Monotonic Rewriting Automata with a Restart Operation*

František Mráz† Martin Plátek† Petr Jančar‡ Jörg Vogel§

Abstract

We introduce a hierarchy of monotonic rewriting automata with a restart operation and show that its deterministic version collapses in a sequence of characterizations of deterministic context-free languages. The nondeterministic version of it gives a proper hierarchy of classes of languages with the class of context-free languages on the top.

1 Introduction

We introduce rewriting automata with a restart operation ($RRWW$-automata) which generalize in two ways the restarting automata with rewriting (RW-automata) introduced in [2].

A RW-automaton can be roughly described as follows. It has a finite control unit, a head with a lookahead window attached to a list, and it works in certain cycles. In a cycle, it moves the head from left to right along the word on the list; according to its instructions, it can at some point rewrite the contents of its lookahead by a shorter string of input symbols and "restart" – i.e. reset the control unit to the initial state and place the head on the left end of the shortened word. The computation halts in an accepting or a rejecting state.

A $RRWW$-automaton can after the rewriting check the remaining part of the list, and in the rewriting instruction it can use also some noninput symbols.

As usual, we define some subclasses of the automata. Similarly as in [1] we study a natural property of monotonicity (during a monotonic computation, "the places of rewriting do not increase their distances from the right end"). We show that all introduced types of deterministic monotonic $RRWW$-automata recognize exactly deterministic context-free languages ($DCFL$). On the other side the nondeterministic version of the introduced hierarchy of $RRWW$-automata gives a proper hierarchy of classes of languages with the class of context-free languages (CFL) on the top.

Our motivation for introducing the restarting automata with rewriting in [3] was to model (elementary) syntactic analysis of natural languages in a similar way as in [4]. Such syntactic analysis consists in stepwise simplification of an extended sentence so that the (in)correctness of the sentence is not affected. Thus after some number of steps a simple sentence is got or an error is found. Such computations can be done by so called RRW-automata, which are the $RRWW$-automata using only the input symbols in rewriting. Formally is this property expressed by the so called "error preserving property". The $RRWW$-automata using the non-input symbols does not ensure the error preserving property automatically, but on the other side they are able to recognize all CFL.

*Supported by the Grant Agency of the Czech Republic, Grant-No. 201/96/0195

†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

‡University of Ostrava, Department of Computer Science, Bráfova 7, 701 03 OSTRAVA, Czech Republic, e-mail: jancar@osu.cz

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

2 Definitions and basic properties

We present the definitions informally; the formal technical details could be added in a standard way of the automata theory.

A *RRWW-automaton* M (with bounded lookahead) 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 another special symbol \$, and each other item contains a symbol from a finite alphabet (not containing ¢, \$). The head has a lookahead "window" of length k (for some $k > 0$) – 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). In the *initial configuration*, the control unit is in a fixed, initial, state and the head is attached to the item with the left sentinel ¢ (scanning also the first k symbols of the input word).

The *computation* of M is controlled by a finite set of *instructions* of the following three types:

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

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

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

The left-hand side 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 contents 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 case (2), the activity consists of deleting (removing) some items of the just scanned part of the list (containing au), and of rewriting some 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 right to the item containing the first symbol after the lookahead and the current state of M is changed to q'. There is one exception: 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 \$.

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

We will suppose that the control unit states are divided into two groups – the *nonhalting states* (an instruction is always applicable when the unit is in such a state) and the *halting states* (any computation finishes by entering such a state); the halting states are further divided into the *accepting states* and the *rejecting states*.

Any computation of an *RRWW*-automaton is naturally divided into certain phases or *cycles* by performed *RESTART*-instructions: in one cycle, the head moves right along the input list (with a bounded lookahead) until a halting state is entered or the computation is resumed in a initial configuration (thus a new cycle starts). We demand that the automaton makes exactly one *REWRITE*-instruction in each cycle ending by *RESTART* – i.e. new cycle starts on a shortened word.

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

In general, a *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 a computation which starts in the initial configuration with w (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 which use only the input symbols in the rewriting, i.e. in instructions of the form (2) above, the string v contains symbols from the input alphabet only).

By RR-automata we mean RRW-automata for which rewriting can be replaced by deleting (i.e. in instructions of the form (2) above, the string v can be obtained by deleting some symbols from au).

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

The next obvious claim express the mentioned lucidness of computations of RRW-automata. The notation $u \Rightarrow_M v$ means that there exists a cycle of M starting in the initial configuration with the word u and finishing in the initial configuration with the word v; the relation $\Rightarrow_M^*$ is the reflexive and transitive closure of $\Rightarrow_M$.

Claim 2.1 (The error preserving property) *Let M be an RRW-automaton, u, v some words in the alphabet of M. If $u \Rightarrow_M^* v$ and $v \in L(M)$, then $u \in L(M)$. [Equivalently, if $u \Rightarrow_M^* v$ and $u \notin L(M)$, then $v \notin L(M)$.]*

We can easily see that the general $RRWW$-automata does not ensure the error preserving property, because of using symbols not belonging to the input alphabet during their computations.

It will be useful to note the following 'pigeonhole' fact.

Lemma 2.2 *Let M be a $RRWW$-automaton. There is a constant p such that for any cycle $w_1vw_2 \Rightarrow_M w'$ (for some words w_1, v, w_2, w'), where $|v| \geq p$, the subword v can be written $v_1auv_2auv_3$, (for some words v_1, u, v_2, v_3 and symbol a), $|u| = k$ and $|auv_2| \leq p$ where both occurrences of a are entered in the same state by M during the given cycle (including the possibility that both are not entered at all) and in the cycle nothing in auv_2 is deleted or rewritten.*

We will use the fact that for any $i \geq 0$ the given cycle can be naturally extended for the 'pumped' word $w_1v_1(auv_2)^iauv_3w_2$, where $i \geq 0$ (including also the case of removing – $i = 0$).

Next let us introduce the monotonicity property of computations of $RRWW$-automata. Let for any cycle Cyc, in which a *REWRITE*-instruction is performed, $Dist(Cyc)$ denote the distance of the last item in the lookahead window at the place of rewriting from the right sentinel (\$) in the current list. We say that a computation C of an $RRWW$-automaton M is monotonic if for the sequence of its cycles $Cyc_1, Cyc_2, ..., Cyc_n$ the sequence $Dist(Cyc_1), Dist(Cyc_2), ..., Dist(Cyc_n)$ is monotonic (not increasing).

By a *monotonic $RRWW$-automaton* we mean a $RRWW$-automaton for which all its computations are monotonic.

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

Theorem 2.3 *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 [3].

□

3 Monotonic $RRWW$-automata

For technical reasons we introduce a special form of $RRWW$-automata. We say that a $RRWW$-automaton M is in a *det-MVR*-form if for any couple (q, v), where q is a state of M and v is a string of its input symbols, there is at most one instruction of the form $(q, v) \to (q', MVR)$.

Lemma 3.1 *For any mon-RRWW-automaton M there is a equivalent mon-RRWW-automaton M' in the det-MVR-form.*

Proof: We omit the proof here. It can be done in a similar way as the construction of equivalent deterministic finite automaton for a given nondeterministic one. □

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

Proof: Let L be a language recognized by a *mon-RRWW*-automaton M, with a lookahead of length k. W.l.o.g. we suppose that M in each cycle scans the whole input list until the right sentinel. Scanning the right sentinel \$, M either restarts or accepts or rejects. Moreover we will suppose M being in the *det-MVR*-form, i.e. in each configuration at most one *MVR*-instruction is applicable (and any number of *REWRITE*-instructions).

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

We show how to construct a (nondeterministic) pushdown automaton P which simulates M and accepts $L(M).\{\$\}$. Thus $L(M) = L(P)/\{\$\}$, where / denotes the right quotient. Because the class *CFL* is closed on the right quotient with any regular language, thus $L(M)$ is a context-free language. The construction of P is based on the construction of a pushdown automaton equivalent to a monotonic restarting automaton with rewriting in [3]. Modification must be made for simulating also the right parts of cycles. Actually, if M rewrites in a cycle C_1 in which it enters a halting state (at the right sentinel) and w_1 is the contents of its working list at the end of the cycle C_1 then P will simulate also computation of M which starts with w_1 on its list. P will simulate further cycles of M – let C_2 denote the next cycle in which M enters a halting state. If M rewrites in a cycle C_2 and w_2 is the contents of the list at the end of C_2 then P will simulate also computation of M which starts with w_2 on the list and so on until M enters a halting state in a cycle with the left part only (a cycle without rewriting). At the end of simulation P must decide which was the first halting state entered by M.

At any time, P simulates left or middle part of a cycle denoted by *Cycle* and right parts of all cycles preceding *Cycle*. 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 parts of) cycles following after *Cycle*.

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 *Cycle*. The component B contains a string of input symbols of length at most $(2k + 1)$ – it will contain the scanned item and the lookahead of M in *Cycle* and will be used in *REWRITE*-instruction simulation. The next component R contains a string of at most $k + 1$ input symbols – it will contain the scanned symbol and lookahead for simulation of the right parts of cycles preceding *Cycle*. Let n denote the number of states of M. The last component is a vector RS of length n. The element RS_i,

for $1 \leq i \leq n$, can be empty or can contain a state of M – this vector will be used in simulation of the right parts of cycles preceding *Cycle*.

The pushdown store of M will contain symbols already scanned by M in *Cycle* with some auxiliary information. Actually the symbols stored in the pushdown are composed of the input symbol from the list and a state in which the symbol was entered (from the left) in *Cycle*. Because of the *det-MVR*-form of M, this information can be used in simulation of left parts of the cycles following after *Cycle*.

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

Get_input_symbol: If R=\$ then do nothing (we are at the end of the list). Otherwise for each RS_i different from λ ($i = 1, 2, \ldots, n$) simulate one step in the right part of some cycle preceding *Cycle* – when 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 RS_i. Because of the *det-MVR*-form there is exactly one instruction of the form $(RS_i, R) \rightarrow (q_i, MVR)$ (for $i = 1, 2, \ldots, n$). P replaces the contents of each RS_i by q_i. If $RS_i = RS_j \neq \lambda$, for some $1 \leq i < j \leq n$, it means that in right parts of some two cycles (preceding *Cycle*) M enters the same state $q_i = q_j$ at the same symbol (corresponding to the second symbol in R). But then due to *det-MVR*-form of M, these two cycles would continue by the same steps and both end in the same state. Accepting/rejecting depends on the first halting state entered by M at the right sentinel, thus we need not simulate both cycles, it is enough to continue in simulation of the former cycle – we discard RS_j by shifting the contents of $RS_{j+1}, \ldots, RS_n$ to $RS_j, \ldots, RS_{n-1}$ and entering λ into RS_n.

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

This procedure simulates stepwise the right parts of cycles preceding *Cycle*. 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. Then q is the state in which M scans the first symbol x stored in R in the right part of some cycle C preceding the current cycle *Cycle* such, that there is no cycle preceding C in which this item was scanned in the same state q and for any j, $1 \leq j < i$, there is a cycle preceding C in which x is scanned in the state RS_j.

The simulation algorithm 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 B 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 λ. 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 in which M can be visiting the simulated (currently scanned) item in the simulated cycle *Cycle*,
- the first cell of B contains the current symbol of M (scanned by the head in *Cycle*) 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 *Cycle*, the leftmost symbol (¢) being at the bottom. In fact, any pushdown symbol will be composed – it will contain the relevant symbol of the working list and the state of M in which this

symbol could be entered in *Cycle*,
- for RS the RS-invariant holds.

The mentioned invariants will be preserved by the following simulation 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. The activity to be performed depends on the right-hand sides of applicable instructions of M:

1. At most one possible instruction of the form $(q, au) \to (q', MVR)$:
   P puts the contents 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 contents 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 instruction 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 part of the next cycle, which is the new *Cycle*.

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

End of the simulation: At this point B is empty, i.e. the head of M is scanning the right sentinel \$ in the current cycle *Cycle* and the vector RS contains states in which ended right parts in all the cycles preceding *Cycle*. At this point P using RS decides which was the first halting state state entered by M. Let i is minimal such that RS_i contains a halting state. If RS_i is accepting, then P accepts, otherwise rejects.

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 $RESTART(v)$-operation. Hence the described construction is correct which proves $\mathcal{L}(mon\text{-}RRWW) \subseteq CFL$.

Obviously, deterministic *mon-RRWW*-automaton is in *det-MVR*-form and the above construction applied to a *det-mon-RRWW*-automaton yields a deterministic pushdown automaton recognizing $L(P) = L(M).\{\$\}$. Because $DCFL$ is closed under 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 $DCFL \subseteq \mathcal{L}(det\text{-}mon\text{-}R)$

Proof: This lemma was proved in [1]. We omit the proof here. □

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 holds:*
$\mathcal{L}(det\text{-}mon\text{-}R) = \mathcal{L}(det\text{-}mon\text{-}RR) = \mathcal{L}(det\text{-}mon\text{-}RRW) = \mathcal{L}(det\text{-}mon\text{-}RRWW) = DCFL$.

Despite the previous theorem we will show that the corresponding nondeterministic classes of languages create a hierarchy of CFL.

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

Proof: To prove this theorem it remains to show that for any $L \in CFL$ there exists a $mon\text{-}RRWW$-automaton M such that $L(M) = L$.

W.l.o.g. we can suppose that for L there is a pushdown-automaton P such that: $L(P) = L$, P puts at most two symbols on the pushdown store (the length of the pushdown increases by 1) in one step and any prolongation of the pushdown is preceded by reading two input symbols at least, P does not use any input symbol as a pushdown symbol, P accepts with the input head on the right end and the control unit in an accepting state. We outline M simulating P. M will keep the following invariant: After any cycle the list of M contains a word of the form $\#uzav\$$, where u is the content of the simulated pushdown without the top of the pushdown (with its bottom on the left side), z is a couple – the symbol on the top of the pushdown and the current state of P, a is the scanned symbol by the input head, and v is the nonread part of the input word. The supposed form of P ensures that M can work as a monotonic $RRWW$-automaton. □

Actually, in the previous construction M could be a RW-automaton enhanced with the possibility to use noninput symbols in rewriting.

Theorem 3.6 $\mathcal{L}(mon\text{-}R) \not\subseteq \mathcal{L}(mon\text{-}RR) \not\subseteq \mathcal{L}(mon\text{-}RRW) \not\subseteq \mathcal{L}(mon\text{-}RRWW)$.

Proof: Trivially, $\mathcal{L}(mon\text{-}R) \subseteq \mathcal{L}(mon\text{-}RR) \subseteq \mathcal{L}(mon\text{-}RRW) \subseteq \mathcal{L}(mon\text{-}RRWW)$.

a) Next 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 $mon\text{-}RR$-language and is not a R-language.

i) L is recognized by a (nondeterministic) $mon\text{-}RR$-automaton M which works as follows:

- M accepts if it scans $\¢c\$$, $\¢d\$$ or $\¢abc\$$, otherwise
- M moves to the "middle" (a followed by a different symbol) and 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.

ii) On the other hand the language L cannot be accepted by any R-automaton. For a contradiction let M_1 be a R-automaton recognizing L. Any accepting computation of M_1 on a sufficiently long word $a^n b^{2n} d$ has at least two cycles – otherwise using Lemma 2.2 we can construct a word outside L which will be accepted by M_1 in one cycle. M_1 can only shorten the "middle" (and immediately restart – it is an R-automaton) to get a word of the form $a^m b^{2m} d$. But then the M_1 can reduce the word $a^n b^{n+(n-m)} c \notin L$ to the word $a^m b^m c$ from L_2. But this fact contradicts the error preserving property (Claim 2.1).

b) Secondly we will show that $\mathcal{L}(mon\text{-}RR)$ is a proper subclass of $\mathcal{L}(mon\text{-}RRW)$. Let $L_3 = \{c^n d^n \mid n \geq 0\}$, $L_4 = \{c^n d^m \mid m > 2n \geq 0\}$ and $L = \{f, ee\}.L_3 \cup \{g, ee\}.L_4$. The language L can be recognized by a nondeterministic monotonic RRW-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 lookahead of length k. Let us choose (and fix) a sufficiently large n ($n > k$) s.t. n is divisible by $p!$ (and hence by all $p_1 \leq p$) where p is taken from Lemma 2.2. 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 two cycles – otherwise using Lemma 2.2 we can construct a word outside L which will be accepted by M in one cycle). Due to Lemma 2.2, 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.

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 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).

c) It remains to show that $\mathcal{L}(mon\text{-}RRW)$ is a proper subclass of $\mathcal{L}(mon\text{-}RRWW)$. 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 point b) and $L = L_3 \cup L_4$. This language is obviously context-free. Because of Theorem 3.5 the language L can be recognized by a $mon\text{-}RRWW$-automaton. We show by a contradiction in a very similar way as in the point b) that it is not a 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 ($n > k$) s.t. n is divisible by $p!$ (and hence by all $p_1 \leq p$) where p is taken from Lemma 2.2. Now consider the first cycle C of an accepting computation of M on $c^n d^n$. M can only shorten both segments of c's and d's in the same way, i.e. $c^n d^n \Rightarrow_M c^l d^l$, for some $l < n$. (Any accepting computation of M on $c^n d^n$ has at least two cycles – otherwise using Lemma 2.2 we can construct a word outside L which will be accepted by M in one cycle). Due to Lemma 2.2, 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.

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 $c^n d^{2n} \notin L(M)$ but (by the natural extending of C) we surely get $c^n d^{2n} \Rightarrow_M c^l d^{n+l}$, where $2l < n + l$ and therefore $c^l d^{n+l} \in L(M)$ – a contradiction with the error preserving property (Claim 2.1).

□

Conclusion remark: In future work we will consider also nonmonotonic classes of $RRWW$-automata. We will also consider the $RRWW$-automata in an explicit way as reduction systems (see [3]).

References

[1] 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

[2] 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

[3] 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

[4] M.Novotný: *S algebrou od jazyka ke gramatice a zpět*, Academia, Praha, 1988, (*in Czech*)