages 111–116. Institut für Informatik, Universität Potsdam, 2004.

[Nar84] P. Narendran. Church-Rosser and related Thue systems. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1984.

[Nie02] G. Niemann. Church-Rosser Languages and Related Classes. Doktorarbeit, Fachbereich Mathematik/Informatik, Universität Kassel, 2002.

[NO98] G. Niemann and F. Otto. The Church-Rosser languages are the deterministic variants of the growing context-sensitive languages. In M. Nivat, editor, *Foundations of Software Science and Computation Structures, FoSSaCS'98, Proc.*, Lecture Notes in Computer Science 1378, pages 243–257. Springer-Verlag, Berlin, 1998.

[NO00] G. Niemann and F. Otto. Restarting automata, Church-Rosser languages, and representations of r.e. languages. In G. Rozenberg and W. Thomas, editors, *Developments in Language Theory - Foundations, Applications, and Perspectives, DLT 1999, Proc.*, pages 103–114. World Scientific, Singapore, 2000.

[NO03] G. Niemann and F. Otto. Further results on restarting automata. In M. Ito and T. Imaoka, editors, *Words, Languages and Combinatorics III, Proc.*, pages 352–369, Singapore, 2003. World Scientific.

[NO05] G. Niemann and F. Otto. The Church-Rosser languages are the deterministic variants of the growing context-sensitive languages. Information and Computation, 197:1–21, 2005.

[OJ03] F. Otto and T. Jurdziński. On left-monotone restarting automata. Mathematische Schriften Kassel 17/03, Universität Kassel, November 2003.

[Ott03] F. Otto. Restarting automata and their relations to the Chomsky hierarchy. In Z. Esik and Z. Fülöp, editors, *Developments in Language Theory, DLT 2003, Proc.*, Lecture Notes in Computer Science 2710, pages 55–74. Springer-Verlag, Berlin, 2003.

[Ott06] F. Otto. Restarting automata. In Z. Ésik, C. Martin-Vide, and V. Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 269–303. Springer, Berlin, 2006.

[Ott07] F. Otto. Left-to-right regular languages and two-way restarting automata. *manuscript*, 2007.

[Ott08] F. Otto. Lower bound techniques. Private communication, Universität Kassel, 2008.

[Plá01] M. Plátek. Two-way restarting automata and j-monotonicity. In L. Pacholski and P. Ružička, editors, *SOFSEM 2001: Theory and Practice of Informatics, Proceedings*, *Lecture Notes in Computer Science 2234*, pages 316–325. Springer-Verlag, Berlin, 2001.

[Plá07] M. Plátek. Private communication, December 2007.

[vBV79] B. von Braunmühl and R. Verbeek. Finite-change automata. In Weihrauch K., editor, *4th GI Conference Proc.*, Lecture Notes in Computer Science 67, pages 91–100. Springer, Berlin, 1979.