Theoretical Computer Science 115 (1993) 261–275
Elsevier

Synchronized finite automata and 2DFA reductions*

Oscar H. Ibarra and Nicholas Q. Trân

Department of Computer Science, University of California, Santa Barbara, CA 93106, USA

Communicated by A. Salomaa
Received May 1991
Revised December 1991

Abstract

Ibarra, O.H. and N.Q. Trân, Synchronized finite automata and 2DFA reductions, Theoretical Computer Science 115 (1993) 261–275.

We establish a tight hierarchy of two-way synchronized finite automata with only universal states on the number of allowed processes ($\mathscr{L}(2\text{SUFA}(k\text{-proc})) \subset \mathscr{L}(2\text{SUFA}((k+1)\text{-proc}))$) by studying the reduction functions made by two-way deterministic finite automata with a one-way write-only output tape. As corollaries, we show that, for every $k>1$, $\mathscr{L}(2\text{SUFA}(k\text{-proc}))$ has a complete set under 2DFA reductions and is not closed under Boolean operations.

In contrast, we show that the corresponding hierarchy collapses for unary alphabets; this follows from our characterization of $\bigcup_{k=1}^{\infty} \mathscr{L}(2\text{SUFA}(k\text{-proc}))$ to be exactly the class of unary regular languages. Note that, for binary alphabets, it was shown by Ibarra and Trân (1990) that $\mathscr{L}(2\text{DFA}) \subset \bigcup_{k=1}^{\infty} \mathscr{L}(2\text{SUFA}(k\text{-proc})) \subset \mathscr{L}(2\text{DFA}(2\text{-heads}))$.

On the other hand, we show that synchronization dramatically enhances the power of pushdown automata. In fact, even under the severe restriction of the pushdown store to a counter making only one reversal, synchronized pushdown automata still recognize all recursively enumerable languages.

1. Introduction

Synchronized alternating Turing machines (SATM) were introduced in [4] to study the effect of allowing processes of an alternating Turing machine to communicate via synchronization. Informally, a synchronized alternating machine is an alternating machine with a special subset of internal states called synchronizing states. Each of these synchronizing states is associated with a synchronizing symbol. If, during the course of computation, some process enters a synchronizing state, then it has to wait

Correspondence to: O.H. Ibarra, Department of Computer Science, University of California, Santa Barbara, CA 93106, USA.

*This research was supported in part by NSF Grants CCR89-18409 and DCR90-96221.

until all other processes enter either an accepting state or a synchronizing state with the same synchronizing symbol. When this happens, all processes are allowed to continue their computation; otherwise, the machine is said to have a deadlock. A computation is successful if no deadlocks occur and all processes terminate in accepting states.

It turns out that synchronization significantly increases the computational power of alternating Turing machines [1, 3, 6, 8–11]. In fact, despite the severe restriction of space used to a constant, synchronized alternating finite automata are still very potent. It was shown in [3] that $\mathscr{L}(1\text{SAFA}) = \mathscr{L}(2\text{SAFA}) = \text{NSPACE}(n)$, i.e. one-way/two-way synchronized alternating finite automata recognize exactly the class of context-sensitive languages. This contrasts with the well-known results that neither nondeterminism nor alternation increases the power of finite automata beyond accepting regular languages.

Since then, various restrictions have been placed on synchronized finite automata to make them more realistic models of real-world parallel computers. One way is to restrict the number of processes available (which corresponds to the number of processors) to a constant. Hromkovič et al. [3] nicely characterized one-way and two-way synchronized finite automata with a constant number of processes in terms of multihead nondeterministic finite automata and, thus, obtained tight hierarchies of these machines on the number of processes. It was shown there that, for any $k \geqslant 1$,

(i) $\mathscr{L}(1\text{SAFA}(k\text{-proc})) = \mathscr{L}(1\text{NFA}(k\text{-heads}))$,
$\mathscr{L}(2\text{SAFA}(k\text{-proc})) = \mathscr{L}(2\text{NFA}(k\text{-heads}))$

and, therefore,

(ii) $\mathscr{L}(1\text{SAFA}(k\text{-proc})) \subset \mathscr{L}(1\text{SAFA}((k+1)\text{-proc}))$,
$\mathscr{L}(2\text{SAFA}(k\text{-proc})) \subset \mathscr{L}(2\text{SAFA}((k+1)\text{-proc}))$,

where 1SAFA(k-proc) denotes one-way synchronized alternating finite automata with at most k processes, 1NFA(k-heads) denotes one-way k-head nondeterministic finite automata, and 2SAFA(k-proc) and 2NFA(k-heads) denote the respective two-way counterparts.

Another way to restrict the power of synchronized finite automata is to allow only universal states so that they reflect the deterministic nature of real-world computers. It is natural to ask whether similar characterizations of $\mathscr{L}$(1SUFA(k-proc)) and $\mathscr{L}$(2SUFA(k-proc)) can be found, where 1SUFA(k-proc) denotes one-way synchronized alternating finite automata with only universal states and at most k processes, and 2SUFA(k-proc) denotes the two-way counterpart. Ibarra and Trân [6] showed that in terms of deterministic multihead finite automata (1DFA(k-heads) and 2DFA(k-heads)), no similar nice characterizations exist, since $\mathscr{L}(1\text{DFA}(2\text{-heads})) - \bigcup_{k=1}^{\infty} \mathscr{L}(2\text{SUFA}(k\text{-proc})) \neq \emptyset$, and $\bigcup_{k>0}^{\infty} \mathscr{L}(2\text{SUFA}(k\text{-proc})) \subset \mathscr{L}(2\text{DFA}(2\text{-heads}))$. However, a tight hierarchy of 1SUFA(k-proc) was obtained directly with techniques in Kolmogorov complexity theory. Hence, Ibarra and Trân [6] showed that, for $k \geqslant 2$,

(i) $\mathscr{L}(1\text{SUFA}(k\text{-proc})) \subset \mathscr{L}(1\text{DFA}(k\text{-heads}))$,
$\mathscr{L}(2\text{SUFA}(k\text{-proc})) \subset \mathscr{L}(2\text{DFA}(k\text{-heads}))$

and

(ii) $\mathscr{L}(1SUFA((k-1)\text{-proc})) \subset \mathscr{L}(1SUFA(k\text{-proc}))$,

and left as an open question whether the corresponding hierarchy 2SUFA(k-proc) is tight.

In this paper we answer this question in the positive. In fact, we show an analog of the Yao–Rivest result [12] for 1SUFA($(k+1)$-proc) and 2SUFA(k-proc), namely, $\mathscr{L}(1SUFA((k+1)\text{-proc})) - \mathscr{L}(2SUFA(k\text{-proc})) \neq \emptyset$ for $k \geq 1$, by studying reductions made by two-way deterministic finite automata with a write-only output tape. A tight hierarchy of 2SUFA(k-proc) follows from this result, as well as the corollary that, for every $k \geq 2$, $\mathscr{L}(2SUFA(k\text{-proc}))$ has a complete set under 2DFA reductions and is not closed under intersection, union, or complementation.

Next we show that, when restricted to unary alphabets, $\bigcup_{k=1}^{\infty} \mathscr{L}(2SUFA(k\text{-proc}))$ is exactly the class of unary regular languages. This characterization implies that the hierarchies of 1SUFA(k-proc) and 2SUFA(k-proc) collapse; in contrast, while the hierarchy of 1SAFA(k-proc) collapses, the hierarchy of 2SAFA(k-proc) is proper and tight, since $\mathscr{L}(2NFA(k\text{-heads})) \subset \mathscr{L}(2NFA((k+1)\text{-heads}))$ hold even for unary languages [7].

Finally, we consider the power of synchronized pushdown machines. We show that synchronization dramatically enhances their power, since even under the severe restriction of the pushdown store to a counter making only one reversal, synchronized pushdown automata still recognize all recursively enumerable (r.e.) languages.

The rest of this paper is organized as follows. Section 2 gives definitions relating to synchronized alternating Turing machines and 2DFA reductions. Section 3 states a technical result and uses it to obtain the tight hierarchy of 2SUFA(k-proc), the complete sets under 2DFA reductions, and the nonclosure properties under Boolean operations. It is also shown there that over unary alphabets, $\bigcup_{k=1}^{\infty} \mathscr{L}(2SUFA(k\text{-proc}))$ is exactly the class of regular languages. Section 4 discusses the power of synchronized pushdown automata, and Section 5 concludes the paper with a proof of the technical result.

2. Definitions

A *2DFA transducer* is a 2DFA T with a two-way read-only input tape, a one-way write-only output tape, and Σ and Π as its input and output alphabets, respectively. $L(T) \subseteq \Sigma^*$ denotes the language accepted by 2DFA T, $T(w) \in \Pi^*$ denotes the word generated by T on input w, and $K(T) \subseteq \Pi^*$ denotes $\{T(w)\colon w \in \Sigma^*\}$.

Let $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Pi^*$. We say that L_1 is *2DFA-reducible* to L_2 ($L_1 \leqslant_{2DFA} L_2$) if there is some 2DFA transducer T such that $T(w) \in L_2$ iff $w \in L_1$. A language L is said to be *complete under 2DFA reductions* for a class of languages $\mathscr{C}$ if $L \in \mathscr{C}$ and, for every $C \in \mathscr{C}$, $C \leqslant_{2DFA} L$.

Synchronized alternating Turing machines are formally defined in [6]; we describe only their salient features here. An SATM M is an alternating Turing machine whose

internal state set Q is augmented with a synchronizing alphabet Σ in the following way: each state of M is either an internal state in Q, or a pair of an internal state in Q and a synchronizing symbol in Σ. M accepts x if there is an accepting computation tree T of M on x (in the usual sense for ATM) and, furthermore, each sequence of synchronizing symbols associated with a path in T is a prefix of a common string in Σ^*. Intuitively, the sequences of synchronizing symbols serve to communicate information among the processes at various stages of the computation.

1DFA(k-heads), 1NFA(k-heads), 2DFA(k-heads), 2NFA(k-heads) denote the one-way and two-way versions of deterministic and nondeterministic, k-head finite automata. 1SUFA(k-proc), 1SAFA(k-proc), SUFA(k-proc), SAFA(k-proc) denote the one-way and two-way versions of universal states only and general synchronized alternating finite automata whose computation trees on any input w have at most k leaves. 1NPDA(k-heads) denotes one-way k-head pushdown automata; SA-k-reversal-c-CA, SA-AUX-PDA, SA-AUX-SA, SA-AUX-NESA denote the synchronized versions of c-couter automata with finite reversal, auxiliary pushdown automata, auxiliary stack automata, and auxiliary nonerasing stack automata, respectively (see [5] for formal definitions of these pushdown machines).

For each w in $\Gamma^n = \{w_1 \# w_2 \# \cdots \# w_n\colon w_i \in \{0, 1\}^*\}$, $w(i)$ denotes its ith subword, and $w(i, j)$ denotes the jth symbol of $w(i)$. And, finally, $\subset$ denotes proper inclusion for classes of languages.

3. Main results

We begin by stating a technical result, which we will use to obtain our main and related results. Its rather complex proof appears in Section 5.

Definition 3.1. For each $n \geqslant 1$, let

$$L_n = \left\{0^{e_1} \# 0^{e_2} \# \cdots \# 0^{e_n}\colon e_1 = \max_{1 \leqslant i \leqslant n} e_i\right\},$$

$$K_n = \{x_1 \# x_2 \# \cdots \# x_n \in \Gamma^n\colon \exists i \leqslant n\ \forall j \leqslant n\ (x_j \textit{ is a prefix of } x_i)\}.$$

Theorem 3.2 (technical result). $L_{n+1} \not\leqslant_{\text{2DFA}} K_n$ *for* $n \geqslant 1$.

A tight hierarchy on k for $\bigcup_{k=1}^{\infty} \mathscr{L}$(2SUFA($k$-proc)) follows from this result. In fact, we can show even more.

Theorem 3.3. $\mathscr{L}$(1SUFA((k+1)-proc)) $-$ $\mathscr{L}$(2SUFA(k-proc)) $\neq \emptyset$ *for* $k \geqslant 1$.

Proof. It is easy to see that L_{k+1} can be accepted by a 1SUFA((k+1)-proc) M. On input x, M splits into $k+1$ processes. Each process p_i produces a sequence of synchronizing symbols 0^{l_i}, where l_i is the length of the ith segment of x (we can assume,

without loss of generality, that x has the correct form.) The first process also produces an extra synchronizing symbol 1 at the end of its sequence. Clearly, $x \in L_{k+1}$ iff M accepts x.

On the other hand, if there is some 2SUFA(k-proc) N that accepts L_{k+1}, then N can be converted into a 2DFA transducer N' that reduces L_{k+1} to K_k as follows. First we observe that N has at most k processes and, furthermore, on all inputs, each process of N can be made to halt due to its deterministic nature and the form of strings in L_{k+1}. Then we can construct a halting 2DFA N' that uses a stack of constant size in its finite control to simulate sequentially each of the k processes of N. N' outputs the corresponding synchronizing symbol whenever the simulated process enters a synchronizing state. When a process terminates, N' outputs a # before simulating another process, unless there is no more left.

If at any time N' detects that the process being simulated is looping, then N' outputs k #'s and stops.

Clearly, $x \in L_{k+1}$ iff N' on x outputs a word in K_k and, hence, $L_{k+1} \leqslant_{\text{2DFA}} K_k$. But this contradicts Theorem 3.2. □

Corollary 3.4. *For $k \geqslant 1$, the following are true*:

(i) $\mathscr{L}(\text{1SUFA}(k\text{-proc})) \subset \mathscr{L}(\text{1SUFA}((k+1)\text{-proc}))$,

(ii) $\mathscr{L}(\text{2SUFA}(k\text{-proc})) \subset \mathscr{L}(\text{2SUFA}((k+1)\text{-proc}))$,

(iii) *K_k is complete under 2DFA reductions for* $\mathscr{L}(\text{2SUFA}(k\text{-proc}))$.

Proof. Follows from Theorem 3.3. □

It was shown in [6] that $\mathscr{L}(\text{1SUFA}(k\text{-proc}))$ is not closed under complementation, union, or intersection for $k \geqslant 2$, and that $\mathscr{L}(\text{2SUFA}(k\text{-proc}))$ is not closed under complementation for $k \geqslant 2$. We are now able to show the following corollary.

Corollary 3.5. $\mathscr{L}(\text{2SUFA}(k\text{-proc}))$ *is not closed under intersection or union for $k \geqslant 2$.*

Proof. Let $k \geqslant 2$ and

$$R_k = \left\{ 0^{e_1} \# 0^{e_2} \# \cdots \# 0^{e_{k+1}} : e_1 = \max_{1 \leqslant i \leqslant k} e_i \right\},$$

$$S_k = \left\{ 0^{e_1} \# 0^{e_2} \# \cdots \# 0^{e_{k+1}} : e_1 = \max_{1 \leqslant i \leqslant k-1, k+1} e_i \right\}.$$

Then both R_k and S_k are in $\mathscr{L}(\text{2SUFA}(k\text{-proc}))$, but $R_k \cap S_k = L_{k+1}$ is not. This proves the nonclosure of $\mathscr{L}(\text{2SUFA}(k\text{-proc}))$ under intersection.

Define $U_1 = \{0^{e_1} \# 0^{e_2} \# 0^{e_3} : e_1 \geqslant e_2 \ \& \ e_1, e_2, e_3 \geqslant 0\}$, $U_2 = \{0^{e_1} \# 0^{e_2} \# 0^{e_3} : e_1 \geqslant e_3 \ \& \ e_1, e_2, e_3 \geqslant 0\}$, and $U = U_1 \cup U_2$. Clearly, both U_1 and U_2 are in $\mathscr{L}(\text{2SUFA}(2\text{-proc}))$, but, by Corollary 5.1, U is not in $\mathscr{L}(\text{2SUFA}(k\text{-proc}))$ for any $k \geqslant 1$. This proves the nonclosure of $\mathscr{L}(\text{2SUFA}(k\text{-proc}))$ under union. □

For the remaining of this section, we restrict our consideration to unary alphabets. We first give a characterization of $\bigcup_{k=1}^{\infty} \mathscr{L}(2\text{SUFA}(k\text{-proc}))$.

Theorem 3.6. *$\mathscr{L}(2\text{SUFA}(k\text{-proc}))$ over unary alphabets is the class of regular sets for all $k \geqslant 1$.*

Proof. Recall that input alphabets are unary. It was shown in [2] that $\bigcup_{k=1}^{\infty} 1\text{NPDA}(k\text{-heads})$ accepts only regular languages. Hence, it suffices to show that if L is accepted by some 2SUFA(k-proc) M then $\bar{L}$ is accepted by some 1NPDA(k'-heads) M'. Without loss of generality, we can assume that M, after some deterministic moves from the start of the computation, splits into k deterministic processes. Since the input alphabet is unary and the processes of M are deterministic, the input head of each process can change direction only at the first or last c squares of its input, and at most t times without going into a loop, for some constants c and t depending only on M (see Lemma 5.2 for details). Hence, M can be modified so that its processes never get into an infinite loop on any input.

If M rejects input x, then either one of its processes finishes in a nonaccepting state, or the synchronizing sequences of two processes differ at some bit. In the former case, M' needs only to choose a process and verify that it halts in a nonaccepting state. Below we show how M' on input x verifies the latter condition. First, M' nondeterministically chooses a process to simulate. Since a process of M changes the direction of its input head at most t times and only at the first or last c squares, M' needs at most t one-way heads to perform the simulation by using a new head every time the simulated process reverses the direction of its input head. Furthermore, whenever the process enters a synchronizing state, M' pushes a symbol on its pushdown stack. At some point, M' decides that the last synchronizing symbol witnesses a discrepancy, remembers it in its finite control, and nondeterministically chooses another process to simulate. This time M' pops a symbol from its pushdown stack every time the second process enters a synchronizing state. M' accepts iff there is indeed a discrepancy between the two synchronizing sequences. Note that M' uses at most $2t$ one-way input heads. □

Corollary 3.7. *Let the input alphabet be unary. For $k \geqslant 1$, the following are true:*

(i) $\mathscr{L}(1\text{SUFA}(k\text{-proc})) = \mathscr{L}(1\text{SUFA}((k+1)\text{-proc}))$,

(ii) $\mathscr{L}(2\text{SUFA}(k\text{-proc})) = \mathscr{L}(2\text{SUFA}((k+1)\text{-proc}))$,

(iii) $\mathscr{L}(1\text{SAFA}(k\text{-proc})) = \mathscr{L}(1\text{SAFA}((k+1)\text{-proc}))$,

(iv) $\mathscr{L}(2\text{SAFA}(k\text{-proc})) \subset \mathscr{L}(2\text{SAFA}((k+1)\text{-proc}))$.

Proof. (i) and (ii) follow from Theorem 3.6. (iii) follows from the fact that a 1SAFA(k-proc) can be simulated by a 1NPDA(k-heads) and from [2]. (iv) follows from the characterization of 2SAFA(k-proc) as 2NFA(k-heads) [3] and from the result that $\mathscr{L}(2\text{NFA}(k\text{-heads})) \subset \mathscr{L}(2\text{NFA}((k+1)\text{-heads}))$ over unary alphabets [7]. □

4. Synchronized pushdown automata

We have seen that the accepting power of nondeterministic multihead pushdown automata is very limited over unary alphabets. It is surprising that the accepting power of synchronized pushdown automata is on the other end of the computational spectrum, namely, synchronized pushdown automata recognize all recursively enumerable sets. In fact, the following theorem shows that even for a severely restricted model of pushdown automata, the claim still holds.

Theorem 4.1. *Every r.e. language can be accepted by a synchronized alternating one-reversal one-counter machine.*

Proof. It suffices to show that every deterministic two-counter machine M can be simulated by a synchronized alternating one-reversal one-counter machine M', since deterministic two-counter machines can recognize all r.e. languages [5]. At the beginning, M' splits into five processes h, p_1, p_2, q_1 and q_2; the difference of p_1 and p_2 represents the content of counter c_1 of M; the difference of q_1 and q_2 represents the content of the other counter c_2. Process h represents the state and input head position of M.

Every move by M is simulated by M' as follows. Process h first guesses whether the counters c_1 and c_2 are empty, and then produces the synchronizing symbol $[q, a, s_1, s_2]$, where q is its current state, a is the symbol under its input head, and s_1 and $s_2 \in \{0, 1\}$ denote whether the counters are empty. Each of p_1 and p_2 tries to guess and produce the symbol $[q, a, s_1, s_2]$. Furthermore, p_1 and p_2 need to verify that s_1 reflects correctly the state of counter c_1, which together they represent. Let us say that they have to verify that c_1 is not empty. The next paragraph explains how this can be done. To verify that c_1 is empty, a slightly modified method can be used.

To perform the verification, p_1 spawns an identical process u_1, and p_2 spawns an identical process u_2. Process u_1 then produces a special synchronizing symbol B and then deterministically produces a special synchronizing symbol l for every symbol it pops off its own counter until it becomes empty. Finally, u_1 produces a special synchronizing symbol E and halts in an accepting state. Similarly, process u_2 produces a B and then produces an l for every symbol it pops off its own counter, until at some point u_2 nondeterministically decides that it has produced the same number of l as u_1, and its own counter is still not empty. It then produces an E and halts in an accepting state. Meanwhile, the other processes h, p_1, p_2, q_1, q_2 guess and produce a sequence of synchronizing symbols of the form Bl^iE for some $i \geqslant 0$; their counters remain the same during this period.

Next, each of q_1 and q_2 tries to guess and produce the symbol $[q, a, s_1, s_2]$ and, furthermore, they need to verify that s_2 reflects correctly the state of counter c_2, which together they represent. Again, the verification is as described above for p_1 and p_2.

After s_1 and s_2 have been verified, h, p_1, p_2, q_1, q_2 update their configurations according to the next move of M dictated by $[q, a, s_1, s_2]$. Process h moves its head

and updates its internal state to reflect the new state and input head position of M. To increment counter c_1, p_1 does nothing while p_2 increments its counter. To decrement counter c_1, p_1 increments its counter while p_2 does nothing. Counter c_2 can be decremented or incremented with the same method. If q is an accepting state, then h, p_1, p_2, q_1, q_2 halt in accepting states.

Now, it is clear that there is an accepting computation tree of M' (in which all processes accept and synchronizing sequences of h, p_1, p_2, q_1, q_2 and of the processes they spawn are prefixes of a common string) iff all processes accept, all corresponding guesses of synchronizing symbols $[a, q, s_1, s_2]$ match, and synchronizing subsequences $B0^iE$ generated by all processes when verifying the validity of an s_1 or s_2 have the same length. This occurs iff the synchronizing symbols $[a, q, s_1, s_2]$ reflect correctly the configurations of M during an accepting computation, iff M accepts its input. Furthermore, the only processes of M' that make a reversal on its counter are those spawned to verify the status of s_1 or s_2, and they make exactly one reversal.

Hence, every r.e. language can be recognized by a one-counter one-reversal synchronized counter machine. □

Corollary 4.2. *For any $k, c \geqslant 1$ and any space-bound function $s(n)$,* SA-k-*reversal-*c-CA, SA-AUX-PDA($s(n)$), SA-AUX-SA($s(n)$), SA-AUX-NESA($s(n)$) *accept the class of r.e. languages.*

5. Proof of the technical result

We prove in this section the technical result mentioned and used in Section 3. Our objective is to show that $L_{n+1} \not\leqslant_{\text{2DFA}} K_n$ for $n \geqslant 1$. We begin with the trivial case when $n = 1$.

Lemma 5.1. $L_2 \not\leqslant_{\text{2DFA}} K_n$.

Proof. If there is some 2DFA transducer T that reduces L_2 to K_1 then we can modify T to obtain a 2DFA T' which accepts L_2 whenever T generates a word in $\{0, 1\}^* = K_1$. But this means that L_2 must be regular, a contradiction. □

The proof is more involved when $n \geqslant 2$. In the following, let T be a 2DFA transducer of q states with $\Sigma = \Pi = \{0, 1, \#\}$ and $a \geqslant q + 1$. We will be concerned with inputs in the special form $x = 0^{a+e_1 q!} \# 0^{a+e_2 q!} \# \cdots \# 0^{a+e_n q!} \in \Gamma^n$. The next four lemmas establish the form of $T(x)$ for such inputs.

Lemma 5.2. *Suppose T enters the input segment $s = \# 0^{a+iq!} \#$, $i \geqslant 0$, from the left (right) in state p_0, first exits s in some state p_1, and, during that period, generates the output segment w. Then $w = uv^i x$ for some u, v, and x depending only on a and p_0. Further, the position and state of T when it exits s depend only on a and p_0.*

Proof. We assume, without loss of generality, that T enters s from the left; the other case is symmetric. Let T enter s at time t_0 and first exit s at time t_f. If, after entering s in state p_0, T never moves to the right of the ath zero then, because it is deterministic, T never moves to the right of the ath zero of $\#0^{a+iq!}\#$ for any $i \geqslant 0$ after entering it in state p_0. In this case, let $u=w$, and $v=x=\varepsilon$.

On the other hand, if T visits more than a zeros after entering s, then it must first visit the $(q+2)$th zero (counting from the left) at some time $t \geqslant t_0+q+2$ steps. Let p_i be the state T is in at time t_i, the last time T visits the ith zero before time t, $1 \leqslant i \leqslant q+1$. Since T has only q states, there are some $j<k$ such that $p_j=p_k$. Also, during the time period (t_j, t_k), T does not move to the left of the jth zero. Since T is deterministic and the input is unary, T repeats this "loop" of computation (moving to the right $k-j$ squares after t_k-t_j steps) until it reaches the right $\#$.

Let $l=q!/(k-j)$. There are at least il iterations of this loop in the computation of T on s. In this case, let u, v', x be the words T outputs on input s during time periods (t_0, t_k), (t_j, t_k), and $(t_k+(t_k-t_j)il+1, t_f)$, respectively, and $v=(v')^l$.

Finally, note that whether T exits s from the left or right side depends only on a and p_0, and that varying i changes only the number of loops in the computation and not the final state p_1. □

Lemma 5.3. *On input* $x\#0^{a+iq!}\#y$, $i \geqslant 0$, T *generates* $w=c_1v_1^ic_2v_2^i\ldots c_mv_m^ic_{m+1}$, *where* m, $\{v_k \neq \varepsilon\colon 1 \leqslant k \leqslant m\}$, *and* $\{c_k\colon 1 \leqslant k \leqslant m+1\}$ *depend only on* T, x, y, *and* a.

Proof. Consider the computation of T on $x\#0^{a+iq!}\#y$, $i \geqslant 0$. T enters and exits $\#0^{a+iq!}\#$ m' times, where m' does not depend on i, because the final position and state of T each time it exits s does not depend on i, by Lemma 5.2. Then the word that T outputs on $x\#0^{a+iq!}\#y$ can be expressed as $b_1z_1b_2z_2\ldots b_{m'}z_{m'}b_{m'+1}$, where z_k, $1 \leqslant k \leqslant m'$, is the word T outputs while visiting $\#0^{a+iq!}\#$ the kth time, and b_k, $1 \leqslant k \leqslant m'+1$, are the words T outputs while visiting x or y. By Lemma 5.2, each z_k can be written as $u_kv_k^ix_k$, $1 \leqslant k \leqslant m'$. The lemma follows from setting $x_0=\varepsilon$, and $c_k=x_{k-1}b_ku_k$, $1 \leqslant k \leqslant m'+1$, and combining c_k and c_{k+1} whenever $v_k=\varepsilon$.

Lemma 5.4. *On input* $x=0^{a+e_1q!}\#0^{a+e_2q!}\#\cdots\#0^{a+e_nq!}$, *where* $e_k \geqslant 0$ *for* $1 \leqslant k \leqslant n$, T *generates* $w=c_1v_1^{f_1}c_2v_2^{f_2}\ldots c_mv_m^{f_m}c_{m+1}$, *where* $f_k \in \{e_k\colon 1 \leqslant k \leqslant n\}$ *for* $1 \leqslant k \leqslant m$, *and* m, $\{c_k\colon 1 \leqslant k \leqslant m+1\}$, $\{v_k \neq \varepsilon\colon 1 \leqslant k \leqslant m\}$ *depend only on* T, a, *and* n.

Proof. Follows from repeated applications of Lemma 5.3. □

The last three lemmas lead to the normal-form theorem for output words $T(x)$, where $x=0^{a+e_1q!}\#0^{a+e_2q!}\#\cdots\#0^{a+e_nq!}$.

Theorem 5.5 (Normal form). *Suppose* T *reduces* L_n *to* $K_{n'}$ *for some* $n, n' \geqslant 1$. *Then on input* $x=0^{a+e_1q!}\#0^{a+e_2q!}\#\cdots\#0^{a+e_nq!}$, *where* $e_k \geqslant 0$ *for* $1 \leqslant k \leqslant n$, T *generates* $w=w(1)\#w(2)\#\cdots\#w(n')\in\Gamma^{n'}$, *where each* $w(k)$, $1 \leqslant k \leqslant n'$, *has the form described in Lemma* 5.4.

Proof. By Lemma 5.4, on input x, T generates $w=c_1v_1^{f_1}c_2v_2^{f_2}\cdots c_mv_m^{f_m}c_{m+1}$, where $f_k\in\{e_k\colon 1\leqslant k\leqslant n\}$ for $1\leqslant k\leqslant m$, and m, $\{c_k\colon 1\leqslant k\leqslant m+1\}$, $\{v_k\colon 1\leqslant k\leqslant m\}$ depend only on T, a, and n. Furthermore, since T is a reduction function to $K_{n'}$, no v_k can contain a # symbol or else T generates too many #'s for some word in L_n. Hence, there are exactly $n'-1$ #'s in w, and each # belongs to some c_k. Since no v_k is split, w can expressed as $w(1)\#w(2)\#\cdots\#w(n')$, each subword $w(k)$ having the form described in Lemma 5.4. □

From now on, T, x, and y will be as defined in Theorem 5.5; furthermore, since the proof of Theorem 5.5 shows that each e_i is independent of $|v_1|,|v_2|,\ldots,|v_m|$, we may assume that $e_i>2L(y)$ for each i, where $L(y)=2\sum_{1\leqslant k\leqslant m}|v_k|+1$.

Now, since Theorem 5.5 guarantees that $y\in\Gamma^{n'}$, $y\notin K_{n'}$ only when two of its subwords differ at some bit. When this is true, we say that y has a *conflict point*. Suppose we change the value of some e_i of x to obtain x'; will $y'=T(x')$ have a conflict point too, and how does it relate to conflict points of y? The next definition and lemma give us some tools to deal with this question and its consequences.

Definition 5.6. Suppose $y(i)=c_1v_1^{e_k}c_2v_2^{e_k}\cdots y(i,j)\cdots c_rv_r^{e_k}c_{r+1}$, where $v_1,\ldots,v_r$ are the loops generated by $0^{a+e_kq!}$ (called k-loops) in $y(i)$ for some i,j, and k. Define

$$l(y,i,j,k)=\begin{cases}\sum\limits_{1\leqslant t\leqslant s}|v_t| & \text{if } c_{s+1}=w_1y(i,j)w_2,\\ \sum\limits_{1\leqslant t\leqslant s}|v_t| & \text{if } v_s^{e_k}=v_s^{L(y)}w_1y(i,j)w_2,\\ \sum\limits_{1\leqslant t<s}|v_t| & \text{if } v_s^{e_k}=w_1y(i,j)w_2v_s^{e_k-L(y)}.\end{cases}$$

If $x\notin L_n$, then there are some i_1 and i_2, $1\leqslant i_1<i_2\leqslant n$, and j such that $y(i_1,j)\neq y(i_2,j)$. We say that j is a *conflict point* of y (between subwords i_1 and i_2). We say that j is *k-invariant* if $l(y,i_1,j,k)=l(y,i_2,j,k)$, and *$k$-variant* otherwise.

Intuitively, if $T(x)$ has a k-invariant conflict point, then $T(x')$ also has a conflict point, where x' is obtained from x by slightly varying e_k. The following lemma formalizes this idea by describing the properties of the function l.

Lemma 5.7. *Let* $x=0^{a+e_1q!}\#\cdots\#0^{a+e_kq!}\#\cdots\#0^{a+e_nq!}$ *and* $y=T(x)$. *For* $t\leqslant L(y)$, *let* $x_{-t}=0^{a+e_1q!}\#\cdots\#0^{a+(e_k-t)q!}\#\cdots\#0^{a+e_nq!}$, $x_t=0^{a+e_1q!}\#\cdots\#0^{a+(e_k+t)q!}\#\cdots\#0^{a+e_nq!}$, $y_{-t}=T(x_{-t})$, *and* $y_t=T(x_t)$. *Then, for all* i,j,j_1,j_2,i_1, *and* i_2, *the following hold:*

(i) $L(y)=L(y_{-t})=L(y_t)$,
$l(y,i,j,k)<L(y)$,
if $j_1\leqslant j_2$ *then* $l(y,i,j_1,k)\leqslant l(y,i,j_2,k)$;

(ii) *if* $j_2-j_1\geqslant L(y)$ *then* $j_2-l(y,i,j_2,k)>j_1-l(y,i,j_1,k)$;

(iii) $y_{-t}(i,j-tl(y,i,j,k))=y(i,j)$;

(iv) $y_t(i,j+tl(y,i,j,k))=y(i,j)$;

(v) $l(y_t, i, j+tl(y,i,j,k), k') = l(y,i,j,k')$ *for all* k';

(vi) $l(y_{-t}, i, j-tl(y,i,j,k), k') = l(y,i,j,k')$ *if* $k' \neq k$,
$l(y_{-1}, i, j-l(y,i,j,k)+s, k) = l(y,i,j,k)$ *for some* s *equal to zero or the length of the* k*-loop* $y(i,j)$ *belongs to*;

(vii) *if* $0 \leqslant j_2 - j_1 < L(y)$ *then* $y_{-1}(i, j_1 - l(y,i,j_2,k)) = y(i,j_1)$ *and* $y_{-1}(i, j_2 - l(y,i,j_1,k)) = y(i,j_2)$.

Proof. (i) Immediate from definition.

(ii) $(j_2-j_1)-(l(y,i,j_2,k)-l(y,i,j_1,k)) > L(y)-L(y)=0$.

(iii) When e_k is decremented by t, y_{-t} is obtained from y by removing t instances of every k-loop in y. If $y(i,j)$ belongs to a k-loop, then we remove the leftmost or rightmost t instances of that loop, depending on whether $l(y,i,j,k)$ includes the length of that loop or not. The definition of l ensures that there are enough instances to be removed in both cases. Clearly, $y_{-t}(i, j-tl(y,i,j,k)) = y(i,j)$.

(iv) Similar to (iii).

(v) Immediate from the definition of l if $k \neq k'$. If $k = k'$, adding instances does not change the number or the lengths of k-loops strictly preceding $y(i,j)$. Furthermore, if $y(i,j)$ belongs to a k-loop, then there are at least $L(y)$ instances to the left of $y(i,j)$ iff there are at least $L(y)$ instances to the left of $y_t(i, j+tl(y,i,j,k))$. Hence, $l(y,i,j,k) = l(y_t, i, j+tl(y,i,j,k), k)$.

(vi) Immediate from the definition of l if $k \neq k'$. If $k = k'$, removing instances does not change the number or the lengths of k-loops strictly preceding $y(i,j)$. If $y(i,j)$ belongs to the $(L(y)+1)$th instance of some k-loop of length s, then $y_{-1}(i, j-l(y,i,j,k))$ belongs to the $L(y)$th instance of that k-loop. In this case, $y_{-1}(i, j-l(y,i,j,k)+s)$ belongs to the $(L(y)+1)$th instance, and $l(y_{-1}, i, j-l(y,i,j,k)+s, k) = l(y,i,j,k)$. Otherwise, $l(y_{-1}, i, j-l(y,i,j,k), k) = l(y,i,j,k)$.

(vii) Let $l_1 = l(y,i,j_1,k)$ and $l_2 = l(y,i,j_2,k)$. If $l_1 = l_2$ the result follows from (iii). Else, since $0 \leqslant j_2 - j_1 < L(y)$, it must be the case that $y(i,j_1)$ and $y(i,j_2)$ belong to the same k-loop, where $s = l_2 - l_1$ is the length of that loop. Furthermore, $y_{-1}(i,j_1)$ belongs to a nonleftmost instance, and $y_{-1}(i,j_2)$ belongs to a nonrightmost instance, since $e > 2L(y)$. Hence, $y_{-1}(i, j_1 - l_2) = y_{-1}(i, j_1 - l_1 - s) = y_{-1}(i, j_1 - l_1) = y(i,j_1)$. Similarly, $y_{-1}(i, j_2 - l_1) = y_{-1}(i, j_2 - l_2 + s) = y_{-1}(i, j_2 - l_2) = y(i,j_2)$. □

We are now ready to describe the nature of conflict points of $T(x)$ for inputs x of a special form.

Lemma 5.8. *Let* $x = 0^{a+2eq!} \# 0^{a+(2e+1)q!} \# 0^{a+eq!} \# \cdots \# 0^{a+eq!} \in \Gamma^n$, $y = T(x)$, *and* $e > 2L(y)$. *Let* c *be a conflict point of* y *between subwords* i_1 *and* i_2. *Then* c *is* 1*-variant,* 2*-variant, and* k*-invariant for* $3 \leqslant k \leqslant n$.

Proof. Define $\delta_k = l(y, i_1, c, k) - l(y, i_2, c, k)$ for $1 \leqslant k \leqslant n$. Note that $\delta_k < L(y)$ for all k. Consider the diophantine equation

$$\sum_{k=1}^{n} \delta_k u_k = 0. \tag{1}$$

If $(u_1, u_2, \ldots, u_n)$ is a solution to (1), and $|u_k| \leqslant L(y)$ for all k then, by Lemma 5.7(iii)–(vi), $T(0^{a+(2e+u_1)q!} \# 0^{a+(2e+1+u_2)q!} \# 0^{a+(e+u_3)q!} \# \cdots \# 0^{a+(e+u_n)q!})$ has a conflict point at $(c + \sum_{k=1}^{n} l(y, i_1, c, k)u_k)$ between subwords i_1 and i_2.

First, it follows from (1) that $\delta_1 \neq 0$, or else $(1, 0, \ldots, 0)$ is a solution to (1) and, therefore, $x' = 0^{a+(2e+1)q!} \# 0^{a+(2e+1)q!} \# 0^{a+eq!} \# \cdots \# 0^{a+eq!} \notin L_n$, a contradiction. Similarly $\delta_2 \neq 0$, or else $(0, -1, 0, \ldots, 0)$ is a solution to (1) and, therefore, $x' = 0^{a+2eq!} \# 0^{a+2eq!} \# 0^{a+eq!} \# \cdots \# 0^{a+eq!} \notin L_n$, a contradiction. Hence, c is 1-variant and 2-variant.

Now suppose $\delta_s \neq 0$ for some $s \geqslant 2$. If δ_s and δ_1 have the same sign, then $(u_1, u_2, \ldots, u_n)$, where $u_1 = |\delta_s|$, $u_s = -|\delta_1|$, and every other $u_k = 0$, is a solution to (1). But then the word obtained from x by adding $|\delta_s|q!$ to the first exponent and reducing $-|\delta_1|q!$ from the sth exponent is not in L_n, a contradiction. So, the signs of δ_1 and δ_s are different for $s \geqslant 2$.

So, if $\delta_s \neq 0$ for some $s \geqslant 3$, then δ_s and δ_2 have the same sign. Hence, $(u_1, u_2, \ldots, u_n)$, where $u_1 = 0$, $u_2 = -|\delta_s|$, $u_s = |\delta_2|$, and $u_k = 0$ for all other k, is a solution to (1). But then the word obtained from x by removing $|\delta_s|q!$ from the second exponent and adding $|\delta_2|q!$ to the sth exponent is not in L_n. This is a contradiction because $e + |u_s| < e + L(y) < 2e$. Hence, c is k-invariant for $3 \leqslant k \leqslant n$. □

Theorem 5.9. *There exist i_1 and i_2 such that, for all $x' = 0^{a+2eq!} \# 0^{a+(2e+1)q!} \# 0^{a+e_3 q!} \# \cdots \# 0^{a+e_n q!} \in \Gamma^n$, where $2L(y) < e \leqslant e_i$ for $3 \leqslant i \leqslant n$, $y' = T(x')$ has a conflict point between subwords i_1 and i_2. Furthermore, the first such conflict point is 1-variant, 2-variant, and k-invariant for all other k.*

Proof. We use induction on $e_3, \ldots, e_n$. When $e_3 = \cdots = e_n = e$, all conflict points of $T(x)$ are 1-variant, 2-variant, and k-invariant, $3 \leqslant k \leqslant n$, by Lemma 5.8. In fact, the same proof applies when $e_i + L(y) \leqslant 2e$, $3 \leqslant i \leqslant n$. This establishes the basis. Choose and fix two subwords i_1 and i_2 that witness a conflict point.

For the induction step, let $x_t = 0^{a+2eq!} \# 0^{a+(2e+1)q!} \# 0^{a+(e_3+t)q!} \# 0^{a+e_4 q!} \# \cdots \# 0^{a+e_n q!}$ and suppose the theorem is true for x_0 and x_1. We show that the theorem is true also for x_2. Let y_0, y_1, y_2 be $T(x_0)$, $T(x_1)$, $T(x_2)$, and let c_1 be the first conflict point of y_1 between i_1 and i_2. By Lemma 5.7(iv) and (v), $c_2 = c_1 + l(y_1, i_1, c_1, 3)$ is a conflict point of y_2 between i_1 and i_2, and c_2 is 1-variant, 2-variant, and k-invariant for all other k.

If c_2 is the first conflict point of y_2 between i_1 and i_2, then we are done. Else, let $c < c_2$ be the first such conflict point. We note that $c_2 - c \geqslant L(y)$, or else, by Lemma 5.7(vii), $c - l(y_2, i_1, c_2, 3) < c_2 - l(y_2, i_1, c_2, 3) = c_1$ is a conflict point of y_1 that comes before c_1, a contradiction. Also, c must be 3-variant, or else, by Lemma 5.7(ii), $c - l(y_2, i_1, c, 3) < c_2 - l(y_2, i_1, c_2, 3) = c_1$ is a conflict point of y_1, a contradiction.

So,

$$c_2 \geqslant c + L(y) \tag{2}$$

and, without loss of generality, $l(y_2, i_1, c, 3) > l(y_2, i_2, c, 3)$. Let l_1 and l_2 denote the last two quantities, respectively, $p_1 = c - l_1$ and $p_2 = c - l_2$. Then

$$p_2 - l_1 = p_1 - l_2. \tag{3}$$

Since $p_1 = c - l_1 < c - l_1 + L(y) - l(y_2, i_1, c_2, 3) \leqslant c_2 - l(y_2, i_1, c_2, 3) = c_1$ and similarly $p_2 < c_1$, and since c_1 is the first conflict point of y_1,

$$y_1(i_2, p_1) = y_1(i_1, p_1) = y_2(i_1, c) \neq y_2(i_2, c) = y_1(i_2, p_2) = y_1(i_1, p_2). \tag{4}$$

By Lemma 5.7(vi) there exist some s_1 and s_2, which are equal to either zero or the lengths of some k-loops, such that $l(y_1, i_1, p_1 + s_1, 3) = l_1$ and $l(y_1, i_2, p_2 + s_2, 3) = l_2$. Since $|p_1 + s_1 - p_2|$ and $|p_2 + s_2 - p_1|$ are both less than $L(y)$, by Lemma 5.7(vii), we have

$$y_1(i_1, p_2) = y_0(i_1, p_2 - l_1) \tag{5}$$

and

$$y_1(i_2, p_1) = y_0(i_2, p_1 - l_2). \tag{6}$$

But then $p_2 - l_1 = p_1 - l_2$ (from (3)) is a conflict point of y_0 between i_1 and i_2, because from (4)–(6) we have $y_0(i_1, p_2 - l_1) \neq y_0(i_2, p_1 - l_2)$.

Let c_0 be the first conflict point of y_0 between i_1 and i_2. By induction hypothesis, c_0 and c_1 are 3-invariant. Hence, $c_1 = c_0 + l(y_0, i_1, c_0, 3)$ or, equivalently, $c_0 = c_1 - l(y_1, i_1, c_1, 3)$. Otherwise, if $0 < c_0 + l(y_0, i_1, c_0, 3) - c_1 < L(y)$ then, by Lemma 5.7(vii), $c_1 - l(y_0, i_1, c_0, 3) < c_0$ is a conflict point of y_0 that comes before c_0, a contradiction; if $c_0 + l(y_0, i_1, c_0, 3) - c_1 \geqslant L(y)$ then by Lemma 5.7(ii), $c_1 - l(y_1, i_1, c_1, 3) < c_0$ is again a conflict point of y_0 that comes before c_0, a contradiction.

Combining this with the fact that $c_1 = c_2 - l(y_2, i_1, c_2, 3)$, we have

$$c_0 = c_2 - l(y_2, i_1, c_2, 3) - l(y_1, i_1, c_1, 3) \leqslant p_2 - l_1 = c - l_2 - l_1$$

and, hence,

$$c_2 < c - l_2 - l_1 + L(y) \leqslant c + L(y),$$

which contradicts (2).

So, no such c could exist and, therefore, c_2 must be the first conflict point of y_2 between i_1 and i_2. This proves the theorem. □

Figure 1 illustrates the idea in Theorem 5.9. Note that Theorem 5.9 still holds when the second segment of x is swapped with the kth segment for $k \geqslant 3$.

Theorem 5.10. *$L_{n+1} \not\leqslant_{\text{2DFA}} K_n$ for $n \geqslant 2$.*

Proof. Fix an $n \geqslant 2$ and suppose there is some 2DFA transducer T that reduces L_{n+1} to K_n. Let $x = 0^{a+2eq!} \# 0^{a+(2e+1)q!} \# \cdots \# 0^{a+(2e+1)q!} \in \Gamma^{n+1}$, $y = T(x)$, and $e > 2L(y)$. Then it follows from Theorem 5.9 that, for each $k \geqslant 2$, there exist i_k and j_k such that the first conflict point c_k of $y \in \Gamma^n$ between i_k and j_k is 1-variant, k-variant, and k'-invariant for all other k'. We say that c_k is only 1-variant and k-variant. There are such n conflict points.

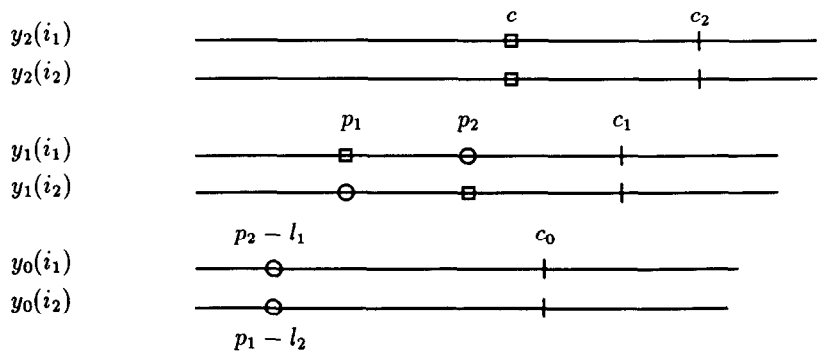

Fig. 1. Conflict points between subwords i_1 and i_2 of y_0, y_1, and y_2.

Partition $\{2,\ldots,n+1\}$ into subsets $C_1, C_2, \ldots, C_m$, such that if s and t are in the same subset then $c_s = c_t$. For each such subset S construct a graph G whose vertices are i_s and j_s nd whose edges connect i_s and j_s for all $s \in S$. G must be acyclic; otherwise, there is a cycle $v_1, v_2, \ldots, v_l$, and, without loss of generality, subwords v_1 and v_2 are only 1-variant, 2-variant, v_2 and v_3 are only 1-variant, 3-variant, ..., v_l and v_1 are only 1-variant, $(l+1)$-variant, with respect to the conflict point representing S. But the path from v_1 to v_l implies that they are $(l+1)$-invariant, a contradiction.

It is impossible to place such n conflict points on n subwords: we start with the largest conflict points, say those in C_1, and select subwords i_1 and j_s for all $s \in C_1$. $|C_1| + p_1$ subwords must be selected, where p_1 is the number of connected components of the graph for C_1, since each conflict point contributes a new vertex to a unique component, except for the first of that component, which contributes two. Afterward, all selected subwords in a component are effectively the same because $c_1, c_2, \ldots, c_{|C_1|}$ are the *first* conflict points for those subwords. Hence, after the first $|C_1|$ conflict points have been selected, only $n - |C_1| - p_1 + p_1 = n - |C_1|$ subwords remain for $n - |C_1|$ conflict points. Continue this process until only $|C_m|$ subwords remain for the last set C_m. But we need at least $|C_m| + 1$ subwords.

Hence, $L_{n+1} \not\leqslant_{\text{2DFA}} K_n$ for $n \geqslant 2$. □

Define $U = \{0^{e_1} \# 0^{e_2} \# 0^{e_3} \colon e_1 \geqslant e_2 \text{ or } e_1 \geqslant e_3\}$. Nonclosure under union for $\mathscr{L}$(2SUFA(k-proc)), $k \geqslant 2$, follows from the next corollary and the observation that U is the union of two languages in $\mathscr{L}$(2SUFA(2-proc)).

Corollary 5.11. *$U \not\leqslant_{\text{2DFA}} K_n$ for any $n \geqslant 1$.*

Proof. Suppose T is a 2DFA transducer that reduces U to K_n for some fixed $n \geqslant 1$. Let $x = 0^{a+eq!} \# 0^{a+(e+1)q!} \# 0^{a+(e+1)q!}$, $y = T(x)$, and $e > 2L(y)$. Since $x \notin U$, y must have a conflict point c. Using arguments similar to those in Lemma 5.8, we can show that δ_1, δ_2, and δ_3 are not zero, and that the sign of δ_1 is different from that of δ_2 and δ_3.

Hence, $x' = 0^{a+eq!} \# 0^{a+(e+|\delta_3|)q!} \# 0^{a+(e-|\delta_2|)q!}$ also has a conflict point, but x' is in U, a contradiction. Therefore, $U \not\leqslant_{2DFA} K_n$ for any $n \geqslant 1$. □

Acknowledgment

We thank Tao Jiang for a very careful reading of a preliminary version of this paper. Valuable discussion with him led to the proof of Theorem 5.10.

References

[1] J. Dassow, J. Hromkovič, J. Karhumäki, B. Rovan and A. Slobodová, On the power of synchronization in parallel computations, in: *Proc. 14th MFCS '89*, Lecture Notes in Computer Science, Vol. 379 (Springer, Berlin, 1989) 196–206.

[2] M.A. Harrison and O.H. Ibarra, Multi-tape and multi-head pushdown automata, *Inform. and Control* **13** (1968) 433–470.

[3] J. Hromkovič, J. Karhumäki, B. Rovan and A. Slobodová, On the power of synchronization in parallel computations, Tech. Report, Comenius Univ., Bratislava, Czechoslovakia, Dept. of Theoretical Cybernetics and Institute of Computer Science, 1989.

[4] J. Hromkovič, How to organize the communication among parallel processes in alternating computations, manuscript, January 1986.

[5] J. Hopcroft and J. Ullman, Introduction to Automata Theory, Languages, and Computation (Addison-Wesley, Reading, MA, 1979).

[6] O.H. Ibarra and N.Q. Trân, On space-bounded synchronized alternating Turing machines, *Theoret. Comput. Sci.* **99** (1992) 243–264.

[7] B. Monien, Two-way multihead automata over a one-letter alphabet, *RAIRO Inform. Theory* **14** (1980) 67–82.

[8] A. Slobodová, On the power of communication in alternating computations, Student Research Papers Competition, Section Computer Science (in Slovac), Comenius Univ., Bratislava, Czechoslovakia, April 1987.

[9] A. Slobodová, On the power of communication in alternating machines, in: *Proc. 13th MFCS '88*, Lecture Notes in Computer Science, Vol. 324 (Springer, Berlin, 1988) 518–528.

[10] A. Slobodová, Some properties of space-bounded synchronized alternating Turing machines with only universal states, in: *Proc. 5th IMYCS '88*, Lecture Notes in Computer Science, Vol. 381 (Springer, Berlin, 1988) 102–113.

[11] J. Wiedermann, On the power of synchronization, Tech. Report, VUSEIAR, Bratislava, Czechoslovakia, 1987.

[12] A.C.C. Yao and R.L. Rivest, $k+1$ heads are better than k, *J. ACM* **25** (1978) 337–340.