Journal of Automata, Languages and Combinatorics **u** (v) w, x–y

On Monotonic Automata with a Restart Operation[1]

Petr Jančar
Technical University of Ostrava, Dept. of Computer Science, 17. listopadu 15, 708 33 OSTRAVA, Czech Republic
e-mail: `Petr.Jancar@vsb.cz`

František Mráz, Martin Plátek
Charles University, Department of Computer Science, Malostranské nám. 25, 118 00 PRAHA 1, Czech Republic
e-mail: `mraz@ksvi.mff.cuni.cz, platek@ksi.mff.cuni.cz`

and

Jörg Vogel
Friedrich Schiller University, Computer Science Institute, 07740 JENA, Germany
e-mail: `vogel@informatik.uni-jena.de`

ABSTRACT
Automata with a restart operation define a special class of rewrite (reduction) systems which has a close relation to the dependency syntax of natural languages. We impose a natural condition of monotonocity and introduce a hierarchical structure of several versions of such automata. The language classes recognized by these automata form a proper hierarchy, with the class of context-free languages on the top, and with the class of deterministic context-free languages on the bottom. In particular, the deterministic monotonic versions of all the introduced automata recognize the same class of languages – namely that of deterministic context-free languages.
Keywords: monotonic automata, monotonic automata with a restart operation, context-free languages.

1. Introduction

This paper summarizes and completes the conference papers [8], [18], [11]. We study restarting automata ($RRWW$-automata) which generalize in two ways the restarting automata with rewriting (RW-automata) introduced in [9].

Our motivation for introducing automata with a restart operation [10] was to model *analysis by reduction* of natural language sentences in a similar way as in [19]. The analysis by reduction consists of stepwise simplification of an extended sentence so that the (in)correctness of the sentence is not affected. Thus, after some number

[1]Supported by the Grant Agency of the Czech Republic, Grant-No. 201/99/0236

of steps, a simple sentence is got or an error is found. It is possible that native English speakers are not familiar with the analysis by reduction. But Czech children learn analysis by reduction of Czech sentences in basic school. This type of training has a very long tradition in the Czech republic (and some other European countries). Analysis by reduction creates a basis for explanations and teaching how to build a dependency grammar for a natural language like Czech, French, German, Russian etc. (see [24] and [19], [22], [15], [17]). Among the types of reductions, we consider the reduction by deleting as methodically significant (and related to contextual grammars [16, 4]). It allows, in a transparent way, to express some techniques which are (often implicitly) used in development of a grammar-checker for the Czech language [14, 5].

We illustrate analysis by reduction on the sentence

'Martin, Peter and Jane work very slowly.'

This sentence can be simplified, for example, as follows:

'Peter and Jane work very slowly.'
'Jane works very slowly.'
'Jane works slowly.'
'Jane works.'

Notice that every simplification is realized by deleting and possible rewriting of words. E.g., the sentence *'Jane works slowly.'* is obtained from the sentence *'Peter and Jane work slowly.'* by deleting *'Peter and'* and rewriting *'work'* by *'works'*.

Let us also demonstrate how to use reductions in a stepwise localization of a syntactic inconsistency on an example which is similar to the previous one:

'Martin, Peter and Jane works slowly.'
'Peter and Jane works slowly.'
'Peter and Jane works.'

The last sentence is not reduced further; it constitutes a core of an error.

The given examples should illustrate that, for the purposes of the natural language processing, it is useful to model reductions of correct as well as incorrect sentences when we are interested in checking (syntactic) correctness.

$RRWW$-automata can serve as a natural framework for a formalization of analysis by reduction. Such an automaton can be roughly described as follows. It has a finite control unit, a head with a lookahead window attached to a list (see Fig. 1), and it works in certain cycles. In a cycle, it moves the head from the left to the right along the word on the list; according to its instructions, it can, at some place, rewrite—once in a cycle—the contents of its lookahead by a shorter string, continue by scanning some further symbols, and finally "restart" – i.e., reset the control unit into the initial state and place the head on the left end of the shortened word. The computation in a cycle halts when the automaton enters an accepting or a rejecting state.

In a straightforward way, analysis by reduction can be modelled by the subclass of RRW-automata, which are the $RRWW$-automata using only the input alphabet symbols in rewriting. These automata enjoy "error preserving property": if a word

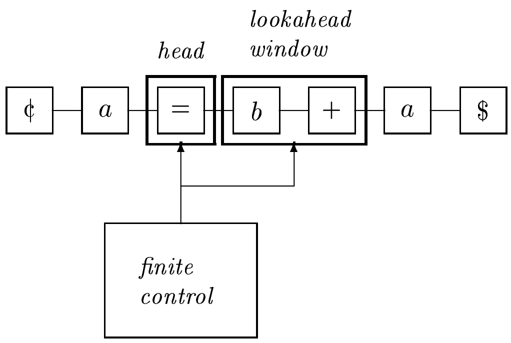

Figure 1: An $RRWW$-automaton with the lookahead window of size 2.

(sentence) is incorrect then the reduced word is also incorrect (after any number of cycles). In addition, deterministic RRW-automata have "correctness preserving property": if a word is correct then the reduced word is correct as well. Similar properties for (general) $RRWW$-automata can not be stated so directly since also non-input symbols can be used for rewriting; nevertheless, this feature provides the power to recognize all context-free languages.

Besides RRW-automata, we introduce further subclasses of $RRWW$-automata. A taxonomy of (nondeterministic versions of) these automata is presented in [11]. Originally we have studied a monotonicity property for computations of a subclass of RW-automata which can delete only – R-automata – [8] and for RW-automata itself [10]. Informally speaking, monotonicity of a computation means that the distance of the place of rewriting to the right end in the $(n+1)$th cycle is not larger than in the nth cycle. In our examples of English sentences, we were reducing in such a monotonic way. One source of motivation for this paper is the observation that the *monotonic* reductions play a significant role in syntax of natural languages. We define an $RRWW$-automaton as monotonic if all its computation are monotonic. We show that all introduced (sub)types of deterministic monotonic $RRWW$-automata recognize exactly deterministic context-free languages ($DCFL$). On the other hand, the language classes recognized by nondeterministic monotonic versions form a proper hierarchy of language classes between $DCFL$ and CFL, yielding, in particular, two characterizations of context-free languages (CFL). General (nonmonotonic) $RRWW$-automata recognize more than context-free languages [11].

The rest of the paper can be outlined as follows. Section 2 has two subsections. In the first subsection we define $RRWW$-automata and show some basic facts about them, namely error preserving and correctness preserving properties and a "pumping lemma". Subsection 2.2 introduces the monotonicity property for $RRWW$-automata, together with a proof of its decidability. Section 3 shows that monotonic $RRWW$-automata recognize exactly CFL and the deterministic monotonic versions of all introduced subclasses of $RRWW$-automata recognize exactly $DCFL$. Section 4 provides a proper hierarchy (between $DCFL$ and CFL) of language classes recognized by the corresponding versions of $RRWW$-automata. In Section 5 we give a sample sentence (in czech) which should illustrate usefulness of the analysis by reduction in linguistic study of some syntactic phenomena. Section 6 contains some additional remarks.

2. Definitions and basic properties

In the first subsection we introduce $RRWW$-automata (nondeterministic and deterministic versions) and we show their basic properties. In the second subsection we define the monotonicity property and monotonic $RRWW$-automata; we show that it is decidable if a given $RRWW$-automaton is monotonic.

By λ we denote the empty word, and by $\emptyset$ the empty set. $\subseteq$ denotes the inclusion relation and $\subset$ denotes the proper inclusion relation. For any two languages L_1, L_2 their concatenation is denoted by $L_1 \cdot L_2$.

2.1. RRWW-automata

A *restarting automaton* is a tuple $M = (Q, \Sigma, \Gamma, k, I, q_0, Q_A, Q_R)$, the components of which are described in what follows. Q is a finite set of *states*, Σ and Γ are disjoint finite sets of symbols, called the *input alphabet* and the *work alphabet* respectively. We denote $V = \Sigma \cup \Gamma$, and suppose that V does not contain the special symbols ¢, and \$ called the *left sentinel* and the *right sentinel* respectively. k is a nonnegative integer called the *size of lookahead*, $q_0 \in Q$ is the *initial state*, $Q_A \subseteq Q$ is the set of *accepting states*, $Q_R \subseteq Q$ is the set of *rejecting states*. I is a finite set of *instructions* of the following three types (where $q, q' \in Q, a \in V \cup \{¢, \$\}$, $u, v \in V^* \cup V^* \cdot \{\$\}$, $k \geq |u| > |v| \geq 0$):

(1) $(q, au) \to (q', MVR)$

(2) $(q, au) \to (q', REWRITE(v))$

(3) $(q, au) \to RESTART$

The *restarting automaton* M is a device with a finite state control unit, and one head moving on a finite linear (doubly linked) list of items (cells). The first item always contains a special symbol ¢, the last one contains another special symbol \$, and each other item contains a symbol from V. The head has a lookahead 'window' of length k ($k \geq 0$) – besides the current item, M also scans the next k right neighbour items (or simply the end of the word when the distance to \$ is less than k).

We suppose that the set of states Q is divided into two classes – the set of *nonhalting states* $Q - (Q_A \cup Q_R)$ (there is at least one instruction which is applicable when the unit is in such a state) and the set of *halting states* $Q_A \cup Q_R$ (any computation finishes by entering such a state); the set of halting states is further divided into the set of accepting states Q_A and the set of rejecting states Q_R.

A *configuration* of the automaton M is (u, q, v), where $u \in \{\lambda\} \cup \{¢\} \cdot V^*$ is the content of the list from the left sentinel to the position of the head, $q \in Q$ is the current state and $v \in \{¢, \lambda\} \cdot V^* \cdot \{\$\} \cup \{\lambda\}$ is the content of the list from the scanned item to the right sentinel.

In the *restarting configuration* on a *word* $w \in (\Sigma \cup \Gamma)^*$, the word ¢$w$\$ is stored in the items of the list, the control unit is in the initial state q_0, and the head is attached to that item which contains the left sentinel (scanning ¢, looking also at the first k symbols of the word w). An *initial computation* of M starts in an *initial configuration* which is a restarting configuration on *an input word* ($w \in \Sigma^*$).

The *computation* of M is controlled by a finite set of instructions I of types (1), (2) and (3) from above. The left-hand side (q, au) of an instruction determines when it is applicable – q means the current state (of the control unit), a the symbol being scanned by the head, and u means the content of the lookahead window (u being a string of length k or less if it ends with \$). The right-hand side describes the activity to be performed.

In case (1), M changes the current state to q' and moves the head to the right neighbour item of the item containing a. In particular, if $a = \$$ then q' must be a halting state.

In case (2), the activity consists of deleting (removing) some items (at least one) of the just scanned part of the list (containing au), and of rewriting some (possibly none) of the nondeleted scanned items (in other words au is replaced with v, where v must be shorter than au). After that, the head of M is moved to the right to the item containing the first symbol after the lookahead and the current state of M is changed to q'. There are two exceptions: if au ends by \$ then v also ends by \$ (the right sentinel cannot be deleted or rewritten) and after the rewriting the head is moved to the item containing \$; similarly, the left sentinel ¢ cannot be deleted or rewritten.

In case (3), *RESTART* means entering the initial state and placing the head on the first item of the list (containing ¢).

Any computation of a restarting automaton M is composed of certain phases. A *phase* called *cycle* starts in a restarting configuration, the head moves to the right along the input list until a restart operation is preformed, and M is resumed in a new restarting configuration. In this paper, we work with $RRWW$-automata which are restarting automata making exactly one *REWRITE*-instruction in each cycle – i.e., new phase starts on a shortened word. A phase of a computation called *tail* starts in a restarting configuration, the head moves to the right along the input list until one of the halting states is reached. M can also once rewrite during the tail.

It immediately implies that any computation of any $RRWW$-automaton is finite (finishing in a halting state).

In general, an $RRWW$-automaton is *nondeterministic*, i.e., there can be two or more instructions with the same left-hand side (q, au). If it is not the case, the automaton is *deterministic*.

An input *word w is accepted by M* if there is an initial computation which starts in the initial configuration with $w \in \Sigma^*$ (bounded by sentinels ¢,\$) on the list and finishes in an *accepting configuration* where the control unit is in one of the accepting states. $L(M)$ denotes the language consisting of all words accepted by M; we say that M *recognizes the language* $L(M)$.

By RRW-automata we mean $RRWW$-automata with empty working alphabet. RRW-automata use only the input symbols in the rewriting, i.e., in all instructions of the form (2) above, the string v is from Σ^*. We can see that all restarting configurations of an RRW-automaton are also its initial configurations.

By RR-automata we mean RRW-automata which use deleting without rewriting (i.e., in all instructions of the form (2) above, the string v can always be obtained by deleting some symbols from au).

By RWW-automata we mean $RRWW$-automata which do restart immediately after any $REWRITE$-instruction.

By RW-automata we mean RRW-automata which do restart immediately after any $REWRITE$-instruction.

By R-automata we mean RR-automata which do restart immediately after any $REWRITE$-instruction.

The notation $u \Rightarrow_M v$ means that there exists a cycle of M starting in the restarting configuration with the word u and finishing in the restarting configuration with the word v; the relation $\Rightarrow_M^*$ is the reflexive and transitive closure of $\Rightarrow_M$. If $u \Rightarrow_M v$ holds, we say that u is (can be) *directly reduced* to v by M. If $u \Rightarrow_M^* v$ holds, we say that u can be *reduced* to v by M.

In the proofs, we often use the next claim:

Claim 2.1 (The error preserving property) *Let $M = (Q, \Sigma, \Gamma, k, I, q_0, Q_A, Q_R)$ be an arbitrary RRWW-automaton, u, v be arbitrary input words from Σ^*. If $u \Rightarrow_M^* v$ and $u \notin L(M)$, then $v \notin L(M)$. [Equivalently, if $u \Rightarrow_M^* v$ and $v \in L(M)$, then $u \in L(M)$.]*

Proof. Let $u, v \in \Sigma^*$ be such that $u \Rightarrow_M^* v$ and $v \in L(M)$. Since $v \in L(M)$ there is some y such that $v \Rightarrow_M^* y$, where y can be accepted by M directly in a tail (without any restart). Because of $u \Rightarrow_M^* v$, the relation $u \Rightarrow_M^* y$ holds. Since $u \in \Sigma^*$ then there is an accepting computation starting in an initial configuration with the word u in the list. $\square$

Corollary. For RRW-automata (with $\Gamma = \emptyset$), $u \Rightarrow_M^* v$ and $v \in L(M)$ always implies $u \in L(M)$, since all its restarting configurations are initial.

The next claim is a useful complement of the previous one for deterministic $RRWW$-automata.

Claim 2.2 (The correctness preserving property) *Let $M = (Q, \Sigma, \Gamma, k, I, q_0, Q_A, Q_R)$ be an arbitrary deterministic RRWW-automaton, u, v be arbitrary input words from Σ^*. If $u \Rightarrow_M^* v$ and $u \in L(M)$, then $v \in L(M)$.*

Proof. Let $u \in L(M)$ and $u \Rightarrow_M^* v$ for some word $v \in \Sigma^*$. Because of determinism of M there is exactly one accepting computation for u by M. The computation represented by sequence of cycles $u \Rightarrow_M^* v$ is a prefix of this computation. Thus the rest of this computation (starting in the initial configuration with v on the list) is an accepting computation for v and therefore $v \in L(M)$. $\square$

Corollary. For deterministic RRW-automata (with $\Gamma = \emptyset$), $u \Rightarrow_M^* v$ and $u \in L(M)$ always implies $v \in L(M)$.

Often we will use also the following 'pigeonhole' lemma.

Lemma 2.3 *Let $M = (Q, \Sigma, \Gamma, k, I, q_0, Q_A, Q_R)$ be an arbitrary RRWW-automaton, and $V = \Sigma \cup \Gamma$. There is a constant p such that for an arbitrary phase (cycle or tail) starting on a word $w_1 v w_2$ (for some words $w_1, v, w_2 \in V^*$), where $|v| \geq p$, the subword v can be written $v_1 a u v_2 a u v_3$ (for some words $v_1, u, v_2, v_3 \in V^*$ and symbol $a \in V$), $|u| = k$ and $|auv_2| \leq p$ where both occurrences of a are entered in the same state by M during the given cycle or tail (including the possibility that both are not entered at all) and in the cycle or tail nothing in auv_2 is deleted or rewritten.*

Proof. We choose p such that $p = 3 \times (k+1) \times |Q| \times |V|^{k+1}$, where $|V|, |Q|$ denote sizes of V and Q, respectively. We can see that if we take a phase starting on the word $w_1 v w_2$, where $|v| \geq p$, then we have two possibilities:

a) M does some rewriting inside v, or partially inside v. Then there is a subword $\overline{v}$ of v which is not changed and its length is surely greater then $p/3$, what means $\overline{v}$ is long enough to be written $\overline{v}_1 a u v_2 a u \overline{v}_3$ (for some words $\overline{v}_1, u, v_2, \overline{v}_3 \in V^*$ and some symbol $a \in V$), $|u| = k$ and $|auv_2| < p$, where both occurrences of a are entered in the same state by M during the considered phase (including the possibility that both occurrences are not entered at all). Since the $\overline{v}$ is not changed at all, then nothing in auv_2 is deleted or rewritten.

b) M does not change v during the phase at all. Then we can easily see that v is long enough to fulfil the assertion.

□

We will use the fact that for any $i \geq 0$ the given phase can be naturally extended for the 'pumped' word $w_1 v_1 (auv_2)^i auv_3 w_2$, where $i \geq 0$ (including also the case of removing – $i = 0$).

2.2. Monotonic RRWW-automata

Next we introduce the monotonicity property of computations of *RRWW*-automata.

Any cycle *Cyc* performed by an *RRWW*-automaton M has a significant configuration $c_w = (¢u, q, v\$)$ in which begins the rewriting step of the cycle *Cyc*. We call the number $|v|$ ($|v|$ means the length of the string v) the distance of the place of the rewriting in *Cyc* to the right sentinel and we denote it by $D_r(Cyc)$.

We say that a computation $\mathcal{C}$ of an *RRWW*-automaton M is *monotonic* if for the sequence of its cycles $Cyc_1, Cyc_2, \ldots, Cyc_n$ the sequence $D_r(Cyc_1)$, $D_r(Cyc_2)$, …, $D_r(Cyc_n)$ is not increasing, i.e., monotonic. Notice, that the tails are not considered for the monotonicity.

By a *monotonic RRWW-automaton* we mean an *RRWW*-automaton for which all its computations (also the non-initial computations which can start in any restarting configuration) are monotonic.

Theorem 2.4 *There is an algorithm which for any RRWW-automaton M decides whether M is monotonic or not.*

Proof. The proof can be made in a quite similar way as for the RW-automata in [10]. It suffices to show that for any (nondeterministic) $RRWW$-automaton $M = (Q, \Sigma, \Gamma, k, I, q_0, Q_A, Q_R)$ we can (effectively) construct a nondeterministic finite automaton A_M s.t. a nonmonotonic computation of M exists iff A_M accepts a nonempty language.

In principle, A_M simulates two possible consecutive cycles. Let us denote them by Cyc_1, Cyc_2, of M on a given word (from V^*) simultaneously, and accepts iff it reveals that $D_r(Cyc_1) > D_r(Cyc_2)$.

A_M uses a sufficiently long buffer to simulate Cyc_1 and to keep $(k+1)$ symbols "behind" the scanning head in Cyc_1. This enables to simulate steps of Cyc_2 on these symbols already passed by M in Cyc_1. At the time of rewriting in Cyc_2, the automaton A_M checks whether there are symbols scanned in Cyc_1 before or during the rewriting step (of Cyc_1) such that they have been not scanned in Cyc_2. If it happens, then A_M checks whether M can restart in both cycles – if yes, then A_M accepts, otherwise rejects.

A_M is constructed as follows. It has a sufficiently long buffer, e.g. with positions $0, 1, 2, \ldots, 2k+2$, in the finite control unit (k is the size of lookahead of M). At first, it fills the buffer with the initial segment of the input word. Then it fills the 'variables' $CSt_1 := q_0$, $CSt_2 := q_0$, $Pos := 0$. Here the variable CSt_1 always contains the current state in Cyc_1, CSt_2 the current state in Cyc_2; Pos determines the position of the head in the buffer in Cyc_1. During the simulation, the symbol on the position 0 corresponds to the item scanned by the head in the second simulated cycle – Cyc_2.

The main computation of A_M is controlled by the following scheme:

- if $Pos \le k$ then one step of Cyc_1 is simulated in the buffer (always when the buffer content is shortened, the buffer is filled at the right end by reading further symbols from the current list unless \$ is already in the buffer);
- if $Pos > k$ then one step of Cyc_2 is simulated and the buffer is shifted to the left (and completed from the right) – so that symbol at position 0 is the symbol scanned in the next step of Cyc_2.

A_M simulates both cycles until rewriting or halting occurs in one of them:

- When halting in some of the cycles is simulated, then A_M rejects.
- When rewriting in Cyc_2 is simulated (i.e., $Pos > 0$ and no rewriting occurred in Cyc_1), then A_M continues to simulate both cycles. If afterwards both cycles end by restarting, then A_M accepts, otherwise it rejects.
- When rewriting in Cyc_1 is simulated, then A_M rewrites the corresponding part of the buffer and marks the last symbol of the replacing string (or the symbol at position $Pos - 1$ if the replacing string was empty). Then the simulation continues until the marked symbol appears at position less or equal to k – i.e., it is scanned in Cyc_2. If rewriting in Cyc_2 does not occur before the marked symbol appears in lookahead in Cyc_2, then A_M rejects, otherwise it continues to simulate both cycles. If afterwards both cycles end by restarting, then A_M accepts, otherwise it rejects.

□

For brevity, prefix *det-* denotes the deterministic versions of $RRWW$-automata, similarly *mon-* the monotonic versions. $\mathcal{L}(\mathcal{A})$, where $\mathcal{A}$ is some class of automata, denotes the class of languages recognizable by automata from $\mathcal{A}$. E.g. the class of languages recognizable by deterministic monotonic R-automata is denoted by $\mathcal{L}(\mathit{det\text{-}mon\text{-}R})$.

3. Monotonic $RRWW$-automata and $(D)CFL$

In this section there are presented characterizations of *CFL* and *DCFL* by monotonic $RRWW$-automata.

For technical reasons we introduce a special form of $RRWW$-automata. We say that an $RRWW$-automaton $M = (Q, \Sigma, \Gamma, k, I, q_0, Q_A, Q_R)$ is in the *RR-semi-deterministic form* (*RR*-semidet-form) if it fulfils the following properties:

a) M moves in each phase through the whole input list until the head is scanning the right sentinel. Then M either restarts or moves into a halting state.

b) For any couple (q, v), where $q \in Q$ and $v \in (\Sigma \cup \Gamma)^* \cup (\Sigma \cup \Gamma)^* \cdot \{\$\}$, there is at most one instruction of the form $(q, v) \to (q', MVR)$ in I.

Obviously, in a configuration where the head of an $RRWW$-automaton in the *RR*-semidet-form is scanning the right sentinel there is at least one instruction applicable.

Lemma 3.1 *For any mon-RRWW-automaton M there is an mon-RRWW-automaton M' in the RR-semidet-form such that $L(M) = L(M')$. Moreover, if M is deterministic, then M' is also deterministic.*

Proof. It is not hard to see that for any *mon-RRWW*-automaton M there is an equivalent *mon-RRWW*-automaton M_1 which does not halt or restart inside of a list, i.e., each of its phase ends at the right sentinel by restarting or halting. Such a transformation can be made also for any deterministic $RRWW$-automaton in such a way that it preserves the determinism.

If M_1 is deterministic, we can take it as M', otherwise we transform M_1 into M' in a way which is similar to the transformation of a nondeterministic finite automaton into a deterministic one (cf. [7]) – i.e., the states of M' are the subsets of the set of states of M_1. All states entered by M' while its head is not scanning the right sentinel are nonhalting. At the right sentinel M' either restarts or enters a halting state (using *MVR*-instruction). A halting state of M' is accepting if it contains at least one accepting state, otherwise it is a rejecting state.

M' can rewrite at the same places as M, thus M' is also monotonic. □

The following theorem states one of the main results of this paper. Any (deterministic) monotonic $RRWW$-automaton recognizes a (deterministic) context-free language.

Theorem 3.2 *$\mathcal{L}(\mathit{mon\text{-}RRWW}) \subseteq \mathit{CFL}$ and $\mathcal{L}(\mathit{det\text{-}mon\text{-}RRWW}) \subseteq \mathit{DCFL}$.*

Proof. Let L be a language recognized by a *mon-RRWW*-automaton $M = (Q, \Sigma, \Gamma, k, I, q_0, Q_A, Q_R)$. W.l.o.g. we suppose that M is in the *RR*-semidet-form, i.e., in each configuration at most one *MVR*-instruction is applicable (and any number of *REWRITE*-instructions) and M restarts or halts on the right sentinel only.

Each phase (cycle or tail) by M containing *REWRITE*-instruction can be naturally divided into three parts: the left-hand part – steps until the *REWRITE*-instruction, the middle part – the *REWRITE*-instruction itself and the right-hand part – steps after the *REWRITE*-instruction. Tails without any *REWRITE*-instruction are considered as having the left-hand part only. The particular parts of a phase determine the corresponding left-hand, middle, and the right-hand part of the corresponding list. The middle part begins with the item scanned by the *REWRITE*-instruction and ends by the item scanned after the performance of the *REWRITE*-instruction.

In the following we show how to construct a (nondeterministic) pushdown automaton P which simulates nondeterministic M and accepts $L(M) \cdot \{\$\}$. Thus $L(M) = L(P)/\{\$\}$, where / denotes the right quotient operation. Because the class *CFL* is closed on the right quotient with any regular language, thus $L(M)$ is a context-free language. As M is nondeterministic (the modifications needed in the case of deterministic *RRWW*-automaton will be handled separately), we can suppose that M does not rewrite in any tail (at the place where it can make *REWRITE*-instruction, it can guess, that this is a tail computation, skip the rewriting and check the guess; if at the right sentinel M cannot accept, then it rejects, even if M could restart). The construction of P is based on the construction of a pushdown automaton equivalent to a monotonic *RW*- automaton in [10]. Modification must be made in order to simulate also the right-hand parts of phases.

At any step, P simulates left-hand or middle part of a phase denoted by *Phase* and right-hand parts of all phases preceding *Phase*. This is enabled by the monotonicity property of M – the places of *REWRITE* do not increase their distances from the right sentinel. Simultaneously, in the pushdown store of P an auxiliary information will be kept for simulation of (left-hand parts of) phases following after *Phase*.

The control unit of P has several components for storing finite information. The component CSt contains the current state of M in the corresponding step of *Phase*. The component B contains a string of input symbols of the length at most $(2k+1)$ – it will contain the scanned item and the lookahead of M in *Phase* and it will be used in the *REWRITE*-instruction simulation. The next component R contains a string of at most $k+1$ input symbols – it will contain the scanned symbol and the string in the lookahead for the purpose of simulation of the right-hand parts of phases preceding *Phase*. Let n denote the number of states of M. The last component is a vector RS of the length n. The elements RS_i, for $1 \leq i \leq n$, are sets containing at most one state of M – this vector will be used in the simulation of the right-hand parts of phases preceding *Phase*.

The pushdown store of M will contain symbols already scanned by M in *Phase* with some auxiliary information. Actually each symbol stored in the pushdown is composed of an input symbol from the list and a state in which the symbol was entered (from the left) in *Phase*. Because of the *RR*-semidet-form of M, this information can be used in the simulation of the left-hand parts of the phases coming after *Phase*.

Any reading of an input symbol by P is done using the following procedure:

Get_input_symbol: If $R = \lambda$ then do nothing (the input head of P has reached the end of input). Otherwise, for each $i = 1, 2, \ldots, n$, if RS_i is nonempty, i.e., $RS_i = \{q_i\}$, for some $q_i \in Q$, simulate one step in the right-hand part of some phase(s) preceding *Phase* – a step where the head of M scans the first symbol of R and its lookahead window contains the rest of R and the control unit is in the state q_i.

- If $R \neq \$$, then because of the RR-semidet-form it has exactly one instruction of the form $(q_i, R) \to (q_i', MVR)$. P replaces the content of RS_i by q_i'.
- If $R = \$$, then M has at most one instruction of the form $(q_i, \$) \to (q_i', MVR)$ (in such a case, q_i' is a halting state) and/or the instruction $(q_i, \$) \to RESTART$. M nondeterministically selects one of its applicable instructions:
  - If it is an instruction of the form $(q_i, \$) \to (q_i', MVR)$ then P replaces the content of RS_i by q_i'.
  - If it is the *RESTART*-instruction, then P discards RS_j by entering $\emptyset$ into RS_j.

After updating whole RS, if $RS_i = RS_j \neq \emptyset$, for some $1 \leq i < j \leq n$, it means that in right-hand parts of some two phases (preceding *Phase*) M enters the same state at the same symbol (corresponding to the second symbol in R). But then due to RR-semidet-form of M, these two phases would continue by the same steps and both end in the same state. The computation of M in the remaining parts of both phases is the same, thus we need not further simulate both phases. It suffices to continue in simulation of the former phase – we discard all such RS_j by entering $\emptyset$ into RS_j. Then we 'shake-down' the nonempty entries in RS so that all nonempty items from RS are in a contiguous segment of RS with indices $1, 2, \ldots$.

Next P removes the first symbol from R and appends it to B, shifts the content of R to the left (the content of R is shortened) if R is nonempty and the last symbol of R is not \$ then P reads the next input symbol and appends it to R.

This procedure simulates stepwise the right-hand parts of phases preceding *Phase*. The following *RS-invariant* is kept:

> If RS_i contains a state q of M then all $RS_1, \ldots, RS_{i-1}$ contain states of M. q is the state in a configuration from the right-hand part of some phase C of M preceding the current phase *Phase*, such that in this configuration the head of M is scanning the part of the list stored in R. There is no phase preceding C in which this part was scanned in the same state q. For any j, $1 \leq j < i$, there is a phase preceding C in which this part is scanned in the state stored in RS_j.

The simulation-procedure of P:

Initialization: P starts by storing the initial state of M into CSt, pushing ¢ (the left endmarker of M) into the first cell of the buffer R and the first k symbols of the input word of M into the next k cells of the buffer R (cells $2, 3, \ldots, k+1$). All RS_i, for $i = 1, 2, \ldots, n$, are initialized to $\emptyset$. P initializes B by performing the procedure *Get_input_symbol* $(k+1)$-times.

The main cycle: During the simulation, the following conditions will hold invariantly:
– CSt contains the state of M visiting the simulated (currently scanned) item in the simulated phase *Phase*,
– the first cell of B contains the current symbol of M (scanned by the head in *Phase*) and the rest of B concatenated with R contains m right neighbour symbols of the current one (lookahead of length $m \leq 3k+1$),
– the pushdown contains the left-hand side (w.r.t. the head) of the list in *Phase*, the leftmost symbol (¢) being at the bottom. In fact, any pushdown symbol will be composed of the relevant symbol in the working list and the state of M in which this symbol is entered in *Phase*,
- for RS the RS-*invariant* holds.

The mentioned invariants will be preserved by the following simulation of applications of instructions of M. The left-hand side (q, au) of the instruction to be simulated is determined by the information stored in the control unit – $q = CSt$, au is the prefix of B. The activity to be performed depends on the right-hand sides of applicable instructions of M. P nondeterministically selects:

1. Instruction of the form $(q, au) \to (q', MVR)$ (at most one possible):
   P puts the content of the first cell of B and CSt as a composed symbol on the top of the pushdown, stores q' into CSt, and shifts the content of B to the left. If the length of B is less than $k+1$ then P executes *Get_input_symbol*.
2. One of several possible instructions of the form $(q, au) \to (q', REWRITE(v))$:
   The first $|au|$ symbols from B are replaced by the shorter sequence v. The topmost $k+1$ symbols are successively popped from the pushdown and the relevant symbols are added from the left to B (shifting the rest to the right). The state parts of (composed) symbols are forgotten, the state part of the $(k+1)$-th symbol (the leftmost symbol in B) is stored in CSt. Thus not only the $REWRITE(v)$-instruction is simulated but also the beginning of the left-hand part of the next phase, which is the new *Phase*.

   It should be clear that because of monotonicity of M, at the time of simulating rewriting the first symbol after the lookahead u is the first symbol of R and the simulation of the corresponding right-hand part will start on it in the state q'. Let $l = \max(\{0\} \cup \{i \mid 1 \leq i \leq s : RS_i \neq \emptyset\}$. If there is no RS_i $(1 \leq i \leq l)$ equal to q', then P stores q' in RS_{l+1}.

   The simulation of the $REWRITE$-instruction has one exception – in the situation when v ends by the right sentinel (\$), the head of M is placed on the right sentinel (see definition of $RRWW$-automaton in Sect. 2.1). In that case M has at most one instruction of the form $(q', \$) \to (q'', MVR)$ (in such a case, q'' is a halting state) and/or the instruction $(q', \$) \to RESTART$. M nondeterministically selects one of its applicable instructions:

- If it is an instruction of the form $(q', \$) \to (q'', MVR)$ – i.e., q'' is a halting state – then let $l = \max(\{0\} \cup \{i \mid 1 \leq i \leq s : RS_i \neq \emptyset\}$. If there is no RS_i $(1 \leq i \leq l)$ equal to q'', then P stores q'' in RS_{l+1}.
- If it is the *RESTART*-instruction, then P does nothing – cycle ended by restart and after the right sentinel we store in R halting states only.

Note, please, that instructions of the form $(q, \$) \to RESTART$ in the right-hand parts are simulated in the procedure *Get_input_symbol*, because *RRWW*-automaton can restart only after rewriting.

End of the simulation: At this point B is empty, i.e., M has processed the right sentinel and the last simulated *Phase* was a tail (M cannot rewrite in a tail, thus the right-hand side was empty), which must end in a halting state. Thus CSt contains a halting state and the vector RS contains halting states in which ended right-hand parts in all the phases preceding *Phase*. At this point P using RS and CSt decides whether the simulated computation was an accepting computation by M or a rejecting one. The computation was accepting if

- RS_1 is nonempty and it contains an accepting state, or
- each RS_i $(i = 1, 2, \ldots, n)$ is empty and CSt is an accepting state.

Otherwise the input word is rejected.

It should be clear that due to monotonicity of M the second half of B (cells $k + 2$, $k + 3, \ldots, 2k + 1$) is empty at the time of simulating a $REWRITE(v)$-operation. Hence the described construction is correct which proves the inclusion

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

Next we will show how to modify the above construction in the case of deterministic *RRWW*-automaton M. According Lemma 3.1, we can consider any deterministic *mon-RRWW*-automaton M to be in the *RR*-semidet-form. On the other hand we do not know at this moment how to transform M in such a way that it does not rewrite in tails, and at the same time keeps the property of determinism. From this reason the construction which should keep the determinism will be a bit more complex. If M is monotonic and simultaneously deterministic, a rewriting in its tail can be "nonmonotonic", but not nonmonotonic in an arbitrary way. Such nonmonotonic rewriting in a tail must be close to the rewriting done in the last phase – at least one symbol in the lookahead in the nonmonotonic rewriting instruction must be from the part rewritten in the last phase (otherwise the rewriting should be done in some previous phase). If the rewriting was monotonic, then the simulation can continue without modifications. Otherwise, such nonmonotonic rewriting can be detected by the pushdown automaton P above because by simulating such rewriting, the buffer B contains more than $k + 1$ symbols. After simulating such an instruction, P knows that this case can occur only in a tail. Thus it continues in simulating the right-hand parts of *Phase* and the preceding phases. This can be done simply by performing procedure *Get_input_symbol* until the buffer R is empty. The end of the simulation is a bit modified comparing to the nondeterministic case – RS_1 cannot be empty (this would mean that M can restart after the nonmonotonic rewriting and thus M is not monotonic). If RS_1 contains an accepting state, then the input word is accepted otherwise it is rejected.

The above construction applied to a *det-mon-RRWW*-automaton yields a deterministic pushdown automaton recognizing $L(P) = L(M).\{\$\}$. Because *DCFL* is closed on the right quotient with any regular language, the language $L(M) = L(P)/\{\$\}$ is a deterministic context-free language – this proves the second part of the statement. □

Lemma 3.3 *DCFL* $\subseteq \mathcal{L}(\textit{det-mon-R})$.

Proof. We use the proof from [8]. It uses the characterization of deterministic context-free languages by means of $LR(0)$–grammars and $LR(0)$–analyzers (cf. e.g. [7]); generally $LR(1)$–grammars (lookahead 1) are needed, but it is not necessary when each word is finished by the special sentinel \$.

Let L' be a deterministic context-free language. Then there is a $LR(0)$–grammar G generating $L = L'\$$ (the concatenation $L' \cdot \{\$\}$ supposing \$ being not in the alphabet of L'), and there is a corresponding $LR(0)$–analyser P.

For any word $w \in L$ there is only one derivation tree T_w; it corresponds to the analysis of w by the analyser P. In fact, P simulates construction of T_w in the left-to-right and bottom-up fashion. Due to the standard pumping lemma for context-free languages, there are constants p, q s.t. for any w with length greater than p there are a (complete) subtree T_1 of T_w and a (complete) subtree T_2 of T_1 with the same root labelling; in addition, T_2 has fewer leaves than T_1 and T_1 has q leaves at most. (Cf. Fig. 2; A is a nonterminal of G). Replacing T_1 with T_2, we get the derivation tree for a shorter word w' (w could be written $w = u_1v_1u_2v_2u_3$ in such a way that $w' = u_1u_2u_3$).

Now we outline a *det-mon-R*-automaton M with lookahead of length $k > q$ which recognizes L'.

M stores the content of the lookahead in a buffer in the control unit. Simulating the $LR(0)$–analyser P, it constructs (in a bounded space in the control unit) all maximal subtrees of the derivation tree which have all their leaves in the buffer. If one of the subtrees is like the T_1 above, M performs the relevant deleting (of at most two continuous segments) in the input list and restarts. If it is not the case then M forgets the leftmost of these subtrees with all its $n \geq 1$ leaves, and reads n new symbols to the right end of the buffer (shifting the content to the left). Then M continues constructing the maximal subtrees with all leaves in the (updated) buffer (simulating P).

In general, the input word w is either shorter than p, such words can be checked using finite memory, or it is longer. If it is longer and belongs to L then M must meet the leftmost above described T_1 (with the subtree T_2); it performs the relevant deleting and restarts on a new, shorter, word. If the (long) input word w does not belong to L, M either meets \$ without restarting and stops in a rejecting state or

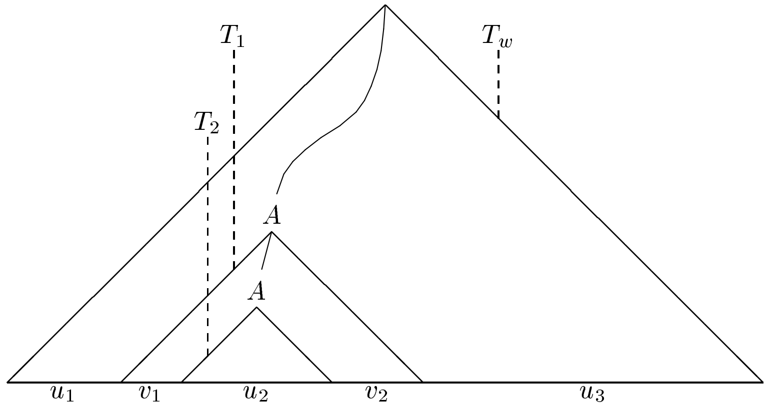

Figure 2:

performs some deleting and restarts. It suffices to show that the resulting shorter word (in both cases) is in L if and only if w is in L.

It can be verified using the following properties of the $LR(0)$–analyser.

a) For each word $w \in L$ there is exactly one derivation of w in G which corresponds to the analysis of w by P.

b) Let u be the prefix of the input word $w = uv$ which has been already read by the $LR(0)$–analyser P. If P did not reject the word until now, then there exists a suffix word v' s.t. uv' is in L, and the computation of P on the prefix u is independent w.r.t. the suffix.

Let u be the prefix of the input word, the last symbol of which corresponds to the last symbol in the lookahead window just before M performs a *RESTART*-operation; let $\overline{u}$ be the rest of u after performing the *RESTART*-operation. There exists a suffix v' such that uv' is in L. uv' has a derivation tree in which there is a complete subtree T_1 (with a subtree T_2; as above) corresponding to the place of cutting. Then in the computation of M on $\overline{u}$ the tree T_2 will appear in the buffer above the same terminal leaves as in the computation on u (it follows from the presence of T_2 in the derivation tree of $\overline{u}v'$ and from the independence of computation on the suffix).

Let $w = uv$ is in L, then obviously $\overline{u}v$ is in L. Conversely if uv is not in L then $\overline{u}v$ is not in L (otherwise, in the corresponding derivation tree of $\overline{u}v$, the subtree T_2 appears over the corresponding terminal leaves of $\overline{u}$ and replacing the tree T_2 by T_1 yields a derivation tree for uv – a contradiction).

The monotonicity of M should be clear from the above description. $\square$

From Lemma 3.3 and Theorem 3.2 follows that all deterministic monotonic subclasses of $\mathcal{L}(RRWW)$ collapse to *DCFL*.

Theorem 3.4 *The following equality holds for* $X \in \{R, RR, RRW, RWW, RRWW\}$*:*

$$\mathcal{L}(\mathit{det\text{-}mon\text{-}X}) = \mathit{DCFL}.$$

The next theorem shows two characterizations of *CFL* by restarting automata.

Theorem 3.5 $\mathit{CFL} = \mathcal{L}(\mathit{mon\text{-}RRWW}) = \mathcal{L}(\mathit{mon\text{-}RWW})$

Proof. Obviously $\mathcal{L}(\mathit{mon\text{-}RWW}) \subseteq \mathcal{L}(\mathit{mon\text{-}RRWW})$. The inclusion $\mathcal{L}(\mathit{mon\text{-}RRWW}) \subseteq \mathit{CFL}$ is proved by Theorem 3.2. It remains to show that for any $L \in \mathit{CFL}$ there is a *mon-RWW*-automaton M such that $L(M) = L$.

The proof is based on the possibility to recognize any context-free language not containing the empty word λ by a pushdown automaton in a special form:

- the pushdown automaton is nondeterministic and accepts by an empty storage;
- before it increases height of its pushdown by 1 symbol, it must read at least two input symbols;
- it reads at least one symbol in each step.

For any context-free language L (not containing the empty word) a pushdown automaton of such a special form can be obtained in two steps:

1. Any context-free language not containing λ can be generated by a context-free grammar G in the quadratic Greibach normal — i.e., all rules of G are of the form $A \to a\alpha$, where a is a terminal symbol and α is a string of nonterminals of length at most two (cf. [1]). For G there can be constructed an equivalent pushdown automaton P_1 accepting the language $L(G)$ by empty storage s.t. it reads an input symbol in every step and the height of its pushdown store at step i is at most i.
2. The pushdown automaton P_1 from the previous step can be transformed into pushdown automaton P which uses "shorter pushdown" — i.e., in step i the height of its pushdown store is at most $i/2 + 1$. This can be accomplished by a standard compressing techniques, where one pushdown symbol of P encodes several (e.g. 2) pushdown symbols of P_1.

Any pushdown automaton P in such a special form can be simulated by a *mon-RWW*-automaton M:

- the current state of P is stored in the list of items, in the item which contains the symbol scanned by the head of P;
- the content of the pushdown store is stored in the current list left to the item carrying the current state of P; we can suppose that the pushdown alphabet of P is disjunctive with the input alphabet of P, thus M in an restarting configuration knows whether it is the initial configuration.
- one phase of M corresponds to two steps of P — two symbols are removed from the list and replaced by (at most) one pushdown symbol.

In the case that a context-free language L contains the empty word λ, we can use the above constructions for $L - \{\lambda\}$ (which is context-free) and modify the final *mon-RWW*-automaton to accept also λ.

$\square$

4. Taxonomy of monotonic *RRWW*-automata

In this section, we show the relations between different versions of monotonic *RRWW*-automata. All the results are depicted in Fig. 3 and proved by the following lemmas.

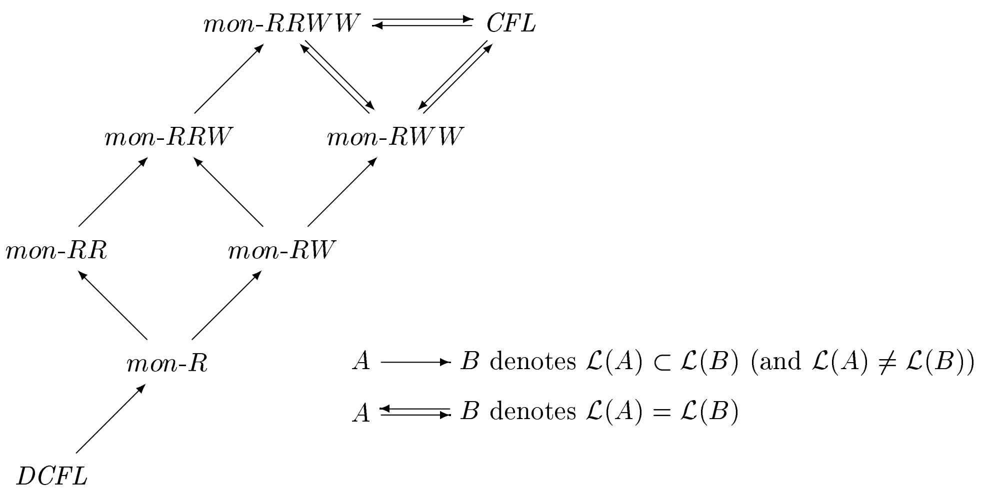

Figure 3: Relations between classes of languages recognized by monotonic versions of *RRWW*-automata. If there is no directed path between two classes, then these classes are incomparable.

Lemma 4.1 $\mathcal{L}(\textit{mon-RR}) - \mathcal{L}(RW) \neq \emptyset$.

Proof. We will prove that the language $L = L_1 \cup L_2$, where $L_1 = \{a^n b^n c \mid n \geq 0\}$, $L_2 = \{a^n b^{2n} d \mid n \geq 0\}$ is a *mon-RR*-language and is not an *RW*-language.

L is recognized by a (nondeterministic) *mon-RR*-automaton M with lookahead 2 which works as follows:

- M accepts if it scans $\cent c\$$, $\cent d\$$ or $\cent abc\$$, otherwise
- M moves to the "middle" (a followed by a different symbol) and it should scan abb. M guesses whether the current word is in L_1 or it is in L_2. In the first case it rewrites abb to b, in the second to λ. After that M moves to the right end to check the last input symbol. In the first case it should be c, in the second one d. If the check is not successful, M rejects, otherwise it restarts.

On the other hand the language L cannot be accepted by any RW-automaton. For a contradiction let M_1 be an RW-automaton recognizing L. Any accepting computation of M_1 on a sufficiently long word $a^n b^{2n} d$ has at least one cycle – otherwise using Lemma 2.3 we can construct a word outside L which will be accepted by M_1 in a tail. W.l.o.g. we can suppose n being greater than k. M_1 can only shorten the "middle" (and immediately restart – it is an RW-automaton) to get a word of the form $a^m b^{2m} d$, for some $m < n$. But then the M_1 can reduce the word $a^n b^{2n-m} c \notin L$ to the word $a^m b^m c$ from L_2. But this fact contradicts the error preserving property (Claim 2.1). □

Lemma 4.2 $\mathcal{L}(mon\text{-}RW) - \mathcal{L}(RR) \neq \emptyset$.

Proof. Let $L_3 = \{c^n d^n \mid n \geq 0\}$, $L_4 = \{c^n d^m \mid m > 2n \geq 0\}$ and $L = \{f, ee\} \cdot L_3 \cup \{g, ee\} \cdot L_4$. The language L can be recognized by a nondeterministic monotonic RW-automaton M in the following way:

- M immediately accepts the word f, otherwise
- if the word starts by fc then M simply deletes cd "in the middle" of the word and restarts,
- if the word starts by gc then M deletes cdd "in the middle" of the word and restarts,
- if the word starts by gd then M scans the rest of the word. If it contains only d's then accepts otherwise rejects,
- if the word starts by ee then M nondeterministically rewrites ee by f or g and restarts.

It is easy to show that M is monotonic and $L(M) = L$.

L cannot be recognized by any RR-automaton. For a contradiction let us suppose $L = L(M)$ for some RR-automaton M with the lookahead of the length k. Let us choose (and fix) a sufficiently large n, $n > 2k$ (i.e., even by deleting $k+1$ symbols c from $c^n c^n$ we cannot get any word from $\{ee\} \cdot L_4$) s.t. n is divisible by $p!$ (and hence by all $p_1 \leq p$) where p is taken from Lemma 2.3. Now consider the first cycle C of an accepting computation of M on $eec^n d^n$. M can only shorten both segments of c's and d's in the same way, i.e., $eec^n d^n \Rightarrow_M eec^l d^l$, for some $l < n$. (Any accepting computation of M on $eec^n d^n$ has at least one cycle – otherwise using Lemma 2.3 we can construct a word outside L which will be accepted by M in a tail). Due to Lemma 2.3, d^n can be written $d^n = v_1 auv_2 auv_3$, $a = d$, $u = d^k$, $|auv_2| \leq p$, where M in the cycle C enters both occurrences of a in the same state (including the possibility that both are not entered at all).

Recall that nothing is deleted out of auv_2 in C and $|auv_2|$ divides n due to our choice of n. Then there is some i s.t. $d^{2n} = v_1 (auv_2)^i auv_3$; hence $eec^n d^{2n} \notin L(M)$ but (by the natural extending of the cycle C) we surely get $eec^n d^{2n} \Rightarrow_M eec^l d^{n+l}$, where $2l < n + l$ and therefore $eec^l d^{n+l} \in L(M)$ – a contradiction with the error preserving property (Claim 2.1). □

Lemma 4.3 *$CFL - \mathcal{L}(RRW) \neq \emptyset$.*

Proof. Let us take languages $L_3 = \{c^n d^n \mid n \geq 0\}$ and $L_4 = \{c^n d^m \mid m > 2n \geq 0\}$ from the previous proof and $L = L_3 \cup L_4$. This language is obviously context-free. We show by a contradiction that it is not an RRW-language.

Suppose $L = L(M)$ for some RRW-automaton M with lookahead of length k. Let us choose (and fix) a sufficiently large n, $n > 3k$ (i.e., even by rewriting k symbols d into c's and deleting one c out of $c^n d^n$ we cannot get any word from L_4) s.t. n is divisible by $p!$ (and hence by all $p_1 \leq p$) where p is taken from Lemma 2.3. The rest of this proof is omitted and can be completed considering the first cycle of an accepting computation on $c^n d^n$ and using the same arguments as in the proof of of Lemma 4.2. □

The next theorems show all the relations depicted in Fig. 3.

Theorem 4.4 *$\mathcal{L}(mon\text{-}R) \subset \mathcal{L}(mon\text{-}RR)$, $\mathcal{L}(mon\text{-}RW) \subset \mathcal{L}(mon\text{-}RRW)$.*

Proof. Clearly holds $\mathcal{L}(mon\text{-}R) \subseteq \mathcal{L}(mon\text{-}RR)$, $\mathcal{L}(mon\text{-}RW) \subseteq \mathcal{L}(mon\text{-}RRW)$. The inclusions are proper because of Lemma 4.1. □

Theorem 4.5 *$\mathcal{L}(mon\text{-}RR) \subset \mathcal{L}(mon\text{-}RRW)$, $\mathcal{L}(mon\text{-}R) \subset \mathcal{L}(mon\text{-}RW)$.*

Proof. Trivially $\mathcal{L}(mon\text{-}RR) \subseteq \mathcal{L}(mon\text{-}RRW)$, $\mathcal{L}(mon\text{-}R) \subseteq \mathcal{L}(mon\text{-}RW)$. The inequalities between the classes follow from Lemma 4.2. □

Theorem 4.6 *$\mathcal{L}(mon\text{-}RW)$ is incomparable with $\mathcal{L}(mon\text{-}RR)$.*

Proof. $\mathcal{L}(mon\text{-}RW) \not\subseteq \mathcal{L}(mon\text{-}RR)$ follows from Lemma 4.1, the opposite relation $\mathcal{L}(mon\text{-}RW) \not\supseteq \mathcal{L}(mon\text{-}RR)$ follows from Lemma 4.2. □

Theorem 4.7 *$\mathcal{L}(mon\text{-}RRW) \subset CFL$.*

Proof. It is clear that $\mathcal{L}(mon\text{-}RRW) \subseteq (mon\text{-}RRWW)$. From Theorem 3.5 follows $CFL = mon\text{-}RRWW$. The inequality in the theorem follows from Lemma 4.3. □

The next lemma (already shown in the [10]) can be used for separating *mon-R*-languages from *DCFL*.

Lemma 4.8 *$\mathcal{L}(mon\text{-}R) - \mathcal{L}(det\text{-}RW) \neq \emptyset$.*

Proof. We will show that the language

$$L = \{a^i b^j \mid 0 \leq i \leq j \leq 2i\}$$

is a *mon-R*-language but not a *det-RW*-language.

L is recognized by a (nondeterministic) *mon-R*-automaton M with lookahead 4 which:

- immediately accepts the empty word,

- immediately rejects any nonempty word starting by b,
- on a nonempty word starting by a, moves to the right to this a. If the lookahead contains $b\$$ ($bb\$$, $abb\$$, resp.) then deletes ab (abb, $aabb$ resp.) and restarts. Otherwise, M moves through a's to the right until its head scans a followed immediately by a different symbol. If its lookahead is not $bbbb$ or $bbb\$$, the word is rejected, else M nondeterministically deletes ab or abb and restarts.

Obviously M recognizes L.

On the other hand the language L cannot be accepted by any deterministic *RW*-automaton. For a contradiction let M be a *det-RW*-automaton recognizing L. Any accepting computation of M on a sufficiently long word $a^n b^n$ has at least one cycle – otherwise using Lemma 2.3 we can construct a word outside L which will be accepted by M in a tail. M is deterministic, thus it keeps the correctness preserving property (Claim 2.2) and $a^n b^n \Rightarrow_M a^r b^s$ for some $r < n$ and $s \geq r$, i.e., the automaton must decrease the number of a's in the word, thus rewriting can occur only when the head is visiting some a in the word. Because of determinism and fixed size of lookahead the automaton must work in the same way on the word $a^n b^{2n}$. But the resulting word at the end of the first cycle is $a^r b^{s+n}$, where $s + n > 2r$, which is not in L. This is a contradiction to the correctness preserving property of *det-RW*-automata (Claim 2.2). □

Theorem 4.9 *DCFL* $\subset \mathcal{L}(mon\text{-}R)$.

Proof. It is clear that $\mathcal{L}(det\text{-}mon\text{-}R) \subseteq (mon\text{-}R)$. From Theorem 3.4 follows *DCFL* $= \mathcal{L}(det\text{-}mon\text{-}R) = \mathcal{L}(det\text{-}mon\text{-}RW)$. The inequality in the theorem follows from Lemma 4.8. □

5. Some illustrations by Czech syntax

We have add this section according to some hints of (anonymous) referees.

The notion of restarting automaton is highly influenced by linguistic work of J. Panevová, her colleages, and (former) students. Let us look on an example from [21]. The following text is in fact an almost unchanged part of the first section of [21].

> A syntactic parsing procedure gives syntactic structures based on language-dependent syntactic rules, where not only morphological information, but also certain types of syntactic is used. One type of such information is illustrated by Example (a):
>
> **Example (a):** *'Trosky 49letého zaměstnance, který tam plnil své úkoly, zcela zavalily.'* (MF Dnes, 24.7.94)
>
> [Ruins - 49 years old$_{Gen/AccSg}$ - employee$_{Gen/AccSg}$ - , - who - there - fulfilled - his - tasks - , - completely - buried.]
>
> (The ruins completely buried a 49 years old employee who fulfilled his tasks there.)

The main verb *zavalit* [buried] needs an Object in the Accusative case. This is the reason why the structure is excluded where the noun *zaměstance* with morphological tags Gen/Acc is analysed as an attribute (NG-type) modifier. (This is the end of the quotation from [21]).

Let us illustrate what is the meaning of the proposition *'The main verb zavalit [buried] needs an Object in the Accusative case.'* in the terms of analysis by reduction. At first we show an example of a correct analysis by reduction of the sentence from Example (a).

'Trosky 49letého zaměstnance, který tam plnil své úkoly, zcela zavalily.'

[Ruins - 49 years old$_{Gen/AccSg}$ *- employee*$_{Gen/AccSg}$ *- , - who - there - fulfilled - his - tasks - , - completely - buried.]*

'Trosky zaměstnance, který tam plnil své úkoly, zcela zavalily.'

[Ruins - employee$_{Gen/AccSg}$ *- , - who - there - fulfilled - his - tasks - , - completely - buried.]*

'Trosky zaměstnance, který plnil své úkoly, zcela zavalily.'

[Ruins - employee$_{Gen/AccSg}$ *- , - who - fulfilled - his - tasks - , - completely - buried.]*

'Trosky zaměstnance, který plnil úkoly, zcela zavalily.'

[Ruins - employee$_{Gen/AccSg}$ *- , - who - fulfilled - tasks - , - completely - buried.]*

'Trosky zaměstnance zcela zavalily.'

[Ruins - employee$_{Gen/Acc}$ *completely - buried.]*

'Trosky zaměstnance zavalily.'

[Ruins - employee$_{Gen/AccSg}$ *- buried.]*

We can observe that *'Trosky zaměstnance zavalily.'* is a correct, complete, simple Czech sentence. On the other hand, we can see, that *'Trosky zavalily.'* is an incorrect (or rather incomplete) Czech sentence.

From it follows that the following reduction is an incorrect one:

'Trosky zaměstnance zcela zavalily.'

[Ruins - employee$_{Gen/AccSg}$ *- completely - buried.]*

* *'Trosky zcela zavalily.'*

[Ruins - completely - buried.]

To make our consideration more complete, let us take also the following correct reductions:

'Trosky zaměstnance zcela zavalily židli.'

[Ruins - employee$_{Gen/AccSg}$ *completely - buried - chair.]*

(The ruins of an employee completely buried a chair.)

'Trosky zcela zavalily židli.'
[Ruins - completely - buried - chair.]

'Trosky zavalily židli.'
[Ruins - buried - chair.]

We believe that we have with the previous example illustrate some ways of reasoning in the Czech syntax, which are based on analysis by reduction. We also believe that it is not hard to see, that the examined type of analysis by reduction, can be captured by a procedure which can be considered for a RR-automaton (even for a monotonic one). Last remark to this section: We have used the standart type of presentation of Czech examples in the english-written linguistic papers.

6. Conclusions

In this paper, we have studied one type of monotonicity of computations of $RRWW$-automata based on the distances of places of rewriting from the right sentinel. In the future we will consider also left-monotonicity based on the distances of places of rewriting from the left sentinel, and some combined types of (non)monotonicity. We will investigate also computations in which the right (or left) rewriting distance is strictly decreasing.

Recently, we have studied some more general types of monotonicity on $RRWW$-automata. The questions of decidability are in this cases more complicated. In this way, we have got new classes of languages. The achieved results are summarized in [11], and [12].

Formerly appeared some other models with some similar features as $RRWW$-automata. Contraction automata introduced by Von Solms in [25] can be considered as a special class of automata with the restart operation, different from the class of $RRWW$-automata. A contraction automaton works as a restricted linear bounded automaton (without a lookahead). It simulates the operation deleting using a special symbol, and works in cycles. Any cycle starts and ends on the right sentinel, and uses one reversal on the left sentinel. The contraction automata work with some technical restrictions, which includes limited number of non-input symbols and limited number of changes, which can be performed on the tape during a cycle. These restrictions allow characterizations of matrix languages, certain class of random context languages and a type of ETOL languages (see [3]).

The RW-automata are closely related to finite controlled (string) rewriting systems (c-system) introduced by Chottin [2]. A finite c-system over an alphabet Σ is a subset S of $\Sigma^* \times \Sigma^* \times \Sigma^*$ which can be decomposed as a finite union: $S = \bigcup_{i=1}^{n} R_i \times \{u_i\} \times \{v_i\}$ where the R_i are rational (regular) sets and u_i, v_i are words. Such a c-system defines a reduction relation $\vdash_S$: $f \vdash_S g$ iff $f = r_i v_i s$, $g = r_i u_i s$ for some $i \in \{1, \ldots, n\}$, $r_i \in R_i$ and $s \in \Sigma^*$. The equivalence relation $(\vdash_S \cup \vdash_S^{-1})^*$, noted $\overset{*}{\leftrightarrow}_S$, is the right-congruence generated by S. In [23] there is established that $DCFL$ are exactly those languages which can be defined as $[R]_{\overset{*}{\leftrightarrow}_S}$ where R is a regular language and $\overset{*}{\leftrightarrow}_S$ is the right-congruence relation generated by a basic, confluent, strictly length-reducing, finite c-system S. The restriction "basic" is a combinatorial property which in some sense resembles the monotonicity property for $RRWW$-automata. In the future, this "algebraical approach" should be compared to our "automata approach".

From the point of view of our current linguistic research, we are convinced in the significance of the classes of (monotonic) RRW-automata. The presented theoretical results should help us to show that the syntactic (but also semantic) valencies can be for Czech (stepwise) described (and verified) by tools which are (roughly speaking) not stronger than monotonic RRW-automata. But also it can be shown that this tools cannot cover at the same time the phenomenon of the Czech free word order in its whole scope. One possible use of restarting automata for a verification of a dependency grammar (described by the method of algebraic specifications) is mentioned in [20].

In [6] there are studied some measures of syntactic discontinuity (non-projectivity) which seem to be unbounded (by a fixed constant) considering at the same time (all possible) syntactic trees (expressing the second part of the relation form - function (meaning)), and the whole scope of the word order in Czech. It is shown there that this observation does not support the general belief, that the (complete syntactic relations of) natural languages should be mildly context-sensitive (i.e., semi-linear and polynomially parsable, see [13]). On the other hand we believe that if we focus our attention only on the phenomenon of the syntactic correctness of Czech, that it can be fully checked by some *"almost monotonic"* RRW-automaton. We also believe that we will be able to formulate the *"almost monotonicity"* in such a way that it helps us also to show that the correctness of Czech sentences can be described in some mildly context-sensitive way. The new types of monotonicity introduced in [12] are certain steps in this direction.

To summarize the previous paragraphs we believe that the paradigms of different types of restarting automata should help by a stepwise representation and verification of different syntactic phenomena into a frame of a gradually developed formal description of a syntax of a natural language.

References

[1] J. Autebert, J. Berstel, L.Boasson: Context-Free Languages and Pushdown Automata, in: G. Rozenberg, A. Salomaa (Eds.): Handbook of Formal Languages, Volume 1 – Word, Language, Grammar, Springer, 1997, pp. 111–174

[2] L. Chottin: Strict Deterministic Languages and controlled rewriting systems, in: Proc. 6th ICALP, Lecture Notes in Computer Science, Vol. 71,Springer-Verlag, Berlin, 1979, pp. 104–117

[3] J. Dassow, G. Păun: Regulated Rewriting in Formal Language Theory; EATCS Monographs on Theoretical Computer Science, Springer Verlag, 1989

[4] A. Ehrenfeucht, G. Păun, G. Rozenberg: Contextual Grammars and Formal Languages, In: Handbook of Formal Languages, Volume 2, G. Rozenberg, A. Salomaa (Eds.), Springer Verlag, 1997, pp. 237–293

[5] T. Holan, V. Kuboň, M. Plátek: A Prototype of a Grammar Checker for Czech, Proceedings of the Fifth Conference on Applied Natural Language Processing,

ed. Association for Computational Linguistics, Washington, March 1997, pp. 147–154

[6] T. Holan, V. Kuboň, K. Oliva, M. Plátek: Two Useful Measures of Word Order Complexity, In: Proceedings of the Workshop 'Processing of Dependency-Based Grammars, ed. A. Polguére, S. Kahane, Université de Montréal, Montreal, Quebeck, Canada, August 1998, pp. 21–28

[7] J. Hopcroft, J. Ullman: Introduction to Automata Theory, Languages, and Computation; Addison-Wesley 1979

[8] P. Jančar, F. Mráz, M. Plátek, J. Vogel: Restarting Automata, in Proc. FCT'95, Dresden, Germany, August 1995, LNCS 965, Springer Verlag, 1995, pp. 283–292

[9] P. Jančar, F. Mráz, M. Plátek, J. Vogel: Restarting Automata with Rewriting, in Proc. SOFSEM'96: Theory and practice of informatics, Milovy, Czech Republic, November 1996, LNCS 1175, Springer Verlag, 1996, pp. 401–408

[10] P. Jančar, F. Mráz, M. Plátek, J. Vogel: On Restarting Automata with Rewriting, in New Trends in Formal Languages (Control, Cooperation and Combinatorics), Eds. G. Paun, A. Salomaa, LNCS 1218, Springer Verlag, 1997, pp. 119–136

[11] P. Jančar, F. Mráz, M. Plátek, J. Vogel: Different Types of Monotony for Restarting Automata. In: V. Arvind, R. Ramanujam (Eds.): FST&TCS'98, LNCS 1530, Springer Verlag, Berlin, 1998, pp. 343–354

[12] P.Jančar, F.Mráz, M.Plátek, J.Vogel: On Monotony for Restarting Automata, in MFCS'98 'Mathematical Linguistics' Workshop Proceedings, Ed. M. Kudlek, Bericht 213, Universität Hamburg, Fachbereich Informatik, 1998, pp. 71–84

[13] A. K. Joshi, K. Vijay-Shanker, D. Weir: The convergence of mildly context-sensitive grammatical formalisms, In: P.Sells, S. Shieber, T. Wasow (eds.), Foundational Issues in Natural Language Processing, MIT Press, Cambridge MA, 1991

[14] V. Kuboň, M. Plátek: A Grammar Based Approach to a Grammar Checking of Free Word Order Languages, In: Proceedings of the 15th International Conference on Computational Linguistics COLING 94, VOl.II, Kyoto, Japan, August 1994, pp. 906–910

[15] J.Kunze: Abhängigkeitsgrammatik, Studia Grammatica XII, Berlin 1975

[16] S. Marcus: Contextual Grammars and Natural Languages, In: Handbook of Formal Languages, Volume 2, G.Rozenberg A.Salomaa (Eds.), Springer Verlag, 1997, pp. 215–235

[17] I. A. Meľčuk: Dependency Syntax: Theory and Practice, State University of New York Press, 1988

[18] F. Mráz, M. Plátek, P. Jančar, J. Vogel: Monotonic Rewriting Automata with a Restart Operation, in SOFSEM'97: Theory and Practice of Informatics, Eds. F. Plášil, K. J. Jeffery, LNCS 1338, Springer Verlag, 1997, pp. 505–512

[19] M. Novotný: *With Algebra from Language to Grammar and back (S algebrou od jazyka ke gramatice a zpět)*, Academia, Praha, 1988, (*in Czech*)

[20] K. D. Oliva, V. Petkevič: Phenomena-Based Description of Dependency Syntax: A Survey of Ideas and Formalization, in: Issues of Valency and Meaning, Studies in Honour of Jarmila Panevová, Karolinum, Charles University Press, 1998

[21] J. Panevová, M. Straňáková: Some Types of Syntactic Ambiguity; How to Treat Them in an Automatic Procedure, accepted to 'Workshop on Text, Speech and Dialog (TSD'99)', Plzeň, Czech Republic, September 13–17, 1999

[22] L. Tesnière: Éléments de syntaxe structurale, Editions Klincsieck, Paris, 1998

[23] G. Sénizergues: A Characterisation of Deterministic Context-Free Languages by Means of Right-Congruences, Theoretical Computer Science, 70(2), 1990, pp. 213–232

[24] P. Sgall, J. Panevová: Dependency Syntax - a Challenge, Theoretical Linguistics, Vol.15, No.1/2, Walter de Gruyter, Berlin - New York 1988/89, pp. 73–86

[25] S. H. von Solms: The Characterization by Automata of Certain Classes of Languages in the Context Sensitive Area; Inform. Control 27, 1975, pp. 262–271