ELSEVIER Theoretical Computer Science 169 (1996) 161–184

Theoretical Computer Science

Complexity results for two-way and multi-pebble automata and their logics

Noa Globerman[a,*], David Harel[b,1]

[a] *Department of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan, Israel*
[b] *Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel*

Abstract

Two-way and multi-pebble automata are considered (the latter appropriately restricted to accept only regular languages), and enriched with additional features, such as nondeterminism and concurrency. We investigate the succinctness of such machines, and the extent to which this succinctness carries over to make the reasoning problem in propositional dynamic logic more difficult. The two main results establish that each additional pebble provides inherent exponential power on both fronts.

1. Introduction

1.1. Background

This paper continues our work in [5, 9, 10] seeking exponential (or higher) discrepancies in the succinctness of finite automata when augmented with various additional mechanisms. It is well known, for example, that NFAs are exponentially more succinct than DFAs, in the following upper and lower bound senses (see [15, 18]): (i) Any NFA can be simulated by a DFA with at most an exponential growth in size; (ii) there is a family of regular sets, L_n, for $n > 0$, such that each L_n is accepted by an NFA of linear size, but the smallest DFA accepting it is at least of size 2^n. By duality, the same is true of the dual machines, sometimes called $\forall$-automata, in which branching is universal. It is also true that AFAs (alternating finite automata), i.e., those that combine *both* types of branching, are exponentially more succinct than either NFAs or $\forall$-automata, and indeed are *double*-exponentially more succinct than DFAs. If, as in [5],

* Corresponding author.
[1] Part of this author's work was carried out during a visit to INRIA, Sophia Antipolis, in the Fall of 1993.

we denote nondeterminism by E and parallelism by A, these results establish that, in the framework of finite automata, E and A are exponentially powerful features, independently of each other (that is, whether or not the other is present), and, moreover, their power is additive: The two combined are double-exponentially more succinct than none.

In [5] a third feature was considered – bounded cooperative concurrency, or simply concurrency for short, denoted by C. The C feature turns out to be very robust, modeling the kind of concurrency present in (bounded, finite-state versions of) languages such as Petri nets, statecharts, CCS, CSP, and many others. It was shown in [5] that C provides a third exponentially powerful feature, which is independent of E and A and additive with respect to them; the savings remain intact in the face of any combination of E and A, and we have, for example, *triple*-exponential upper and lower bounds when transforming an (E,A,C)-machine (i.e., an alternating bounded-token Petri net, or an alternating statechart) into a DFA.

As pointed out by Pratt [17], exponential differences in succinctness give rise to interesting questions about the difficulty of reasoning about regular programs (i.e., regular sets of execution sequences over some alphabet of atomic program letters) that are represented in more succinct ways. The framework in which this issue is addressed is *propositional dynamic logic* (PDL). In the original version of PDL [7] programs are represented by regular expressions. The validity (or satisfiability) problem for this version is known to be logspace-complete for deterministic exponential time [7, 16] (in the sequel, all time complexities are deterministic). The question raised in [17] stems from the exponential gaps in succinctness that exist between automata and regular expressions. For example, it is shown in [6] that NFAs are exponentially more succinct than regular expressions, in the upper and lower bound senses described above.[2] Hence, it is conceivable that PDL in which programs are represented by automata, say, NFAs, instead of regular expressions,[3] requires double-exponential time – one exponential for transforming the NFAs into regular expressions and the other to apply the exponential time decision procedure for the basic version of PDL.

In fact, this is not so: $\mathrm{PDL}_{\mathrm{E}}$, as we may call it, signifying that the programs are automata enriched with the E feature, is also decidable in EXPTIME [11, 17]. Thus, the differences in succinctness between regular expressions and (deterministic or nondeterministic) automata do not affect the exponential time decidability of PDL. Reasoning about abstract regular programs, given in any of these three methods of representation, can be carried out in deterministic exponential time. In contrast, the succinctness provided by the A and C features has been shown in [10] to be stronger, in that the validity problem becomes correspondingly more difficult. Technically, the main result of [10] is that each of A and C adds an exponential to the time complexity of the

[2] For DFAs, there are exponential lower bounds in *both* directions.

[3] Representing regular programs by automata, rather than by regular expressions, is tantamount to moving from textual iterative programs to flowcharts.

decision problem for PDL, and in an additive manner. Thus, for example, the validity problem for $\mathrm{PDL}_{\mathrm{E,A,C}}$ is complete for triple-exponential time.

We may summarize the relevant results of [5, 10] by saying that E, A and C are each exponentially powerful in succinctness, but only for A and C is this power strong enough to affect the difficulty of the reasoning problem too.

The starting point of the present work was to seek additional features of automata that would be powerful enough to yield further exponential amounts of succinctness in the representation of regular sets, and to be also strong enough to raise the complexity of the reasoning problem. We were particularly interested in finding versions of PDL with a decidable (but relatively simple) validity problem that requires more than triple-exponential time.

1.2. Outline

We first consider two-way automata, the "two-wayness" feature being denoted by T. While it is known that T-machines are exponentially more succinct than DFAs [19], we show in Section 2 that in the presence of E, A and C, this additional power disappears: There is a triple-exponential upper bound in transforming (E,A,C,T)-machines into DFAs, just as with (E,A,C)-machines. Thus, the T feature is not what we want.

Our main results concern pebble automata. Section 3 deals with a single pebble and Sections 4 and 5 with multiple pebbles. Now, automata with two pebbles already accept nonregular sets, and we are interested in the *succinctness* of added features, and not on features that provide greater expressive power. Consequently, in Section 4 we restrict multi-pebble automata so that they accept only regular sets. This is done by requiring that the pebbles be manipulated in a stack-like fashion (only the most recently placed pebble can be lifted), and that the automaton behaves at all times like a one-pebble automaton with regard to the most recently placed pebble. Such an automaton can also be considered as a stack of one-pebble automata. Under these conditions, we prove that each pebble adds an exponential amount of power, both to succinctness of the representation and to the difficulty of the reasoning problem.

Definition 1.1. For any constant $c > 1$, we let $c^{[k]}(n)$ denote the k-fold exponential function $c^{c^{\cdot^{\cdot^{\cdot^{n}}}}}$, with k occurrences of c. Also, $exp[k]$ denotes the class consisting of all k-fold exponential functions, for all $c > 1$, applied to some polynomial in the argument. Thus, when we use the term "an $exp[k]$ increase in size", or "an $exp[k]$ blowup", we mean that there is a constant $c > 1$ and a polynomial p, such that the increase is in the order of $c^{[k]}(p(n))$.

The following summarizes our findings regarding deterministic multi-pebble automata:

- Transforming deterministic k-pebble automata into DFAs causes an $exp[k+1]$ increase in size in both the upper and lower bound senses.
- The validity problem for PDL with programs represented by deterministic k-pebble automata is complete for $exp[k+1]$ time.

The latter appears to be the first result that provides a natural hierarchy of logics of programs that are increasingly more difficult to decide, and by an exponential amount of time at each level.

The paper also contains results pertaining to the combination of E, A and C with pebbles. For example, for PDL with programs represented by (E,A,C)-machines with one pebble we have upper and lower bounds of $exp[4]$, and with k pebbles (for $k \geqslant 2$) we have a lower bound of $exp[k+3]$ time, but our best upper bound is $exp[2k+2]$. This leaves a gap of $exp[k-1]$. We feel that the combination of E and A with pebbles is delicate and causes problems that our current techniques cannot cope with. Extra work here is needed.

1.3. Preliminaries

We now provide brief definitions of the various automata used. We first define (E,A,C)-machines as in [5], followed by extensions that admit two-wayness and a pebble, yielding (E,A,C,T)-machines and (E,A,C,P)-machines. We also define the sizes of these automata. Since nondeterminism, pure parallelism, two-wayness and the addition of a pebble are well-known enrichments of automata, the only thing that may require an explanation here is C. The reader who does not want to plow through the following paragraphs may simply think of a C-machine as consisting of a bounded number of communicating DFAs. The way to add E, A, T and P is then quite natural.

The following material, up to Definition 1.4, is adapted from [5].

Definition 1.2. Let Σ be a finite alphabet. Define an (E,A,C)-*machine* to be a tuple $M = (M_1, \ldots M_v, \Phi, \Psi)$, for some $v \geqslant 1$, where each M_i is a triple (Q_i, q_i^0, δ_i). Here, Q_i is a finite set of pairwise-disjoint states, $q_i^0 \in Q_i$ is the initial state, and δ_i, the transition table, is a finite subset of the product $Q_i \times \Sigma \times \Gamma \times Q_i$.[4] We use Γ to denote the collection of propositional formulas over the alphabet of the atomic letters $\bigcup_{1 \leqslant j \leqslant v} Q_j$. Finally, Φ, the E-*condition*, and Ψ, the *termination condition*, are elements of Γ.

The intuition is that M consists of v automata (sometimes called M's *orthogonal components*, or simply *components* for short), each with its own set of states, initial state and transition table. The automata work together in a synchronous manner, taking transitions according to the (common) input symbol being read, their internal states, and the *condition formulas* from Γ. These formulas are interpreted to take on truth values according to the states of possibly *all* the v components. Φ distinguishes between existential and universal state configurations (i.e., between E and A states), and Ψ defines halting configurations.

More formally, a *configuration* of M is an element of $Q_1 \times Q_2 \times \cdots \times Q_v \times \Sigma^* \times \mathcal{N}$, indicating the state each of the M_i is in, the input word and the position of M in the

[4] The definitions could have been given to include ε-moves too, by taking δ_i to be a finite subset of $Q_i \times (\Sigma \cup \varepsilon) \times \Gamma \times Q_i$, and modifying the other parts of the definitions accordingly. Our results all hold for this version too.

word. Thus, $m \leqslant |x|$ for any configuration $(q_1, \dots, q_v, x, m)$. We say that a configuration c *satisfies* a condition $\gamma \in \Gamma$, if γ evaluates to *true* when each symbol therein is assigned *true* iff it appears in c. Thus, e.g., $(q \vee p) \wedge \sim r$, where $q, p \in Q_1$ and $r \in Q_2$, will be satisfied by any configuration for which M_1 is in state q or p, and M_2 is not in state r.

To define the behavior of M, let $x = x_1 x_2 \cdots x_k$ be a word over Σ, and let $t = (q, a, \gamma, p)$ be a transition in M_i's transition table δ_i. We say that t is *applicable to* a configuration $c = (q_1, \dots, q_v, x, j)$, if $x_j = a$, $q_i = q$, and c satisfies γ. A configuration $(p_1, \dots, p_v, x, m)$ is said to be a *successor of* c if for each i there is a transition $(q_i, x_j, \gamma_i, p_i) \in \delta_i$ that is applicable to c, and $m = j + 1$. A configuration is *existential* if it satisfies the E-condition Φ, otherwise it is *universal*. It is *accepting* iff it satisfies the termination condition Ψ.

A *computation* of M on $x \in \Sigma^*$ is defined in a way very similar to that of AFAs (see, e.g., [4]). It consists of a tree, each node of which is labeled with a configuration. The root is labeled with the *initial* configuration $(q_1^0, q_2^0, \dots, q_v^0, x, 1)$, and a node has one successor node for each of its lable's successor configurations, labeled with that successor configuration. Nodes are assigned 1/0 (accept/reject) marks, in a bottom up manner, as in the definition for AFAs, "or"ing the marks of the successors of an existential node and "and"ing those of a universal node. The input word x is accepted iff the root gets marked with 1.

Definition 1.3. The *size* of the machine $M = (M_1 \dots M_v, \Phi, \Psi)$ is defined to be $|M| = |\Phi| + |\Psi| + \sum_{i=1}^{v} |M_i|$, where the size of a formula in Γ is its length in symbols, and the size of each component automaton is defined by $|M_i| = |Q_i| + \sum_{(q,a,\gamma,p)\in\delta_i} (3 + |\gamma|)$.

Note that if $v = 1$, the machine M is simply an AFA (in our terminology, it is an (E,A)-machine); if, in addition, Φ is a tautology, then all states are existential, so that M is an NFA (an E-machine); if Φ is inconsistent, then all states are universal, so that M is an $\forall$-automaton (an A-machine); if δ does not contain two transitions emanating from the same state and labeled with the same symbol, then M is a DFA (an $\emptyset$-machine).

Definition 1.4. An (E,A,C,T)-machine is an (E,A,C)-machine extended by being allowed to move in both directions. Input words are assumed to have end-markers. The transition tables of the M_i include the direction of the move, so that δ_i is a finite subset of $Q_i \times \Sigma \times \Gamma \times Q_i \times \{L, R\}$. A transition is applicable only if in all components it prescribes movement in the same direction.

Definition 1.5. An (E,A,C,P)-machine is an (E,A,C,T)-machine extended with a pebble. The transition tables of the M_i include dependence on the presence or absence of a pebble on the tape cell scanned, and can prescribe placing or picking up the pebble. The details of this are straightforward, and are omitted. A transition is applicable only if in all components it prescribes the same pebble action and movement in the same direction.

The size of an (E,A,C,T)-machine and of an (E,A,C,P)-machine is defined similarly to that of an (E,A,C)-machine.

We now define the exponential and multi-exponential gaps we are interested in establishing between the various kinds of machines.

Definition 1.6. Let ξ_1 and ξ_2 be any two subsets of $\{E,A,C,T\}$ or $\{E,A,C,P\}$. We write $\xi_1 \to \xi_2$ (respectively, $\xi_1 \xrightarrow{k} \xi_2$), if any ξ_1-machine can be transformed into an equivalent ξ_2-machine with at most a polynomial (respectively, an *exp*$[k]$) increase in size. When the arrows in these notations are superscripted by an r, as in $\xi_1 \to^r \xi_2$, the intention is that the claimed-to-exist ξ_2-machine accepts the *reverse* of the language accepted by the ξ_1-machine, rather than that language itself, i.e., the language containing the words in reverse.

Definition 1.7. Let ξ_1 and ξ_2 be any two subsets of $\{E,A,C,T\}$ or $\{E,A,C,P\}$. We write $\xi_1 \underset{k}{\to} \xi_2$, if there is a family of regular languages L_n, for $n > 0$, and a monotonically increasing function f and a function $g \in exp[k]$, such that L_n is accepted by a ξ_1-machine of size $f(n)$, but the smallest ξ_2-machine accepting it is at least of size $g(f(n))$.

2. The power of two-wayness

In this section we exhibit upper and lower bounds on the relative succinctness of features E, A, C and T for finite automata. The following table summarizes them. Each entry represents the transition from an automaton of the type indicated in the row to an automaton of the type indicated in the column. An entry n in the table represents upper and lower bounds of *exp*$[n]$, a 0 represents a polynomial bound, a dash means that the question is trivial (usually the row type is a special case of the column type), a question mark represents a trivial upper bound with an unknown nontrivial lower bound.

Table 1 summarizes known results for transformations involving E, A and C, taken from [5], known results involving E, A and T, some of which appear in [14, 19, 20], new results involving E, A and T (A,T $\to$ E,A; E,T $\to$ E,A; A,T $\to$ E,A), and new results involving both C and T, with or without E and A. New results are boldfaced in the table.

Sheperdson [19] proved that two-way automata accept only regular languages. Indeed, his proof shows also that the transformation to DFAs can be carried out with at most an exponential blowup. In our terminology, he thus shows T $\xrightarrow{1}$ $\emptyset$. We reproduce this proof, since our proof of T $\to$ C (Proposition 2.5) draws upon it.

Proposition 2.1 (Sheperdson [19]). *$T \xrightarrow{1} \emptyset$.*

Proof. Let M be a T-machine with n states, $Q = \{q_1, \dots, q_n\}$, where q_1 is the initial state. We exhibit a DFA M' with $(n+1)^{(n+1)}$ states, such that $L(M) = L(M')$. A

Table 1

								E				E		E	A
					E	E	A	A		E	A	A	C	C	C
	∅	E	A	C	A	C	C	C	T	T	T	T	T	T	T
E	1	–	1	0	–	–	0	–	?	–	?	–	**0**	–	**0**
A	1	1	–	0	–	0	–	–	?	?	–	–	**0**	**0**	
C	1	1	1	–	1	–	–	–	**1**	**1**	**1**	?	–	–	–
EA	2	1	1	1	–	0	0	–	1	1	1	–	**0**	**0**	**0**
EC	2	1	2	1	1	–	1	–	?	**1**	**1**	?	?	–	?
AC	2	2	1	1	1	1	–	–	?	**1**	**1**	?	?	?	
EAC	3	2	2	2	1	1	1	–	**2**	**2**	**2**	**1**	**1**	**1**	1
T	1	1	1	**0**	**0**	0	0	0	–	–	–	–	–	–	–
ET	1	1	1	**0**	**0**	**0**	**0**	**0**	?	–	?	–	**0**	–	**0**
AT	1	1	1	**0**	**0**	**0**	**0**	**0**	?	?	–	–	**0**	**0**	–
EAT	2	1	1	**1**	1	?	?	?	?	1	1	–	?	?	?
CT	**2**	**2**	**2**	**1**	**1**	**1**	**1**	?	**1**	**1**	**1**	?	–	–	–
ECT	**2**	**2**	**2**	**1**	**1**	**1**	**1**	?	?	**1**	?	?	?	–	?
ACT	**2**	**2**	**2**	**1**	**1**	**1**	**1**	?	?	?	**1**	?	?	?	–
EACT	**3**	**2**	**2**	**2**	**2**	?	?	?	?	**2**	**2**	**1**	?	?	?

state of M' is an element of $(Q \cup \{0\})^{(n+1)}$. M' will be in state $(q_{i_0}, q_{i_1}, \ldots, q_{i_n})$ during the computation on some input word w before scanning some letter a, if during its computation on w the original machine M arrives at a for the first time in state q_{i_0}, and for each $1 \leqslant j \leqslant n$, if M scans the letter to the left of a in state q_j, it returns from left to right in q_{i_j} in order to re-scan a. If this return never happens, i.e., M gets stuck or never returns to the a, q_{i_j} is 0.

The initial state of M' is $(q_1, 0, \ldots, 0)$. Its accepting states are the vectors containing an accepting state of M in the first position. The transitions of M' are defined such that the required meaning of the states is preserved: From a state $u = (q_{i_0}, q_{i_1}, \ldots, q_{i_n})$ and a letter a, a transition to v is defined as follows. If M has a right-transition (q_{i_0}, a, q_k, R), then the first element of v will be q_k. If M has a left-transition (q_{i_0}, a, q_k, L), then we check if M has a transition (q_{i_k}, a, q_l, R), since, according to u, if M turns left with q_k it returns with q_{i_k} in order to scan a. If there is such a transition, the first element of v will be q_l. Otherwise, if M contains (q_{i_k}, a, q_l, L), then we check the same repeatedly (at most n times), until M contains an appropriate right-transition. If M does not contain such a right-transition, or if we find 0 in some position that we check in u, the first element of v will be 0. The other elements of v are defined similarly. It follows from the construction that $L(M) = L(M')$ and that M' is of size $exp[1](n)$ ($n^n = 2^{n \log n}$). □

Strengthening the techniques of [19], it is possible to show that the same holds in the presence of E or A.

Proposition 2.2. $(E, T) \xrightarrow{1} \emptyset$; $(A, T) \xrightarrow{1} \emptyset$.

Proof. The construction is similar to the previous one, except for the contents of the vectors. Here, each position contains a subset of the states of M instead of a single state, meaning that the E-machine or A-machine M can arrive at each one of them by some computation path. The meanings of the vector states are the same as in the previous proof. The initial state, accepting states and transitions are defined accordingly. The number of states is now at most $(2^n)^{n+1}$, so that the machine is of size $exp[1](n)$. □

If both E and A are present, the removal costs two exponentials.

Proposition 2.3 (Ladner et al. [14]). $(E,A,T) \xrightarrow{2} \emptyset$.

This proposition follows immediately from the next but for the sake of clarity of the next proof we bring the proof here.

Proof. Let M be an (E, A, T)-machine with n states. We construct a DFA M' similarly to the previous constructions. This time, each element of the vectors will be a formula in CNF, employing the $\vee, \wedge$ connectives. Atoms are the states of M and 0. Each formula represents the "status" of states in which M may be, according to the computation paths of M on the input word, in the sense of the previous proofs. We omit the details.

The initial state is $(q_1, 0, \ldots, 0)$. The accepting states are all the vectors for which the first position contains a formula that becomes true when substituting each accepting state by TRUE and the other atoms by FALSE. The transitions are defined such that the meaning of the states is preserved. From a state $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ and an input letter a, M' moves to state $(\beta_0, \beta_1, \ldots, \beta_n)$, where β_0 is obtained from α_0 by exchanging each state q appearing in α_0 with the formula representing the "status" of states that M would reach from q and renormalize to CNF.

Since a CNF formula is a set of sets, we have $2^{2^{(n+1)}}$ different formulas, and hence there are $(2^{2^{(n+1)}})^{(n+1)}$ different vectors of size $n+1$. Thus, the size of M' is $exp[2](n)$. □

Proposition 2.4. $(E,A,T) \xrightarrow{1} A$; $(E,A,T) \xrightarrow{1} E$.

Proof. Let M be an (E, A, T)-machine with n states, $Q = \{q_1, \ldots, q_n\}$, and let M' be the equivalent DFA constructed in the previous proof. We construct an A-machine M'', similarly to the previous constructions. As in the proof of Proposition 3.3, the states will be vectors of subsets of states of M. These subsets will replace the E feature of M. The transitions of M'' are defined using M', as follows. Let $\widehat{Q} = (Q_0, \ldots, Q_n)$, for $Q_i \subseteq Q$, be some state of M'', and let a be an input letter. Among the states of the DFA M' there is a state $\widehat{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_n)$, such that for each $0 \leqslant i \leqslant n$, α_i is a disjunction of the elements in Q_i. Assume that, when seeing a, M' moves from $\widehat{\alpha}$ to $\widehat{\beta} = (\beta_0, \beta_1, \ldots, \beta_n)$. Recall that each $\beta_i = \gamma_{i_1} \wedge \cdots \wedge \gamma_{i_{k_i}}$ is a formula in CNF. We now define transitions from $\widehat{Q}$, when seeing a, to all the vectors $(P_0, \ldots, P_n)$, where each P_i is the set of states in γ_{i_j}, for some $1 \leqslant j \leqslant k_i$.

The proof for E is dual. □

In contrast to the $\mathrm{T} \xrightarrow{1} \emptyset$ bound, T can be replaced by C or by alternation without an exponential blowup. First, the case of C:

Proposition 2.5. $T \to C$.

Proof. Let M be a T-machine with n states, $q_1, \ldots, q_n$. We exhibit a C-machine M' with $O(n^3)$ states and $O(n^4)$ transitions, that mimics the simulation process of M by a DFA. Let us denote by M'' the DFA constructed in the proof of Proposition 2.1. M' will consist of a control component and two groups of $n+1$ components, denoted A and B (see Fig. 1). Each group represents a vector of $n+1$ states of M (or 0), with the same meaning as in the previous proofs. A encodes the current state, and B computes the next state of M''. The new state is then copied from B to A, and the process is reiterated.

A typical subcomponent of B, say B_i, is used to compute the state q to which M returns when it moves left in state q_i. B_i contains an initial state, as well as n states that are displayed along the last line of B_i in the figure and are denoted by $l_1, \ldots, l_n$. Their role is such, that when q has been computed M' will be in the bottom state that represents q. To carry out the computation, B_i contains an additional $n \times n$ states. In the simulation of M by the DFA M'', q is determined by repeatedly checking the state in which M is when it returns for the first time since last moving left. This needs to be done at most n times, and is simulated by the $n \times n$ states in B_i. When there is a left-transition in M, M' passes to the next column according to the information in

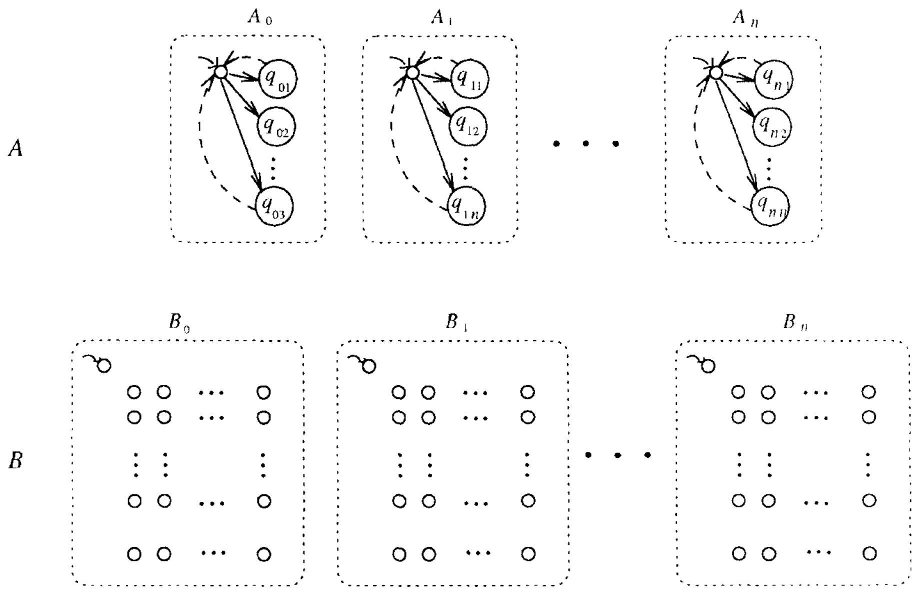

Fig. 1. The general structure of M' in the proof of Proposition 2.5.

the previous state; i.e., according to the appropriate component of A in which M' is at present. When there is a right-transition, M' enters the corresponding state in the botton row.

Here, for example, is how we would construct B_1 (see Fig. 2): If M has a left-transition from q_1 into q_3, then M' has to check in which state M would return if it is in state q_3 and it turns left. Therefore, we define a transition from the initial state into p_{i1}, for $1 \leqslant i \leqslant n$, in the first column, with the condition "in q_{3i}" (q_{3i} is the ith state from the top in A_3). Similarly, we define the same transition from each p_{1i}, $1 \leqslant i \leqslant n-1$, into each $p_{k(i+1)}$, $1 \leqslant k \leqslant n$. If M has an additional left-transition, for example from q_3 to q_6, we define the following transitions in B_1: From each p_{3i}, $1 \leqslant i \leqslant n-1$, into each of the $p_{k(i+1)}$, $1 \leqslant k \leqslant n$, with the condition "in q_{6k}". Now, if there is a right-transition in M, e.g., from q_2 into q_4, then we define transitions from all p_{2k}, for $1 \leqslant k \leqslant n$, into l_4 in the bottom row.

Verifying that this construction does the job is tedious, but straightforward. □

This proof can be extended to work for (E, T)-machines, similarly to the extension of Shepherdson's original proof for the case (E, T) $\xrightarrow{1}$ $\emptyset$, by using subsets of states as elements of the vectors:

Proposition 2.6. $(E,T) \rightarrow C$.

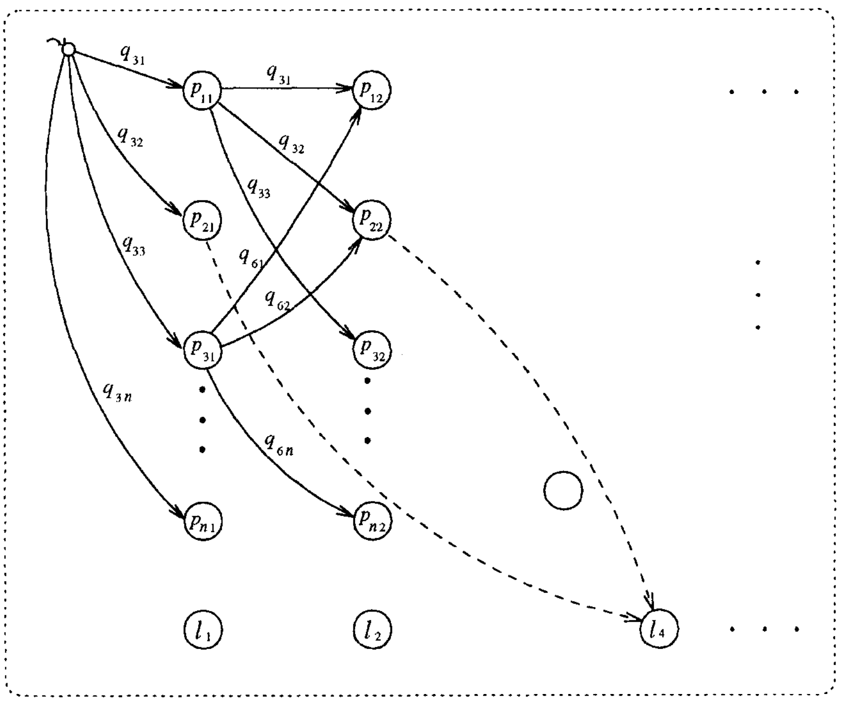

Fig. 2. The inners of B_1 from Fig. 1.

Sketch of Proof. The general structure of the C-machine is similar to the C-machine in the former proof. The only difference is that each location in the vector will represent a subset of n states, by having by n components of two states each that denote the presence or absence of the appropriate state in the subset. □

From Proposition 2.4 and the $E \rightarrow C$ result of [5], we obtain:

Proposition 2.7. $(E,A,T) \xrightarrow{1} C$.

Birget proves in [2] that any T-machine can be simulated by an AFA with a polynomial growth in the number of states. This AFA is of the richer kind, possibly containing arbitrary Boolean functions of the states on the transitions. In terms of machine size, the resulting AFA is *exponentially* larger than the T-machine, since the proof in [2] uses the simulation of a DFA by an AFA via the reverse language, a process that decreases the number of states only, but carries the size in the formulas along the transitions. However, using a C-machine along the way, we can, in fact, show T $\rightarrow$ (E, A), and even the following stronger result:

Proposition 2.8. $(E,T) \rightarrow (E,A)$.

Proof. We have (E, T) $\rightarrow^r$ (E, T) $\rightarrow$ C. We now use the C $\rightarrow^r$ (E, A) bound from [12]. □

Proposition 2.9. $(E,A) \rightarrow (C,T)$.

Proof. Again, from [12] we obtain (E, A) $\rightarrow^r$ C, and then use the trivial C $\rightarrow^r$ (C, T). □

Proposition 2.10 (cf. Drusinsky and Harel [5]). *For any* $\xi \subseteq \{E,A,T\}$, $(C,\xi) \xrightarrow{1} \xi$.

Proof. As in [5], simply simulate the behavior of a (C,ξ)-machine M by a ξ-machine whose set of states is the Cartesian product of the states in M's component machines. □

The rest of the upper bounds appearing in the table follow easily from the known ones. For example, that (E, A, T) $\xrightarrow{1}$ C holds, follows from (E, A, T) $\xrightarrow{1}$ E $\rightarrow$ C; that (E, A, C) $\xrightarrow{1}$ (C, T) holds, follows from (E, A, C) $\xrightarrow{1}$ (E, A) $\rightarrow$ (C, T); that (C, T) $\xrightarrow{1}$ C holds, follows from (C, T) $\xrightarrow{1}$ T $\rightarrow$ C; and that (E, A, C, T) $\xrightarrow{2}$ C holds, follows from (E, A, C, T) $\xrightarrow{1}$ (E, A, T) $\xrightarrow{1}$ C.

All the lower bounds can be proved using the sample languages appearing in [5, 12]:

$$L_n = \{¢ww¢ \mid w \in \{0,1\}^n\}$$

$$K_n = \{¢w_1w_2¢ \mid w_1, w_2 \in \{0,1\}^n, w_1 \neq w_2\}$$

$$V_n = \{¢\, w\# w_1\$w_2 \ldots \$w_m¢ \mid w_1,\ldots,w_m, w \in \{0,1\}^n, \text{ and for some } i,\ w_i = w\}$$
$$U_n = \{¢\, w_1\$w_2 \ldots \$w_m\# w¢ \mid w_1,\ldots,w_m, w \in \{0,1\}^n, \text{ and for some } i,\ w_i = w\}$$

For example, $\mathrm{T} \underset{1}{\rightarrow} \mathrm{E}$ and $(\mathrm{T},\mathrm{C}) \underset{2}{\rightarrow} \mathrm{E}$ can be proved using L_n, as follows. There is a (T, C)-machine M of size $\mathrm{O}(\log n)$ that computes L_n. M uses two counting components (see [5]), one to count the first n bits, and for each such bit the second is used to reach the corresponding bit in order to verify equality. In contrast, the smallest E-machine for L_n must contain at least 2^n states.

$\mathrm{T} \underset{1}{\rightarrow} \mathrm{A}$ and $(\mathrm{T},\mathrm{C}) \underset{2}{\rightarrow} \mathrm{A}$ can be proved similarly, using K_n; $(\mathrm{E},\mathrm{A},\mathrm{T}) \underset{1}{\rightarrow} (\mathrm{E},\mathrm{A})$ and $\mathrm{C} \underset{1}{\rightarrow} (\mathrm{E},\mathrm{T})$ can be proved using V_n; and $(\mathrm{E},\mathrm{A}) \underset{1}{\rightarrow} \mathrm{T}$ can be proved using U_n.

3. A single pebble

Blum and Hewitt [3] (see also [13]) proved that one-pebble automata accept only regular languages. Indeed, their proof shows also that the transformation to DFAs can be carried out with at most a double-exponential blowup, i.e., they really prove $\mathrm{P} \xrightarrow{2} \emptyset$. Strengthening their techniques, we can prove the following result, and a matching lower bound appears in Theorem 4.7:

Theorem 3.1. $P \xrightarrow{1} E$.

(Note: This result, and that of Proposition 3.2, were proved independently in [2], based upon a claim appearing in [1].)

Proof. Let $M = (Q, \Sigma, q_1, \delta, F)$ be a P-machine with $Q = \{q_1, \ldots, q_n\}$. We first build a T-machine M' that operates on words over a new alphabet:

$$\Sigma' = \{(a, p_1, \ldots, p_n) \mid a \in \Sigma, \quad \text{and for all } i,\ 1 \leqslant i \leqslant n,\ p_i \in Q \cup \{0\}\}.$$

With each input word $x \in \Sigma^*$ we associate a new word $x' \in (\Sigma')^*$, obtained by replacing each letter a in x by the vector $(a, p_1, \ldots, p_n)$ that describes the following behavior of M: If M scans the present letter with q_i, keeps going and then returns to the same position without using the pebble in the interim, we let p_i be the state M is in when it thus returns. If this does not happen, we let $p_i = 0$.

The states of M' are those of M, and the transitions of M' are defined such that M' simulates M without using a pebble. All transitions in M that do not include picking up or placing the pebble will be in M', with the appropriate changes in the input symbols. For transitions in M in which M places the pebble, we define a transition to the state that M enters upon picking up the pebble for the first time thereafter. This state can be determined by the information in the letter and the transition of M. This part of the construction implies that if $x \in L(M)$ then $x' \in L(M')$.

In order to satisfy the other direction, i.e., that if any $y \in (\Sigma')^*$ is accepted by M' then the word obtained from y by projecting out the states is accepted by M, the automaton has to make sure that the vectors replacing the letters of the input word x are consistent with the operation of M on x. Therefore, we build another T-machine, M'', whose role is to check the mutual consistency of the vectors in the input word. M'' scans the input word twice – first from left to right and then from right to left. During the first scan it considers each state for which M contains a left-transition, verifying that the appropriate position in the vector contains the state in which M returns. Similarly, in the second scan it checks the same for those states for which M contains a right-transition. This check seems to require exponentially many states, but with some careful programming M'' can be made to have only $O(n^3)$ states. Here is how. It checks the vector element by element; to check some element, it employs n^2+n states and uses the information on the input letter. (Part of this idea is similar to that used to construct B in the proof of Proposition 2.5.) Note that when checking the vector replacing some input letter, M'' has already checked the vector appearing in front of that letter. Hence, it can assume that the information there is consistent with the operation of M on x. Thus, the T-machine obtained by concatenating M'' and M', denoted M''', contains $O(n^3)$ states and $exp[1](n)$ transitions. (Note that going from M' to M''' does not cause the first direction to be violated.)

We now simulate M''' by a DFA U, which causes an exponential blowup in the number of states only. Hence, the total size of U is exponential in n. Finally, we simply replace each letter $(a, p_1, \ldots, p_n)$ by a, thus causing U to become an NFA V of the same size as U. Moreover, $x \in L(M)$ iff $x' \in L(M''')$ iff $x' \in L(U)$ iff $x \in L(V)$. □

The presence of nondeterminism does not make things worse:

Proposition 3.2. $(E,P) \xrightarrow{1} E$.

Proof. Let M be an (E, P)-machine with n states. The simulation is similar to the one in Proposition 3.1, except for the following. Here we simulate M by an (E, T)-machine (not a T-machine) with polynomially many states and exponentially many transitions, using a new alphabet that consists of vectors containing subsets of states of M. We then simulate this machine by a DFA with exponentially many states, replace the vector-symbols on the transitions by the original symbols, obtaining an E-machine of exponential size. □

For alternation, however, we have the following, with Theorem 4.13 providing the matching lower bound.

Proposition 3.3 (Goralcik et al. [8]). $(E,A,P) \xrightarrow{2} E$.

Proof. Let M be an (E, A, P)-machine with n states. Here, we use a new alphabet that consists of vectors containing formulas in CNF as in the proof of Proposition 2.3. We first simulate M by an (E, T)-machine with $exp[1](n)$ states and $exp[2](n)$ transitions, and then simulate it by a DFA with $exp[2](n)$ states, and replace the symbols as before. □

4. Multiple pebbles

In our current work, we are interested in the added succinctness of mechanisms for accepting regular sets. However, pebble automata with more than one pebble can accept nonregular languages too, leading out of our realm of interest. For example, a 2-pebble automaton exists for $\{¢w\$w¢ \mid w \in \{0,1\}^*\}$. Hence, we first restrict the behavior of multi-pebble automata, and show that the restricted machines accept only the regular sets.

Definition 4.1. A *kP-machine*, for $k \geqslant 1$, is a two-way automaton with k pebbles, $P_1, \dots, P_k$, that adheres to the following restrictions:

(1) P_{i+1} may not be placed unless P_i is already on the tape, and P_i may not be picked up unless P_{i+1} is not on the tape. (Thus, pebbles are placed and picked up in an LIFO style.)

(2) Between the time P_{i+1} is placed and the time that either P_i is picked up or P_{i+2} is placed, the automaton can traverse only the subword located between the current location of P_i and the end of the input word that lies in the direction of P_{i+1}. Moreover, in this subword, the automaton can act only as a 1-pebble automaton with pebble P_{i+1}. In particular, it is not allowed to lift up, place, or even sense the presence of any other pebble.

Intuitively, a kP-machine can be viewed as a stack of (at most) k 1-pebble automata. At any given moment in time, only the automaton at the top of the stack gets to work on the input word. An automaton is put on the stack (and is thus set to work) when a pebble is placed. The automaton is popped, leaving the next automaton on the stack to resume work, when a pebble is picked up. In addition, each automaton operates only on the subword located between the previous automaton's pebble and the appropriate end of the word.

Theorem 4.2. *For any $k \geqslant 1$, kP $\xrightarrow{k}$ E. Thus, kP-machines accept the regular sets, and kP $\xrightarrow{k+1}$ $\emptyset$.*

Proof. We actually prove a stronger result, by induction on k. We let kP be partially nondeterministic, being allowed to act nondeterministically as long as all the pebbles are placed.

For $k = 1$, (E, P) $\xrightarrow{1}$ E by Proposition 3.2. Assume that $k > 0$ and kP $\xrightarrow{k}$ E, and let M be a $(k+1)$P-machine of size n. Without loss of generality, assumes that an input word is accepted if M reaches an accepting state and no pebble is located on the word. We imagine the simulation to be using two copies of M. The first is used until P_k is placed, at which time it passes control to the second copy, which operates as a 1-pebble automaton with pebble P_{k+1}. When P_k is to be picked up, control is returned to the first copy. We simulate the second copy, which is actually a 1-pebble automaton, by an E-machine, and then use the inductive hypothesis to complete the simulation.

More specifically, we first build two copies of M, denoted M' and M'' (see Fig. 3). Each state in M'' that M enters after placing P_k is called an "entering state". Each state of M'' in which P_k is picked up is called an "exiting state". All transitions in M'' that involve a pebble from among $P_1, \dots, P_k$ are eliminated, as are all transitions in M' that involve P_{k+1}. Now, instead of just having one copy of M'', we do the following: For each pair (q, p), where q is an entering state and p is an exiting state, we have a separate copy of M'', denoted M''_{qp}. For each transition from r into a q in which P_k is placed, we define a transition from the copy of r in M' to all the entering states q in the copies M''_{qp}. For each transition from p to s in which P_k is picked up, we define

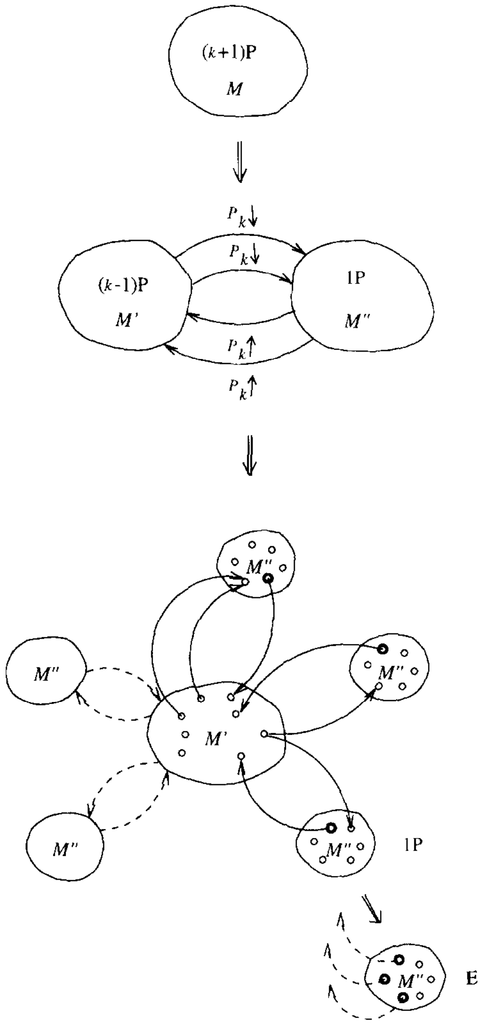

Fig. 3. Illustration for the proof of Theorem 4.2 (downarrow = placing pebble; uparrow = picking up pebble).

transitions from all exiting states p of the copies M''_{qp} to the s of M'. The fact that the compound machine is equivalent in its behavior to the original M follows directly from the limitations on kP-machines.

We are not done yet, since this machine still uses $k+1$ pebbles. Each copy M''_{qp} is now viewed as a 1-pebble automaton, with initial state q and accepting state p, and working on the subword between the pebble P_k on one side of the head and the endmark of the original input word on the other. We say that this subword is accepted if M''_{qp} reaches the P_k end of it in an accepting state. Each such M''_{qp} is now simulated by an E-machine (with no pebble) by Proposition 3.1, causing an exponential growth in size. To the resulting machine we add a portion, which, upon acceptance of the subword, takes the head directly back to P_k's location. Finally, we connect M' to the entering states of the simulated copies M''_{qp}. In this way, we obtain an automaton with k pebbles that is exponentially larger than M. (Note that this connection of M', and the translation of the M''_{qp} into E-machines, can cause the automaton to become nondeterministic.) By the inductive hypothesis, we can simulate the resulting automaton by an E-machine with an additional $exp[k]$ increase in size. □

The previous proof can be seen to work for nondeterministic kP-machines too:

Proposition 4.3. *For any $k \geqslant 1$, $(E, kP) \xrightarrow{k} E$.*

We now prepare for the matching lower bound. Extending the ideas appearing in [10], we define k-level index words (see Fig. 4):

Definition 4.4. Let n be fixed. A *word of degree k over n, called a k-word,* is defined inductively as follows. A 0-word is any word in $\{0,1\}^n$. A $(k+1)$-word is a word w over $\{0,1,\$_1,\ldots,\$_{k+1}\}$ of the form $w = w_0 b_0 \$_{k+1} \cdots w_m b_m \$_{k+1}$, where $m = 2^{[k+1]}(n) - 1$, and for all i, $0 \leqslant i \leqslant m$, $b_i \in \{0,1\}$ and w_i is a k-word whose derivative is the number i written in binary. The *derivative* of a k-word w, denoted by w', is defined as follows. For a 0-word w, take $w' = w$, and for a k-word $w = w_0 b_0 \$_k \cdots w_m b_m \$_k$, take $w' = b_0 \cdots b_m$. The $(k-1)$-words that appear inside a k-word are called its *indices.*

Lemma 4.5. *There is a kP-machine of size* O(n), *which, given an index x followed by a k-word y (respectively, preceeding a k-word y), finds the location of x in y. More precisely, for $k \geqslant 1$, let $w = w_1\$xaw_2\$y\$w_3$ (respectively, $w = w_1\$y\$w_2\$xaw_3$),*

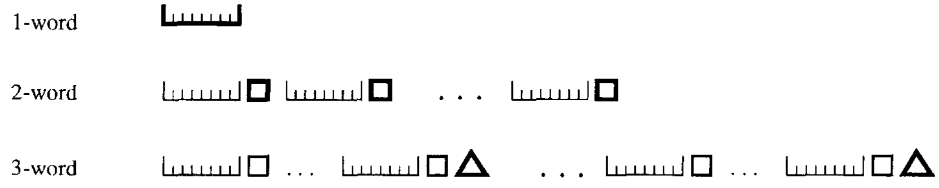

Fig. 4. Schematic form of index-words.

where $w_1, w_2, w_3, x, y \in \{0, 1, \$_1, \ldots, \$_k\}^*$, x is a $(k-1)$-word, $a \in \{0, 1\}$, and $y = y_0b_0\$_k \cdots y_mb_m\$_k$ is a k-word. Then there is a kP-machine of size $O(n)$ that identifies the y_i that is identical to x.

Proof. First note that there must be a y_i as in the lemma, since y is a k-word and x is a $(k-1)$-word. We now prove the claim by induction on k. We deal with the case where x is followed by the k-word; the dual case is proved similarly. For $k = 1$, let $w = w_1\$xaw_2\$y\$w_3$, where $x \in \{0, 1\}^n$, $y = y_0b_0\$_1 \cdots y_mb_m\$_1$, $y_i \in \{0, 1\}^n$, and $b_i \in \{0, 1\}$. The P-machine M that finds the y_i that is identical to x places the pebble in front of each y_i in turn, comparing it with x, until it finds the identical one. Since $|x| = n$, the comparison can be done by running forward n bits and backwards $n-1$ bits, comparing the appropriate pairs. The size of M is $O(n)$, since we must count to n.

Let $k \geqslant 1$, and assume there is a kP-machine M' that finds a $(k-1)$-word within a k-word. Let $w = w_1\$xaw_2\$y\$w_3$, where x is a k-word and y is a $(k+1)$-word. We construct a $(k+1)$P-machine M that compares x to each y_i, $i = 1, 2, \ldots, m$, in order. Recall that y_i is of the form $z_1c_1\$_k \cdots z_lc_l\$_k$, where each index z_j is a $(k-1)$-word and $c_j \in \{0, 1\}$. M runs through the z_j in turn doing the following. It places the first pebble in front of the current z_j, and uses the assumed-to-exist kP-machine M' to find z_j in x. It then compares their next bits. If they are identical, then if all indices of y_i have been checked it stops, and if there are more indices in y_i, it moves the first pebble to the next such index, i.e., to z_{j+1}, and keeps going. If the bits are not identical, M moves the first pebble to y_{i+1}, and tries again there. Before moving the first pebble, M picks up all the other pebbles. Clearly, M is a $(k+1)$P-machine, and is of linear size. □

Lemma 4.6. *Let $k \geqslant 0$, and let $w = x\$y\$z$ for some symbol $\$ \notin xyz$. Then there is a kP-machine of size $O(n)$ that checks whether y is a k-word.*

Proof. By induction on k. Clearly, there is a DFA of linear size that accepts 0-words, by counting up to n. Assume that the claim holds for $k \geqslant 0$. Let $w = x\$y\$z$, with $\$ \notin xyz$. We exhibit a $(k+1)$P-machine M that checks if y is a $(k+1)$-word. By the inductive hypothesis, we can use k pebbles from M to check that every index in y is a k-word. Now, M picks up all the pebbles, and verifies that the derivative of the first index is a sequence of 0s and the derivative of the last index is a sequence of 1s. It now has only to check that the values of the derivatives of consecutive indices are consecutive natural numbers. This is done by finding the first nonidentical pair of corresponding bits in the two indices, and verifying that all bits following the first of these are 1 and the ones following the second are all 0. Recall that there is a $(k-1)$-word prefixing each bit, which serves as an internal index. Comparing corresponding bits is done by moving the first pebble from internal index to internal index. After marking some internal index, M finds it in the next index using k pebbles by Lemma 4.5, and compares the appropriate bits. Thus, M uses a total of $k+1$ pebbles, and is of linear size. □

Now for the lower bound:

Theorem 4.7. *For each $k \geqslant 1$, $kP \underset{k+1}{\longrightarrow} \emptyset$, and $kP \underset{k}{\rightarrow} E$.*

Proof. Let $k \geqslant 1$, and define $L_n = \{\$w_1\$w_2 \cdots \$w_m\#w \mid m \geqslant 1, w_1, \ldots, w_m, w$ are all $(k-1)$-words, and for some i, $w_i = w\}$. We construct a kP-machine M of size $O(n)$ that accepts L_n. M first checks that the input word is a sequence of $(k-1)$-words, using $k-1$ pebbles as in Lemma 4.6. Now, M scans the w_i's in order, using the first pebble, and compares them to w. If $k = 1$, then w and all the w_i are 0-words, and the comparisons can be made by counting to n forwards and backwards. If $k > 1$, then after placing the first pebble in front of some w_i, M proceeds to move it along the indices in w_i. Each index x, which is really a $(k-2)$-word, is found using $k-1$ pebbles by Lemma 4.5, and M then compares the subsequent bit. Thus, M uses a total of k pebbles, and its size is $O(n)$.[5]

To complete the proof, note that the smallest DFA accepting L_n must contain at least $2^{[k+1]}(n)$ states, since there are $2^{[k]}(n)$ different $(k-1)$-words and it has to be able to distinguish all $2^{[k+1]}(n)$ different subsets thereof. An E-machine, therefore, must have at least $2^{[k]}$ states. □

We now prepare for analogous results on alternating k-pebble automata. The best upper bound we were able to obtain for the difficult combination of E and A with pebbles involves a restricted kind of multi-pebble automaton. The lower bound in Theorem 4.13 is way below it for more than one pebble, so that more work seems to be needed here.

Definition 4.8. A *limited-*(E, A, kP)*-machine*, or simply an *l-*(E, A, kP)*-machine* for short, is an (E, A, kP)-machine that never picks up pebbles.

Theorem 4.9. *For any $k \geqslant 1$, l-(E,A,kP) $\overset{2k}{\rightarrow}$ E.*

Proof. By induction on k. For $k = 1$, (E, A, P) $\overset{2}{\rightarrow}$ E, by Proposition 3.3. For $k \geqslant 1$, assume that l-(E, A, kP) $\overset{2k}{\rightarrow}$ E, and let M be an l-(E, A, $(k+1)$P)-machine of size n. We construct an automaton similar to the one constructed in the proof of Theorem 4.2. This time, the number of copies of M'' is the same as the number of entering states, and the accepting states of the copies of M'' are the accepting states of M, instead of its exiting states. We connect M' to the entering states of the M'' by all transitions that involve placing P_k. It is not necessary to connect the copies of M'' back to M', because no pebbles are picked up. Each copy of M'', which can be seen to be an

[5] Note that in this construction the first pebble is used both to pass from k-word to k-word and to pass from index to index within the k-word being checked. Hence, we couldn't have naively used the fact that k pebbles can check the equality of two marked $(k-1)$-words within a given word (see Lemma 5.1).

(E, A, P)-machine, is now simulated by an E-machine with a double-exponential increase in size. The resulting automaton is thus an l-(E, A, kP)-machine of size $exp[2](n)$, and is equivalent to M. By the inductive hypothesis we can simulate this machine by an E-machine with an additional $exp[2k]$ increase in size. □

To prepare for the lower bound, we prove claims similar to those in Lemmas 4.5 and 4.6. Instead of kP-machines, they involve l-(E, A, $(k-1)$P)-machines.

Lemma 4.10. *Let $k \geqslant 1$, and let w be as described in Lemma 4.5. Then there is an l-(E, A, $(k-1)$P)-machine of size* O(n) *that checks if there is an i for which $xa = y_i b_i$.*

Proof. By induction on k. For $k = 1$, let $w = w_1\$xaw_2\$yw_3$, where $x \in \{0,1\}^n$ is a 0-word, $y = y_0 b_0 \$_1 \cdots y_m b_m \$_1$, $y_i \in \{0,1\}^n$, $b_i \in \{0,1\}$, and y is a 1-word. We construct an (E, A, T)-machine M that verifies that xa is identical to some $y_i b_i$. First, M uses nondeterminism (i.e., E) to guess that appropriate $y_i b_i$. It then uses A to compare the $n+1$ bits in parallel, by moving to the left and counting up to n to find the right place in xa. The counting causes the size to be O(n).

For $k > 1$, assume there is an (E, A, $(k-1)$P)-machine for checking if some $(k-1)$-word appears within a k-word. Let $w = w_1\$xaw_2\$yw_3$, where x is a k-word and y is a $(k+1)$-word. Construct a (E, A, kP)-machine M that first guesses the appropriate $y_i b_i$ using E. It then places the first pebble in front of each $z_i c_i$ in parallel, using A, and then uses the assumed-to-exist (E, A, $(k-1)$P)-machine to complete the checking. M uses only k pebbles and is of linear size. □

Lemma 4.11. *Let $k \geqslant 1$, and let $w = x\$y\$z$, for some symbol $\$ \notin xyz$. Then there is an l-(E, A, $(k-1)$P)-machine of size* O(n) *that checks whether y is a k-word.*

Proof. For $k = 1$, we do not need the pebble, as there is an easy way to construct an (E, A, T)-machine for this. For $k \geqslant 1$, let M' be an l-(E, A, $(k-1)$P)-machine of size O(n) that recognizes k-words, and let $w = x\$y\$z$. We construct an l-(E, A, kP)-machine M to check if y is a $(k+1)$-word. In parallel, M does the following: (i) Checks that every index in y is a k-word, by applying M' in parallel to all indices; (ii) verifies that the derivative of the first index is a sequence of 0s, and the derivative of the last index is a sequence of 1s; and (iii) checks, in parallel for all pairs of adjacent indices, that derivatives grow by 1. This third task is carried out by using E to guess the first nonidentical corresponding bits, and checking what has to be checked in parallel (see the proof of Lemma 4.6). To verify that all the previous bits are identical, M places the first pebble in front of the index, positions itself, using A, in front of the bit to be checked, and uses the l-(E, A, $(k-1)$P)-machine of Lemma 4.10. Similar l-(E, A, $(k-1)$P)-machine's are used to verify that the guessed bit is nonidentical to its corresponding bit, and to check whether the remaining significant bits are 1 in the first index, and 0 in the second index.

The resulting machine uses k pebbles and is of linear size. □

Lemma 4.12. *For each $k \geqslant 0$, there is an l-(E,A,kP)-machine of size* $O(n)$ *that checks if two k-words appearing as marked subwords of a given word are identical.*

Proof. For $k = 0$, the claim is trivial. For $k \geqslant 1$, let $w = w_1\$x\$w_2\$y\$w_3$, such that $x$ and $y$ are k-words. The l-(E, A, kP)-machine has to verify that each index in $x$, taken with its subsequent bit, appears in $y$. This can be done by working in parallel, placing the first pebble in front of each index in $x$ (which is a $(k-1)$-word), and using the l-(E, A, $(k-1)$P)-machine from Lemma 4.10 to complete the check. The resulting machine uses k pebbles and is of linear size. □

Theorem 4.13. *For each $k \geqslant 0$, l-$(E,A,kP) \underset{k+2}{\longrightarrow} \emptyset$ and l-$(E,A,kP) \underset{k+1}{\longrightarrow} E$.*

Proof. Define $L_n = \{\$w_1\$w_2\cdots\$w_m\#w \mid m \geqslant 1,\ w_1,\ldots,w_m,\ w$ are k-words, and for some i, $w_i = w\}$. We exhibit an l-(E, A, kP)-machine M of linear size that accepts L_n. M checks that the input word is a sequence of k-words according to Lemma 4.11. (For this it needs only $k-1$ pebbles.)

In parallel, it guesses an i, and checks, using k pebbles, if w_i is identical to w, by Lemma 4.12. However, the smallest DFA accepting L_n is of size at least $2^{[k+2]}(n)$ (see proof of Theorem 4.7). □

To complete our results on succinctness, we note the following. Both the upper and lower bounds we have established for ξ-machines, for any $\xi \subseteq \{$E, A, kP$\}$, grow by an exponential if bounded concurrency (i.e., the C feature) is added. The upper bounds are obtained by taking the Cartesian product of states to translate a (C, ξ)-machine into a ξ-machine with a blowup of $exp[1]$, as in [5]. The lower bounds can be proved as we did here, but by using the C feature to count to n wherever needed with $O(\log n)$ states, rather than $O(n)$, as in [5].

5. PDL of multi-pebble automata

In this section we show that the validity problem for PDL$_{k\mathrm{P}}$ is complete for $(k+1)$-fold exponential time.

Lemma 5.1. *For each $k \geqslant 0$, there is a kP-machine of size* $O(n)$ *that checks if two $(k-1)$-words appearing as marked subwords of a given word are identical.*

Proof. The kP-machine scans the indices of the first $(k-1)$-word one by one, using the first pebble. After marking an index $x$, which is a $(k-2)$-word, it finds $x$ in the second word using $k-1$ pebbles, according to Lemma 4.5. □

Theorem 5.2. *The validity problem for $PDL_{k\mathrm{P}}$ is complete for deterministic $exp[k+1]$ time.*

Proof. The upper bound is obtained by first transforming the kP-machine into an E-machine with an $exp[k]$ growth in size, as per Theorem 4.2, and then applying the exponential time decision procedure for $\mathrm{PDL}_{\mathrm{E}}$ (cf. [11]). We now prove a matching lower bound.

The basic framework of the proof is similar to the proofs appearing in [9, 10] for $\mathrm{PDL}_{\mathrm{C}}$ and $\mathrm{PDL}_{\mathrm{A,E,C}}$. These, in turn, extend and generalize the original proof of [7] for PDL. The idea of the proof in [7] was to simulate a linear-space-bounded alternating Turing machine in PDL. Given such a machine M, and an input word x, a formula $F_{M,x}$ is constructed in polynomial time, such that M accepts x iff $F_{M,x}$ is satisfiable. This can be shown to prove the desired result, since if $F_{M,x}$ were satisfiable in less than exponential time it could be decided whether x is accepted by M in time that would contradict the space bound on M for an appropriately chosen x, since APSPACE=EXPTIME. The formula $F_{M,x}$ is constructed in such a way that any satisfying model must "contain" a computation tree of M on x.

The proofs in [9, 10] work similarly, except that, since the required time bounds for the PDL's considered there were $exp[2]$ and $exp[3]$, the alternating Turing machines had to be bounded by space $exp[1]$ and $exp[2]$. Much of the groundwork for what we call here k-words was also set up in [10], so that building upon that we are able to carry the technique through for our purposes here. We now describe the main parts of the proof; more details can be found in [10].

Let $M = (Q, \Sigma, \Gamma, q_0, b, \delta, U)$ be an alternating Turing machine operating in space bounded by $exp[k]$, and let x be an input word of size n. We define a linear size formula $F_{M,x}$ in $\mathrm{PDL}_{k\mathrm{P}}$, that "forces" an encoding of each configuration of M of length $2^{[k]}(n)$ by a sequence of $2^{[k]}(n)$ states in any satisfying model.[6] More specifically, there is one state for each cell of M's tape, and every two such states are separated by a sequence of states that represents an index indicating the location in the configuration. Indeed, we represent a configuration of length $2^{[k]}(n)$ by a k-word whose derivative is the configuration itself. The formula $F_{M,x}$ employs $k+4$ atomic programs: one for the transitions between the configurations, denoted by $\vdash$, one for the transitions between locations inside a configuration, denoted by α, and $k+2$ atomic programs used to "encode" the indices, denoted by $0, 1, \$_1, \ldots, \$_k$.

The atomic formulas are: P_σ, where $\sigma \in \Gamma$, Q_q, where $q \in Q$, and H. Informally, P_σ means that the current cell contains σ, Q_q means the current state is q, and H means that the head is positioned at the current cell. In the original proof for PDL [7], n atomic formulas $P_{\sigma,i}$ were used (where n is the length of the configuration), with $P_{\sigma,i}$ meaning that the cell at location i contains σ; similarly, H_i was used to indicate that the head is at location i. With this setup, the model satisfying the formula consisted of a state for each configuration in the computation, and in this state the atomic formulas that precisely describe this configuration were all true. In our case, the length of the

[6] We assume, without loss of generality, that the basis of the exponentials in the space bound on M is 2.

configuration of $2^{[k]}(n)$, but we would still like to describe the machine's behavior by a formula of size $O(n)$. To this end, we use P_σ, and we will make sure that P_σ is true in any state of the model that corresponds to a cell containing σ. The location of the cell (i.e., the i of the corresponding $P_{\sigma,i}$ from [7]) will be represented by a sequence of states in the model, as explained earlier. Similarly, H is true in one state of the configuration only – the one corresponding to the cell that is pointed to by the head.

$F_{M,x}$ is constructed as a conjunction of formulas, some for stating that the model indeed represents a computation of M (e.g., the transitions from configuration to configuration are according to the those of M, etc.), and others that guarantee that every model satisfying the first set of formulas indeed describes a computation of M. In order to construct these formulas, we use several automata. The more complex and bigger of these automata are of size $O(n)$ only, and now we describe them:

(1) A kP-machine, denoted A, that checks if all the indices represent increasing sequences; i.e., that we have a k-word. By Lemma 4.6, such a machine exists, and is of linear size.

(2) A kP-machine, denoted B, that moves from a state representing some location in the configuration to the corresponding location in the next configuration. It has to be able to move from any index to the identical index in the next configuration. In attempting this, it has to be able to check if the two indices are identical. Since the indices are $(k-1)$-words, by Lemma 5.1 such a machine exists, and is of linear size.

We also use additional kP-machines, R and L, that move from a state in the model corresponding to some tape cell to the state corresponding to the following or previous cell in the same configuration, respectively.

We now describe some of the formulas constituting $F_{M,x}$. Letting $x = \sigma_1 \ldots \sigma_n$, the following formula describes the initial configuration, by asserting that the first n cells contain $\sigma_1, \ldots, \sigma_n$, the head is located at the first cell, and from cell $n+1$ onward there are only blanks:

$$\bigwedge_{i=1}^{n} [\vdash; R^i]P_{\sigma_i} \wedge [\vdash](\neg H \wedge [R]H \wedge [R^2R^*]\neg H \wedge [R^{n+1}R^*]P_b).$$

The next formula asserts that the contents of cells not pointed to by the head remain the same in the next configuration:

$$[R^*] \bigwedge_{\sigma\in\Gamma} ((\neg H \wedge P_\sigma) \Rightarrow [B]P_\sigma).$$

We now assert that the state remains the same in the entire configuration:

$$\bigwedge_{q\in Q} (Q_q \Rightarrow [R^*]Q_q).$$

This formula states that each configuration is "represented" by a k-word:

$$[R^*](\langle\vdash\rangle\ \mathit{true} \Rightarrow \langle A; \vdash\rangle\ \mathit{true}).$$

Finally, we show how to assert that the universal states behaves correctly:

$$[R^*]\left(\bigwedge_{\sigma\in\Gamma}\bigwedge_{q\in U}(H\wedge P_\sigma\wedge Q_q)\right.$$
$$\Rightarrow\left(\bigwedge_{(q,\sigma,q',\sigma',R)\in\delta}\langle B;R\rangle(H\wedge P_{\sigma'}\wedge Q_{q'})\right.$$
$$\left.\left.\wedge\bigwedge_{(q,\sigma,q',\sigma',L)\in\delta}\langle B;L\rangle(H\wedge P_{\sigma'}\wedge Q_{q'})\right)\right).\qquad\square$$

The validity problem for $\text{PDL}_{\text{E},k\text{P}}$ is also complete for $(k+1)$-exponential time, since by Proposition 4.3 we also have $(\text{E},k\text{P})\xrightarrow{k}\text{E}$.

In contrast, we have not been able to match the upper and lower bounds for pebbles combined with alternation: We have an upper bound of $exp[2k+1]$ time for $\text{PDL}_{\text{E,A},k\text{P}}$, but our best lower bound is $exp[k+2]$ time. To prove this lower bound, we use the previous proof starting with an alternation Turing machine operating in $exp[k+1]$ space. Then, using Lemmas 4.11 and 4.12 instead of Lemmas 4.5 and 4.6, we get two appropriate (E, A,kP)-machines of linear size. We can also use the C feature in these two automata, reducing their size to $\text{O}(\log n)$ and thereby proving:

Theorem 5.3. *The validity problem for* $\text{PDL}_{\text{E,A,C},k\text{P}}$ *is hard for deterministic* $exp[k+3]$ *time.*

Acknowledgements

We wish to thank one of the ICALP '94 referees for bringing Refs. [1, 2, 8] to our attention.

References

[1] J.C. Birget, Two-way automata and length-preserving homomorphisms, Report No. 109, Dept. of Computer Science, University of Nebraska, 1990.

[2] J.C. Birget, State-complexity of finite-state devices, state compressibility and incompressibility, *Math. Systems Theory* **26** (1993) 237–269.

[3] M. Blum and C. Hewitt, Automata on a 2-dimensional tape, in: *Proc. 8th IEEE Symp. Switching and Automata Theory* (1967) 155–160.

[4] A.K. Chandra, D. Kozen and L.J. Stockmeyer, Alternation, *J. ACM* **28** (1981) 114–133.

[5] D. Drusinsky and D. Harel, On the power of bounded concurrency I: finite automata, *J. ACM* **41** (1994) 517–539; preliminary version appeared in: *Proc. Concurrency '88*, Lecture Notes in Computer Science, Vol. 335 (Springer, New York, 1988) 74–103.

[6] A. Ehrenfeucht and P. Zeiger, Complexity measures for regular expressions, *J. CSS* **12** (1976) 134–146.

[7] M.J. Fischer and R.E. Ladner, Propositional dynamic logic of regular programs, *J. CSS* **18** (1979) 194–211.

[8] P. Goralcik, A. Goralcikova and V. Koubek, Alternation with a pebble, *Inform Process. Lett.* **38** (1991) 7–13.

[9] D. Harel, A thesis for bounded concurrency, in: *Proc. 14th Symp. on Math. Found. of Comput. Sci.*, Lecture Notes in Computer Science, Vol. 379 (Springer, Berlin, 1989) 35–48.

[10] D. Harel, R. Rosner and M. Vardi, On the power of bounded concurrency III: reasoning about programs, in: *Proc. 5th Symp. on Logic in Computer Science* (IEEE Press, New York, 1990) 479–488.

[11] D. Harel and R. Sherman, Propositional dynamic logic of flowcharts, *Inform. and Control* **64** (1985) 119–135.

[12] T. Hirst, Succinctness results for statecharts, M.Sc. Thesis, Bar-Ilan Univ., Ramat Gan, Isreal, 1989 (in Hebrew).

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

[14] R. Ladner, R. Lipton and L. Stockmeyer, Alternating pushdown automata, *SIAM J. Comput.* **13** (1984) 135–155; Preliminary version appeared in: *Proc. 19th IEEE Symp. on Foundations of Computer Science* (1978) 92–106.

[15] A.R. Meyer and M.J. Fischer, Economy of description by automata, grammars, and formal systems, in: *Proc. 12th IEEE Symp. on Switching and Automata Theory* (1971) 188–191.

[16] V.R. Pratt, Models of program logics, in: *Proc. 20th IEEE Symp. Foundations of Computer Science* (1979) 115–122.

[17] V.R. Pratt, Using graphs to understand PDL, in: D.C. Kozen, ed., *Proc. Workshop on Logics of Programs*, Lecture Notes in Computer Science, Vol. 131 (Springer, Berlin, 1981) 387–396.

[18] M.O. Rabin and D. Scott, Finite automata and their decision problems, *IBM J. Res.* **3** (1959) 115–125.

[19] J.C. Shepherdson, The reduction of two-way automata to one-way automata, *IBM J. Res. Develop.* **3** (1959).

[20] M. Sipser, Lower bounds on the size of sweeping automata, in: *Proc. 11th ACM Symp. on Theory of Comput.* (1979).