Alternating Automatic Register Machines*

Ziyuan Gao[1], Sanjay Jain[2], Zeyong Li[3], Ammar Fathin Sabili[2], and Frank Stephan[1,2]

[1] Department of Mathematics, National University of Singapore, 10 Lower Kent Ridge Road, S17, Singapore 119076, Republic of Singapore
matgaoz@nus.edu.sg

[2] School of Computing, National University of Singapore, 13 Computing Drive, COM1, Singapore 117417, Republic of Singapore
{sanjay,fstephan}@comp.nus.edu.sg
athin2008@gmail.com

[3] Center for Quantum Technologies, National University of Singapore, 3 Science Drive 2, S15, Singapore 117543, Republic of Singapore
li.zeyong@u.nus.edu

Abstract. This paper introduces and studies a new model of computation called an Alternating Automatic Register Machine (AARM). An AARM possesses the basic features of a conventional register machine and an alternating Turing machine, but can carry out computations using bounded automatic relations in a single step. One surprising finding is that an AARM can recognise some NP-complete problems, including 3SAT (using a particular coding), in $\mathcal{O}(\log^* n)$ steps. We do not yet know if every NP-complete problem can be recognised by some AARM in $\mathcal{O}(\log^* n)$ steps.

Furthermore, we study an even more computationally powerful machine, called a Polynomial-Size Padded Alternating Automatic Register Machine (PAARM), which allows the input to be padded with a polynomial-size string. It is shown that each language in the polynomial hierarchy can be recognised by a PAARM in $\mathcal{O}(\log^* n)$ steps, while every language recognised by a PAARM in $\mathcal{O}(\log^*(n))$ steps belongs to PSPACE. These results illustrate the power of alternation when combined with computations involving automatic relations, and uncover a finer gradation between known complexity classes.

Keywords: Theory of Computation · Computational Complexity · Automatic Relation · Register Machine · Nondeterministic Complexity · Alternating Complexity · Measures of Computation Time

* Z. Gao (as RF) and S. Jain (as Co-PI), F. Stephan (as PI) have been supported by the Singapore Ministry of Education Academic Research Fund grant MOE2019-T2-2-121 / R146-000-304-112. Furthermore, S. Jain is supported in part by NUS grant C252-000-087-001. Further support is acknowledged for the NUS tier 1 grants AcRF R146-000-337-114 (F. Stephan as PI) and R252-000-C17-114 (F. Stephan as PI).

1 Introduction

Automatic structures generalise the notion of regularity for languages to other mathematical objects such as functions, relations and groups, and were discovered independently by Hodgson [13,12], Khoussainov and Nerode [16] as well as Blumensath and Grädel [1,2]. One of the original motivations for studying automaticity in general structures came from computable structure theory, in particular the problem of classifying the isomorphism types of computable structures and identifying isomorphism invariants. In computer science, automatic structures arise in the area of infinite state model checking; for example, Regular Model Checking, a symbolic framework for modelling and verifying infinite-state systems, can be expressed in Existential Second-Order Logic over automatic structures [19]. Although finite-state transducers are a somewhat more popular extension of ordinary finite-state automata for defining relations between sets of strings, there are several advantages of working with automatic relations, including the following: (1) in general, automatic relations enjoy better decidability properties than finite-state transducers; for example, equivalence between ordinary automata is decidable while this is not so for finite-state transducers; (2) automatic relations are closed under first-order definability [12,16,15] while finite-state transducers are not closed under certain simple operations such as intersection and complementation.

In this paper, we introduce a new model of computation, called an *Alternating Automatic Register Machine (AARM)*, that is analogous to an alternating Turing machine but may incorporate *bounded* automatic relations[4] into each computation step. The main motivation is to try to discover new interesting complexity classes defined via machines where automatic relations are taken as primitive steps, and use them to understand relationships between fundamental complexity classes such as P, PSPACE and NP. More powerful computational models are often obtained by giving the computing device more workspace or by allowing non-deterministic or *alternating* computations, where alternation is a well-known generalisation of non-determinism. We take up both approaches in this work, extending the notion of alternation to automatic relation computations. An AARM is similar to a conventional register machine in that it consists of a *register* R containing a string over a fixed alphabet at any point in time, and the contents of R may be updated in response to *instructions*. One novel feature of an AARM is that the contents of the register can be non-deterministically updated using an automatic relation. Specifically, an instruction J is an automatic relation. Executing the instruction, when the content of the register R is r, means that the contents of R is updated to any x in $\{x : (x, r) \in J\}$; if there is no such x, then the program halts. Each AARM contains two finite classes, denoted here as A and B, of instructions; during a computation, instructions are selected alternately from A and B and executed.

[4] This means there is a constant bounding $|y| - |x|$ for each pair (x, y) of strings in the relation; see Section 2. Since we only consider bounded automatic relations in this paper, such relations will occasionally be called "automatic relations".

To further explain how a computation of an AARM is carried out, we first recall the notion of an *alternating Turing machine* as formulated by Chandra, Kozen and Stockmeyer [5]. As mentioned earlier, alternation is a generalisation of non-determinism, and it is useful for understanding the relationships between various complexity classes such as those in the polynomial hierarchy (PH). The computation of an alternating Turing machine can be viewed as a game in which two players – Anke and Boris – make moves (not necessarily strictly alternating) beginning in the start configuration of the machine with a given input w [8]. Anke moves from existential configurations (configurations with an existential state), and Boris from universal configurations (configurations with a universal state) to successor configurations according to the machine's transition function. Anke wins if, after a finite number of moves, an accepting configuration is reached. The input w is then accepted by the machine iff Anke has a winning strategy. Our definition of an AARM computation is inspired by this game-theoretic interpretation of alternating Turing machines. Given an input w, which is the string in R at the start of the computation, Anke and Boris move in alternating turns, with Anke moving first. During Anke's turn, she carries out any single instruction in the predefined set A, possibly changing the contents of the register. Boris moves similarly during his turn, except that he carries out an instruction in B. Anke wins if a configuration is reached such that Boris is no longer able to carry out any instructions in B, and w is *accepted* iff Anke has a winning strategy.

An extended version of this paper that presents examples and additional basic results on AARMs and PAARMs is available at `https://arxiv.org/pdf/2111.04254.pdf`.

2 Preliminaries

Let Σ denote a finite alphabet. We consider set operations including union ($\cup$), concatenation ($\cdot$), Kleene star ($*$), intersection ($\cap$) and complement ($\neg$). Let Σ^* denote the set of all strings over Σ. A *language* is a set of strings. Let the empty string be denoted by ε. For a string $w \in \Sigma^*$, let $|w|$ denote the length of w and $w = w_1 w_2 ... w_{|w|}$ where $w_i \in \Sigma$ denotes the i-th symbol of w. Fix a special symbol $\#$ not in Σ. Let $x, y \in \Sigma^*$ such that $x = x_1 x_2 \dots x_m$ and $y = y_1 y_2 \dots y_n$. Let $x' = x'_1 x'_2 \dots x'_r$ and $y' = y'_1 y'_2 \dots y'_r$ where $r = \max(m, n)$, $x'_i = x_i$ if $i \leq m$ else $\#$, and $y'_i = y_i$ if $i \leq n$ else $\#$. Then, the *convolution* of x and y is a string over $(\Sigma \cup \{\#\}) \times (\Sigma \cup \{\#\})$, defined as $conv(x, y) = (x'_1, y'_1)(x'_2, y'_2) \dots (x'_r, y'_r)$. A relation $J \subseteq X \times Y$ is *automatic* if the set $\{conv(x, y) : (x, y) \in J\}$ is regular, where the alphabet is $(\Sigma \cup \{\#\}) \times (\Sigma \cup \{\#\})$. Likewise, a function $f : X \to Y$ is *automatic* if the relation $\{(x, y) : x \in domain(f) \wedge y = f(x)\}$ is automatic [23]. An automatic relation J is *bounded* if $\exists$ constant c such that $\forall (x, y) \in J, \text{abs}(|y| - |x|) \leq c$. On the other hand, an unbounded automatic relation has no such restriction. Automatic functions and relations have a particularly nice feature as shown in the following theorem.

Theorem 1 ([12,16]). *Every function or relation which is first-order definable from a finite number of automatic functions and relations is automatic, and the corresponding automaton can be effectively computed from the given automata.*

3 Results

3.1 Alternating Automatic Register Machines

An *Alternating Automatic Register Machine* (AARM) consists of a *register* R and two finite sets A and B of *instructions*. A and B are not necessarily disjoint. Formally, we denote an AARM by M and represent it as a quadruple (Γ, Σ, A, B). At any point in time, the register contains a string, possibly empty, over a fixed alphabet Γ called the *register alphabet*. The current string in R is denoted by r. Initially, R contains an input string over Σ, an *input alphabet* with $\Sigma \subseteq \Gamma$. Strings over Σ will sometimes be called *words*. The contents of the register may be changed in response to an instruction. An instruction $J \subseteq \Gamma^* \times \Gamma^*$ is a bounded automatic relation; this changes the contents of R to some x such that $(x, r) \in J$ (if such an x exists). The instructions in A and B are labelled $I_1, I_2, \ldots,$ (in no particular order and not necessarily distinct). A *configuration* is a triple (ℓ, w, r), where I_ℓ is the current instruction's label and $w, r \in \Gamma^*$. Instructions are generally nondeterministic, that is, there may be more than one way in which the string in R is changed from a given configuration in response to an instruction. A *computation history* of an AARM *with input* w for any $w \in \Sigma^*$ is a finite or infinite sequence of configurations

$$c_1, c_2, c_3, \ldots$$

such that the following conditions hold. Let $c_i = (\ell_i, w_i, r_i)$ for all i.

- $r_1 = w$. We call c_1 the *initial configuration* of the computation history.
- For all i, $(w_i, r_i) \in I_{\ell_i}$. This means that I_{ℓ_i} can be carried out using the current register contents, changing the contents of R to w_i.
- Instructions executed at odd terms of the sequence belong to A, while those executed at even terms belong to B:
$$I_{\ell_i} \in \begin{cases} A \text{ if } i \text{ is odd;} \\ B \text{ if } i \text{ is even.} \end{cases}$$
- If c_{i+1} is defined, then $r_{i+1} = w_i$. In other words, the contents of R are (nondeterministically) updated according to the instruction and register contents of the previous configuration.
- Suppose i is odd (resp. even) and $c_i = (I_{\ell_i}, w_i, r_i)$ is defined. If there is some $I_\ell \in B$ (resp. $I_\ell \in A$) with $\{x : (x, w_i) \in I_\ell\}$ nonempty, then c_{i+1} is defined. In other words, the computation continues so long as it is possible to execute an instruction from the appropriate set, either A or B, at the current term.

We interpret a computation history of an AARM as a sequential game between two players, Anke and Boris, where Anke moves during odd turns and Boris

moves during even turns. During Anke's turn, she must pick some instruction J from A such that $\{x : (x, r) \in J\}$ is nonempty and select some $w \in \{x : (x, r) \in J\}$; if no such instruction exists, then the game terminates. The contents of R are then changed to w at the start of the next turn. The moving rules for Boris are defined analogously, except that he must pick instructions only from B. Anke *wins* if the game terminates after a finite number of turns and she is the last player to execute an instruction; in other words, Boris is no longer able to carry out an instruction in B and the *length* of the game (or computation history), measured by the total number of turns up to and including the last turn, is odd. Boris wins the game if Anke cannot win (this includes the case that Anke does not make any move). The AARM *accepts* a word w if Anke can move in such a way that she will always win a game with an initial configuration (ℓ, v, w) for some $I_\ell \in A$ and $v \in \Gamma^*$, regardless of how Boris moves. To state this acceptance condition more formally, one could define Anke's and Boris' *strategies* to be functions $\mathcal{A}$ and $\mathcal{B}$ respectively with $\mathcal{A} : (\mathbb{N} \times \Gamma^* \times \Gamma^*)^* \times \Gamma^* \mapsto A \times \Gamma^*$ and $\mathcal{B} : (\mathbb{N} \times \Gamma^* \times \Gamma^*)^* \times \Gamma^* \mapsto B \times \Gamma^*$, which map each segment of a computation history together with the current contents of R to a pair specifying an instruction as well as the new contents of R at the start of the next round according to the moving rules given earlier. The AARM accepts w if there is an $\mathcal{A}$ such that for every $\mathcal{B}$, there is a finite computation history $\langle c_1, \ldots, c_{2n+1} \rangle$ where

- $c_i = (\ell_i, w_i, r_i)$ for each i,
- $r_1 = w$,
- $\mathcal{A}((\langle c_i : i < 2j+1 \rangle, r_{2j+1})) = (I_{\ell_{2j+1}}, w_{2j+1})$ for each $j \in \{0, \ldots, n\}$,
- $\mathcal{B}((\langle c_i : i < 2k \rangle, r_{2k})) = (I_{\ell_{2k}}, w_{2k})$ for each $k \in \{1, \ldots, n\}$;
- there is no move for B in c_{2n+1}, that is no instruction in B contains a pair of the form $(\cdot, w_{2n+1})$.

Here $\langle c_i : i < k \rangle$ denotes the sequence $\langle c_1, \ldots, c_{k-1} \rangle$, which is empty if $k \leq 1$. Such an $\mathcal{A}$ is called a *winning strategy for Anke with respect to* (M, w). Given a winning strategy $\mathcal{A}$ for Anke with respect to (M, w) and any strategy $\mathcal{B}$, the corresponding computation history of M with input w is unique and will be denoted by $\mathcal{H}(\mathcal{A}, \mathcal{B}, M, w)$. In most subsequent proofs, $\mathcal{A}$ and $\mathcal{B}$ will generally not be defined so formally. Set

$$L(M) := \{w \in \Sigma^* : M \text{ accepts } w\};$$

one says that M *recognises* $L(M)$. Note that even though we have given the description of AARM via alternation of moves by Anke and Boris, it is possible to define games where strict alternation is not needed. Furthermore, a constant amount of extra state information can be stored in the register.

Definition 1 (Alternating Automatic Register Machine Complexity). *Let $M = (\Gamma, \Sigma, A, B)$ be an AARM and let $t \in \mathbb{N}_0$. For each $w \in \Sigma^*$, M* accepts *w* in time *t if Anke has a winning strategy $\mathcal{A}$ with respect to (M, w) such that for any strategy $\mathcal{B}$ played by Boris, the length of $\mathcal{H}(\mathcal{A}, \mathcal{B}, M, w)$ is not more than t. (As defined earlier, $\mathcal{H}(\mathcal{A}, \mathcal{B}, M, w)$ is the computation history of M with input w when $\mathcal{A}$ and $\mathcal{B}$ are applied.)*

Let f be a function mapping $\mathbb{N}_0$ to the real numbers. M recognises in time *$f(n)$ if for each $w \in L(M)$, M accepts w in time $f(|w|)$. $AAL[f(n)]$ denotes the family of languages recognised by AARMs that recognise in time $\mathcal{O}(f(n))$.*

It can be shown that AARMs recognise precisely the family of all recursively enumerable languages [9, Theorem 5.4], and that AAL$[f(n)]$ is closed under the usual set-theoretic Boolean operations as well as the regular operations [9, Theorem 5.5].

We recall that an alternating Turing machine that decides in $\mathcal{O}(f(n))$ time can be simulated by a deterministic Turing machine using $\mathcal{O}(f(n))$ space. The following theorem gives a similar connection between the time complexity of AARMs and the space complexity of deterministic Turing machines.

Theorem 2. *For any f such that $f(n) \geq n$, $AAL[f(n)] \subseteq DSPACE[(n + f(n))f(n)] = DSPACE[f(n)^2]$.*

Proof. Given an AARM M, there is a constant c such that each register update by an automatic relation used to define an instruction of M increases the length of the register's contents by at most c. Thus, after $\mathcal{O}(f(n))$ steps, the length of the register's contents is $\mathcal{O}(n + f(n))$. As implied by [4, Theorem 2.4], each computation of an automatic relation with an input of length $\mathcal{O}(n + f(n))$ can be simulated by a nondeterministic Turing machine in $\mathcal{O}(n + f(n))$ steps; this machine can then be converted to a deterministic space $\mathcal{O}(n + f(n))$ Turing machine. If M accepts an input w, then there are $\mathcal{O}(f(n))$ register updates by automatic relations when Anke applies a winning strategy, and so there is a deterministic Turing machine simulating M's computation with input w using space $\mathcal{O}((n + f(n))f(n))$.

As a consequence, one obtains the following analogue of the equality between AP (classes of languages that are decided by alternating polynomial time) and PSPACE.

Corollary 1. *$\bigcup_k AAL[n^k] = PSPACE$.*

Proof. The containment relation AAL$[f(n)] \subseteq$ DSPACE$[(n+f(n))f(n)]$ in Theorem 2 holds whether or not the condition $f(n) \geq n$ holds. Furthermore, the computation of an alternating Turing machine can be simulated using an AARM, where the transitions from existential (respectively, universal) states correspond to the instructions for Anke (respectively, Boris), and each computation step of the alternating Turing machine corresponds to a move by either player. Therefore PSPACE, which is equal to AP – the class of languages decided by alternating polynomial time Turing machines, is contained in $\bigcup_k$ AAL$[n^k]$.

We come next to a somewhat surprising result: an AARM-program can recognise 3SAT using just $\mathcal{O}(\log^* n)$ steps. To prove the theorem, we give the following lemma, which illustrates most of the power of AARMs.

Lemma 1 (Log-Star Lemma). *Let $u, v \in \Sigma^*$. Let $\#, \$ \notin \Sigma$. Then both languages $\{u'\$v' : u' \in \#^*u\#^*, v' \in \#^*v\#^* \text{ and } u = v\}$ and $\{u'\$v' : u' \in \#^*u\#^*, v' \in \#^*v\#^* \text{ and } u \neq v\}$ are in $AAL[\log^* n]$.*

The Log-Star Lemma essentially states that a comparison of two substrings can be done by an AARM in $\mathcal{O}(\log^* n)$ time. This is done by ignoring the unnecessary symbols in the register by replacing them with #'s and adding a separator ($) between the two strings.

We now prove the Log-Star Lemma. Additional technical details for the proof can be found in [9, Section 4]. The algorithm below recursively reduces the problem to smaller sizes of u, v in constant number of steps (the maximum of the length of u, v is reduced logarithmically in constant number of steps). For the base case, if size of u or v is bounded by a constant, then clearly both languages can be recognized in one step.

For larger size u, v, the algorithm/protocol works as follows. For ease of explanation, suppose Anke is trying to show that $u = v$ (case of Anke showing $u \neq v$ will be similar). Given input $s = u'\$v'$, player Anke will try to give each symbol except $'s a mark $\in \{0, 1, 2, 3\}$ as follows:

1. For each # in u' and v', the mark of 3 will be given.
2. For the contiguous symbols of u, starting from the first symbol, the following infinite marking will be given (whitespaces are for the ease of readability and not part of marking):

   20 21 200 201 210 211 2000 2001 2010 2011 2100 2101 2110 2111 20000 $\cdots$

   Namely, a series of blocks of string in ascending length-lexicographical order. Let T be the infinite sequence

   $$\overbrace{20}\ \overbrace{21}\ \overbrace{200}\ \overbrace{201}\ \overbrace{210}\ \overbrace{211}\ \overbrace{2000}\ \overbrace{2001}\ \overbrace{2010}\ \overbrace{2011}\ \overbrace{2100}\ \overbrace{2101}\ \cdots.$$

   Given a string $s = u'\$v'$, where $u' \in \#^* u \#^*$ and $v' \in \#^* v \#^*$ for some $u, v \in \Sigma^*$, each contiguous subsequence of u (resp. v) whose sequence of positions is equal to the sequence of positions of T of some string in $2\{0,1\}^*$ such that the next symbol in T is 2 will be called a *block*. Each block starts with 2 followed by a binary string. Note that each block will have a size of at most $\mathcal{O}(\log n)$.
3. For the contiguous symbols of v, the marking will be similar.

The marking is considered *valid* if all above rules are satisfied. This is an example of a *valid* marking of $S =$ "$\#foobar\#\#\$foobar\#\#$":

s	#	f	o	o	b	a	r	#	#	$	f	o	o	b	a	r	#	#
Mark	3	2	0	2	1	2	0	3	3	$	2	0	2	1	2	0	3	3

If $u = v$ and the marking is *valid*, player Anke will guarantee that each symbol of u and v will be marked with exactly the same marking. However if $u \neq v$ and the marking is *valid*, either the length of u and v are different or there will be at least a single block which differs on at least one symbol between u and v. Therefore, player Boris can have the following choices of challenges:

1. Challenge that player Anke did not make a valid marking, or
2. $|u| \neq |v|$, or
3. the string in a specified block differs on at least one symbol between u and v.

Notice that $u = v$ if and only if player Boris could not successfully challenge player Anke. The first challenge will ensure that player Anke gave a valid marking. There are three possible cases of invalid marking:

(a) There is a # in u' or v' which is not given by a mark of 3. In this case, player Boris may point out its exact position. Here, player Boris needs 1 step.
(b) For u and v, the first block is not marked with "20". This can also be easily pointed out by player Boris. Here, player Boris needs 1 step.
(c) For u and v, a block is not followed by its successor. This can be pointed out by player Boris by checking two things: the length of the 'successor' block should be less than or equal to the length of the 'predecessor' block plus one, and the 'successor' block indeed should be the successor of the 'predecessor' block. Also, we note that the last block may be incomplete.
    (i) The length case can be checked by looking at how many symbols there are between the pair of 2's bordering each block. Let p and q be the length of 'predecessor' block and the 'successor' block respectively. In the case that the 'successor' block is not the last block (not incomplete), player Boris may challenge if $p \neq q$ and $p + 1 \neq q$. This can be done by marking both blocks with 1 separated by \$ and the rest with dummy symbols # and then doing the protocol for equality of the modified u and v recursively. As player Boris may try to find the 'short' challenge, player Boris will find the earliest block which has the issue and thus make sure that p is at most logarithmic in the maximum of the lengths of u and v. As q may be much larger than p, player Boris may limit the second block by taking at most $p + 2$ symbols.
    (ii) The successor case can be checked by the following observation. A successor of a binary string can be calculated by finding the last 0 symbol and flipping all digits from that position to the ending while maintaining the previous digits. As an illustration, the successor of "101100111" will be "101101000" where the symbols are separated in 3 parts: the prefix which is the same, the last 0 digit, which is underlined, becoming 1; and all 1 digits on suffix becoming 0. Player Boris then may challenge the first part not to be equal or the last part not to be the same length or not all 1's by providing the position of the last 0 on the 'predecessor' block (or the last 1 on the 'successor' block, if any). Checking the equality of two strings can be done recursively, also similarly applied for checking the length. Notice a corner case of all 1's which has the successor consisting of 1 followed by 0's with the same length, which can be handled separately. Also notice that the 'successor' block may be incomplete if it is the last block, which can also be handled in a similar manner as above.

For the second challenge, player Boris can (assuming the marking is valid) check whether the last two blocks of u and v are equal. Again, player Boris may limit it for a 'short' challenge so the checking size is decreasing to its log. For the third challenge, player Boris will specify the two blocks on u and v (same block on both) which differ on at least one symbol between u and v. Again, same protocol will apply and the size is decreasing to its log.

Thus, the above algorithm using one alternation of each of the players reduces the problem to logarithmic in the size of the maximum of the lengths of u and v. In particular, when the size of u and v are small enough, the checking will be done in constant number of steps. Thus, the complexity of the problem satisfies: $T(m) = c + T(\log m)$, where m denotes the maximum of the sizes of u and v. This recurence has the solution $T(m) = O(\log^* m)$. As the length of u, v are bounded by the length of the whole string, the lemma follows.

Theorem 3. *There is an NP-complete problem in* $AAL[\log^* n]$.

Proof. Consider a unary encoding of 3SAT where a variable x_i is represented by 0^i. We use $s_l \in \{+, -\}$ to represent whether the l-th literal is positive or negative. A clause $(x_i \vee \neg x_j \vee \neg x_k)$ is represented by $0^i + |0^j - |0^k -$ and different clauses are connected by &. A valid assignment is almost the same as the formula except that s_l is convoluted by the assignment $a_l \in \{T, F\}$. i.e. we are giving an assignment to each literal convoluted with its sign. We will show that $3\text{SAT} \in \text{AAL}[\log^* n]$. Given a 3SAT, player Anke will first nondeterministically assign the truth values of each literal and make sure that each clause has at least one satisfying literal in one step. Player Boris then needs to challenge player Anke on whether there are two literals which denote the same variable but have a different truth value. Player Boris will keep both literals and mark other symbols with some dummy symbols. It is now player Anke's job to prove that those two literals are not the same variables. By the Log-Star Lemma, it needs $\mathcal{O}(\log^* n)$ steps. Hence, $3\text{SAT} \in \text{AAL}[\log^* n]$.

The Log-Star Lemma can also be applied, using a technique similar to that in the proof of Theorem 3, to show that for any $k \geq 3$, the NP-complete problem k-*COLOUR* of deciding whether any given graph G is colourable with k colours belongs to $\text{AAL}[\log^* n]$. Using a suitable encoding of nodes, edges and colours as strings, Anke first nondeterministically assigns any one of k colours to each node and ensures that no two adjacent nodes are assigned the same colour; Boris then challenges Anke on whether there are two substrings of the current input that encode the same node but encode different colours.

As yet, we have no characterisation of those problems in NP which are in $\text{AAL}[\log^* n]$ and we think that for each such problem it might depend heavily on the way the problem is formatted. The reason is that it may be difficult to even prove whether or not P is contained in $\text{AAL}[\log^* n]$, due to the following proposition. However, we will show later that the class $\text{PAAL}[\log^* n]$ which is obtained from $\text{AAL}[\log^* n]$ by factoring in one space enlarging padding step contains the whole of PH plus something more and is contained in PSPACE.

Proposition 1. *If $P \subseteq AAL[\log^* n]$, then $P \subset PSPACE$.*

Proof. By Theorem 2 (the condition $f(n) \geq n$ is not necessary for the first containment relation to hold), $\mathrm{AAL}[\log^* n] \subseteq \mathrm{DSPACE}[(n+\log^*(n))\cdot\log^*(n)]$. Moreover, by the space hierarchy theorem [21, Corollary 9.4], one has $\mathrm{DSPACE}[(n + \log^*(n)) \cdot \log^*(n)] \subset \mathrm{DSPACE}[n^2]$. Thus if $\mathrm{P} \subseteq \mathrm{AAL}[\log^* n]$, then $\mathrm{P} \subset \mathrm{DSPACE}[n^2] \subset \mathrm{PSPACE}$.

3.2 Polynomial-Size Padded Alternating Automatic Register Machine

An AARM is constrained by the use of *bounded* automatic relations during each computation step. This is a real limitation: it can be shown that for any $f(n) = \Omega(\log\log n)$, the class of languages recognised in time $\mathcal{O}(f(n))$ by alternating automatic register machines that use *unbounded* automatic relations contain $\mathrm{DSPACE}\left[2^{2^{\mathcal{O}(f(n))}}\right]$ [18], and so by Theorem 2 and the space hierarchy theorem, this class properly contains $\mathrm{AAL}[f(n)]$.

We study the effect of allowing a polynomial-size padding to the input of an Alternating Automatic Register Machine on its time complexity; this new model of computation will be called a Polynomial-Size Padded Bounded Alternating Automatic Register Machine (PAARM). The additional feature of a polynomial-size padding will sometimes be referred to informally as a "booster" step of the PAARM. Intuitively, padding the input before the start of a computation allows a larger amount of information to be packed into the register's contents during a computation history. We show two contrasting results: on the one hand, even a booster step does not allow an PAARM with time complexity $\mathcal{O}(1)$ to recognise non-regular languages; on the other hand, the class of languages recognised by PAARMs in time $\mathcal{O}(\log^* n)$ encompasses the whole polynomial hierarchy.

Formally, a *Polynomial-Size Padded Bounded Alternating Automatic Register Machine* (PAARM) M is represented as a quintuple $(\Gamma, \Sigma, A, B, p)$, where Γ is the register alphabet, Σ the input alphabet, A and B are two finite sets of instructions and p is a polynomial. As with an AARM, the register R initially contains an input string over Σ, and R's contents may be changed in response to an instruction, $J \subseteq \Gamma^* \times \Gamma^*$ which is a bounded automatic relation. A computation history of a PAARM with input w for any $w \in \Sigma^*$ is defined in the same way that was done for an AARM, except that the initial configuration is (ℓ, x, wv) for some $I_\ell \in A$, some $x, v \in \Gamma^*$, $(x, wv) \in I_\ell$, where $v = @^k$ for a special symbol $@ \in \Gamma - \Sigma$ and $k \geq p(|w|)$. Think of $@^k$ as padding of the input. Anke's and Boris' strategies, denoted by $\mathcal{A}$ and $\mathcal{B}$ respectively, are defined as before. For any $u \in \Gamma^*$, a winning strategy for Anke with respect to (M, u) is also defined as before. Given any $w \in \Sigma^*$, M *accepts* w if for every $v \in @^*$, with $|v| \geq p(|w|)$, Anke has a winning strategy with respect to (M, wv). Similarly, M *rejects* w if for every $v \in @^*$, with $|v| \geq p(|w|)$, Boris has a winning strategy with respect to (M, wv). Note that the winning strategies need to be there for every long enough padding. If Anke and Boris do not satisfy the above properties, then they are not a valid pair.

Definition 2 (Polynomial-Size Padded Bounded Alternating Automatic Register Machine Complexity).

Let $M = (\Gamma, \Sigma, A, B, p)$ be a PAARM and let $t \in \mathbb{N}_0$. For each $w \in \Sigma^*$, M accepts *w in time t if for every $v = @^k$, where $k \geq p(|w|)$ and $@ \in \Gamma - \Sigma$, Anke has a winning strategy $\mathcal{A}$ with respect to (M, wv) and for any strategy $\mathcal{B}$ played by Boris, the length of $\mathcal{H}(\mathcal{A}, \mathcal{B}, M, wv)$ is not more than t.*

Let f be a function mapping $\mathbb{N}_0$ to the real numbers. M recognises *in time $f(n)$ if for each $w \in L(M)$, M accepts w in time $f(|w|)$. $PAAL[f(n)]$ denotes the family of languages recognised by PAARMs that recognise in time $\mathcal{O}(f(n))$.*

Remark 4 *Note that a PAARM-program can trivially simulate an AARM-program by ignoring the generated padding; thus $AAL[f(n)] \subseteq PAAL[f(n)]$. On the other hand, to simulate a booster step, an AARM-program needs $\mathcal{O}(p(n))$ steps as each bounded automatic relation step can only increase the length by a constant.*

As with AAL$[f(n)]$, the class PAAL$[f(n)]$ is closed under the usual set-theoretic Boolean operations as well as regular operations [9, Theorem 5.8]. Our main result concerning PAARMs is that they can recognise all languages in the polynomial hierarchy (PH) in $\mathcal{O}(\log^* n)$ time.

Theorem 5. *$PH \subseteq PAAL[\log^* n] \subseteq PSPACE$.*

To help with the proof, we first extend the Log-Star Lemma as follows. Configuration (Instantaneous description) of a Turing Machine [14] is a string xqw, where the head of Turing Machine is on the first symbol of w, the content of the tape is xw and the state of Turing Machine is q.

Lemma 2. *Checking the validity of a Turing Machine step, i.e., whether a configuration of Turing Machine follows another configuration (given as input, separated by a special separator symbol) can be done in $AAL[\log^* n]$, where n is the length of the shorter of the two configurations.*

Proof. Let the input be the two configurations of the Turing Machine, where the second configuration is supposedly the successive step of the first one and separated by a separator symbol. Each configuration consists of the content of the machine, the marker of the position of the tape head convoluted by the symbol it looks upon, and additional information about the state and the transitions including the symbol replacing the old one and the movement of the tape head. Now there are two things that need to be checked:

1. The configuration is "copied" correctly from the previous step. Note that a valid Turing Machine transition will change only the cell on the tape head and/or both of its neighbour; thus "copied" here means the rest of the tape content should be the same.
2. The local Turing step is correct.

For the first checking, the player who wants to verify, e.g. Anke, will give the infinite valid marking as used in the Log-Star Lemma. In addition, Anke also marks the position of the old tape head on the second configuration. Boris can then challenge the following:

(a) The Log-Star Lemma marking is not valid.
(b) The old tape head position is not marked correctly (in the intended position) on the second configuration.
(c) The string in a specified block differs, but not the symbols around the tape head.
(d) The length difference of the configuration is not bounded by constant.

Challenge (a) can be done in $\mathcal{O}(\log^* n)$ steps; this follows from the Log-Star Lemma. Challenge (b) can also be done in $\mathcal{O}(\log^* n)$ steps where both players reduce the block to focus on that position, and finally check whether it is on the same position or not. Challenge (c) can also be done in $\mathcal{O}(\log^* n)$ steps; this follows again from the Log-Star Lemma. Note that if Boris falsely challenges that the different symbol is around the tape head, Anke can counter-challenge by pointing out that at least one of its neighbours is a tape head. For challenge (d), note that a valid Turing Machine transition will only increase the length by at most one. Thus, Boris can pinpoint the last character of the shorter configuration and also its pair on another configuration, then check whether the longer one is only increased by up to one in length. This again can be done in $\mathcal{O}(\log^* n)$ steps.

For the second checking about the correctness of the Turing step, it can be done in a constant number of checks as a finite automaton can check the computation and determine whether the Turing steps are locally correct, that is, each state is the successor state of the previous steps head position and the symbol to the left or right of the new head position is the symbol following from the transition to replace the old symbol and so on. Therefore, all-in-all the validity of a Turing Machine step can be checked in $\mathrm{AAL}[\log^* n]$ steps.

Proof (of Theorem 5). Note that PH can be defined with alternating Turing machines [6]. We define Σ_k^P to be the class of languages recognised by alternating Turing Machine in polynomial time where the machine alternates between existential and universal states k times starting with existential state. We also define Π_k^P similarly but starting with universal state. PH then defined as the union of all Σ_k^P and Π_k^P for all $k \geq 0$. We now show $\Sigma_k^P \cup \Pi_k^P \subseteq \mathrm{PAAL}[\log^* n]$ for some fixed constant k. As the alternating Turing Machine runs in polynomial time on each alternation,the full computation (i.e., sequence of configurations) in one single alternation can be captured non-deterministically in $p(m)$ Turing Machine steps, for some polynomial p (which we assume to be bigger than linear), where m is the length of the configuration at the start of the alternation. In a PAARM-program, Anke first invokes a booster step to have a string of length at least $p^k(n)$. After that, Boris and Anke will alternately guess the full computation of the algorithm of length $p(p^i(n))$, $i = 0, 1, \ldots$, in their respective alternation: Boris guesses the first $p(n)$ computations (the first alternation), Anke then guesses the next $p(p(n))$ computations on top of it (the second alternation), etc. In addition, they also mark the position of the read head and symbol it looks upon in each step. Ideally, the PAARM-program will take k alternating steps to complete the overall algorithm. Note that a PAARM can keep multiple variables in the register by using convolution, as long as the number of

variables is a constant. Thus, we could store the k computations in k variables: $v_1, v_2, \cdots, v_k$. Now each player can have the following choices of challenges to what the other player did:

1. Copied some symbol wrongly from the input i.e. in v_1.
2. Two successive Turing Machine steps in the computation are not valid (at some v_i).
3. The last Turing Machine step on some computation (at some v_i) does not follow-up with the first Turing Machine step on the next computation (at v_{i+1}).

All the above challenges can be done in $\mathcal{O}(\log^* n)$ steps by Lemma 2. In particular, the third challenge needs one to compare the first Turing Machine configuration of v_{i+1} and the last Turing Machine configuration v_i, which can be done in a way similar to the proof of Lemma 2. Thus, $\Sigma_k^P \cup \Pi_k^P \subseteq \text{PAAL}[\log^* n]$ for every fixed constant k, therefore $PH \subseteq \text{PAAL}[\log^* n]$. As implied by [4, Theorem 2.4], $\text{PAAL}[\log^* n]$ can be simulated by an alternating Turing Machine in $\mathcal{O}(p(n))$ steps for some polynomial p; thus it is contained in AP (alternating polynomial time). As $AP = \text{PSPACE}$ [6], we finally have $\text{PAAL}[\log^* n] \subseteq \text{PSPACE}$.

In addition, we also get the following corollary about polynomial-time Turing reducibility. We recall that a *polynomial-time Turing reduction from problem A to problem B* is an algorithm to solve A in a polynomial number of steps by making a polynomial number of calls to an oracle solving B.

Corollary 2. *$PAAL[\log^* n]$ is closed under polynomial-time Turing reducibility.*

Proof. After Anke invokes the booster step, Boris will guess the accepting computation together with all of the oracle answers. Anke then can challenge Boris on either the validity of the computation (without challenging the oracle) or challenge one of the oracle answers. Both challenges can be done in the same fashion as in Theorem 5 but the latter needs one additional step to initiate the challenge of the oracle algorithm.

In order to obtain the next corollary, we use the fact that the problem of deciding $\text{TQBF}_{\log^*}$ – the class of true quantified Boolean formulas with $\log^* n$ alternations – does not belong to any fixed level of the polynomial hierarchy (PH) when PH does not collapse.

Corollary 3. *If PH does not collapse, then $PH \subset PAAL[\log^* n] \subseteq PSPACE$.*

Finally, we observe that Proposition 1 implies the following.

Proposition 2. *If $P = PSPACE$, then $AAL[\log^* n] \subset PAAL[\log^* n]$.*

The main results on complexity classes defined by AARMs are summed up in Figure 1 while those on complexity classes defined by PAARMs are summed up in Figure 2. For any function f, $\text{AAL}[f(n)]$ denotes the class of languages recognised by an AARM in $\mathcal{O}(f(n))$ time. An arrow is labelled with (a) reference(s) to the corresponding result(s) or definition(s) in the paper; folklore inclusions can be found in [14,21]. Results not stated in the present section are proven in the extended version of this paper [9].

3.3 Related Work

The idea of defining computing devices capable of performing single-step operations that are more sophisticated than the basic operations of Turing machines is not new. For example, Floyd and Knuth [7] studied *addition machines*, which are finite register machines that can carry out addition, subtraction and comparison as primitive steps. *Unlimited register machines*, introduced by Shepherdson and Sturgis [20], can copy the number in a register to any register in a single step. Bordihn, Fernau, Holzer, Manca and Martín-Vide [3] investigated another kind of language generating device called an *iterated sequential transducer*, whose complexity is usually measured by its number of states (or *state complexity*). More recently, Kutrib, Malcher, Mereghetti and Palano [17] proposed a variant of an iterated sequential transducer that performs length-preserving transductions on left-to-right sweeps. Automatic relations are more expressive than arithmetic operations such as addition or subtraction, and yet they are not too complex in that even one-tape linear-time Turing machines are computationally more powerful; for instance, the function that erases all leading 0's in any given binary word can be computed by a one-tape Turing machine in linear time but it is not automatic [22]. Despite the computational limits of automatic relations, we have shown that, rather surprisingly, an NP-complete problem, namely 3SAT, can be recognised by an AARM in $\mathcal{O}(\log^* n)$ steps. The results not only show a proof-of-concept for the use of automatic relations in models of computation, but also shed new light on the relationships between known complexity classes. In a companion paper [10], we have investigated models of computation where automatic functions/relations are combined with deterministic or non-deterministic register machines, called Deterministic Automatic Register Machines (DARM) and Non-Deterministic Automatic Register Machines (NDARM) respectively. Many of the results in [10] have a similar flavour to those in the current work; it is shown in [10], for example, that the membership problem for any Boolean grammar given in Binary Normal Form can be decided using an NDARM in $\mathcal{O}(n^2)$ time, while an NDARM can decide the membership problem for any context-free grammar in Chomsky Normal Form in $\mathcal{O}(n^2)$ time.

4 Further Discussion

We conclude by examining the implications of this work for the open problem of whether PH = PSPACE and discussing several possible variants of AARMs and PAARMs.

PH $\stackrel{?}{=}$ PSPACE. It was shown in Theorem 5 that PH $\subseteq$ PAAL[$\log^* n$] $\subseteq$ PSPACE. This result suggests a potential approach to proving PH $\neq$ PSPACE: one would have to show that PH is properly contained in PAAL[$\log^* n$] or that PAAL[$\log^* n$] is properly contained in PSPACE.

PAARM-Programs with Limited Alternation. One might be interested in PAARM-programs for which the number of alternations is constant but a player can do more than one automatic function step in a single turn. In particular, we

can show that if, after the booster step the number of alternation is just two, then every language in NLOGSPACE can be recognised in $\mathcal{O}(\log \log n)$ steps. After padding the input, Boris will run the nondeterministic logspace computation where he uses location number for the input cell read in a step. Boris can code this computation in the padded space, where each configuration step is of logarithmic size. Now Anke verifies the computation, which can be done in $\mathcal{O}(\log \log n)$ deterministic steps. The idea is to choose (guess) the two symbols from two successive logspace computation steps which contradict a Turing machine computation (that is, there are errors in the Turing machine computation). After that, Anke verifies that the positions of the error symbols are indeed correct by marking off every other symbol in each logspace sized protocol. The marking off of steps is repeated until all symbols except the errors in the logspace sized protocols are marked off, while maintaining the same parities of the errors' positions. It is easy to see that Anke needs $\mathcal{O}(\log \log n)$ marking off steps in total, and another additional step to confirm the error.

Bounded Automatic Relations vs Transducers. Instead of bounded automatic relations, one can also use transducers as basic operations. We define an Alternating Transducer Register Machine (ATL) and a Polynomial-Size Padded Alternating Transducer Register Machine (PATL) similarly to an AARM and a PAARM respectively. When using bounded automatic relations, the "bottleneck" of the computational power, as shown in this paper, lies in the string comparison, which needs $\mathcal{O}(\log^* n)$ steps by the Log-Star Lemma and its extension. This, however, does not apply when transducers are used as the string comparison can be done in constantly many transducer steps. Thus, we can show that not only an NP-complete problem is recognised by ATL in constantly many steps but also PH is contained in PATL in constantly many steps. Moreover, compared to bounded automatic relations, which can only increase the length of a given input by a constant in a single automatic step, transducers can increase its length up to k times for some constant k. Therefore, the booster step in a PATL can be simulated by an ATL in $\mathcal{O}(\log n)$ steps; thus $PH \subseteq ATL[\log n]$. For this reason, we believe that transducers are too powerful to be used as basic operations and that the use of bounded automatic relations as basic operations gives a more appropriate model of computation.

References

1. Achim Blumensath. Automatic structures. Diploma thesis, RWTH Aachen University, 1999.
2. Achim Blumensath and Erich GrÃďdel. Automatic structures. In *Proceedings of IEEE Symposium on Logic in Computer Science (LICS)*, page 51âĂŞ62, 2000.
3. Henning Bordihn, Henning Fernau, Markus Holzer, Vincenz Manca and Carlos Martín-Vide. Iterated sequential transducers as language generating devices. *Theoretical Computer Science*, 369:67–81, 2006.
4. John Case, Sanjay Jain, Samuel Seah and Frank Stephan. Automatic functions, linear time and learning. *Logical Methods in Computer Science*, 9(3), 2013.

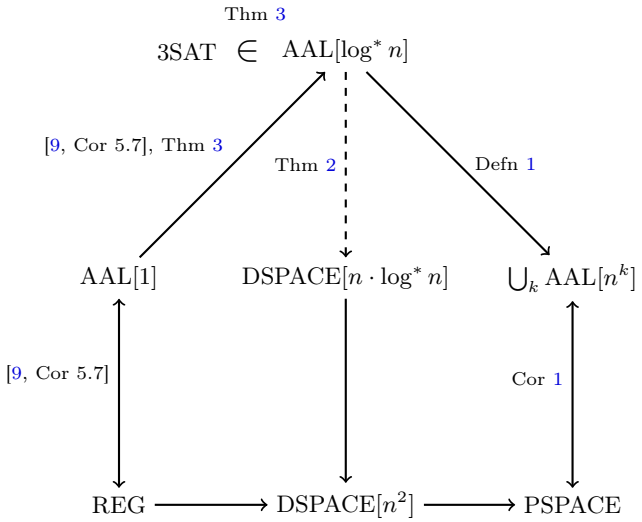

Fig. 1: Relationships between complexity classes/3SAT. A solid arrow from X to Y means that X is a proper subset of Y. If X is a subset in Y but it is not known whether they are equal sets, then the arrow is dashed.

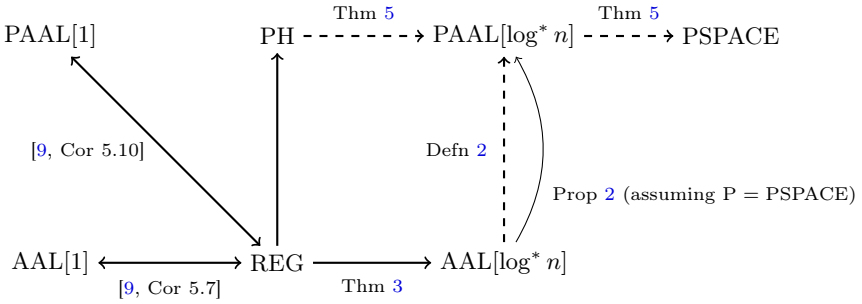

Fig. 2: Relationships between complexity classes. A solid arrow from X to Y means that X is a proper subset of Y. If X is a subset in Y but it is not known whether they are equal sets, then the arrow is dashed.

5. Ashok Chandra, Dexter Kozen and Larry Stockmeyer. Alternation. *Journal of the Association of Computing Machinery*, 28(1):114–133, 1981.
6. Ashok Chandra and Larry Stockmeyer. Alternation. In *17th Annual Symposium on Foundations of Computer Science (FOCS 1976)*, pages 98–108, 1976.
7. Robert Floyd and Donald Knuth. Addition machines. *SIAM Journal on Computing*, 19:329–340, 1990.
8. Martin Fürer. Alternation and the Ackermann case of the decision problem. *L'Enseignement MathÃľmatique*, II(XXVII):137–162, 1981.

9. Ziyuan Gao, Sanjay Jain, Zeyong Li, Ammar Fathin Sabili and Frank Stephan. Alternating Automatic Register Machines. arXiv:2111.04254 [cs.CC], 2021.
10. Ziyuan Gao, Sanjay Jain, Zeyong Li, Ammar Fathin Sabili and Frank Stephan. Models of computation with automatic functions as primitive steps. arXiv:2201.06836 [cs.CC], 2022.
11. Fred Hennie. One-tape, off-line Turing machine computations. *Information and Control*, 8:553–578, 1965.
12. Bernard Hodgson. Décidabilité par automate fini. *Annales des sciences mathématiques du Québec*, 7:39–57, 1983.
13. Bernard R. Hodgson. *ThÃľories dÃľcidables par automate fini*. PhD thesis, UniversitÃľ de MontrÃľal, 1976.
14. John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, Maaschusetts, 1979.
15. Bakhadyr Khoussainov and Mia Minnes. Three lectures on automatic structures. In *Proceedings of Logic Colloquium 2007*, volume 35 of *Lecture Notes in Logic*, pages 132–176, 2010.
16. Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. In *International Workshop on Logic and Computational Complexity*, volume 960, pages 367–392, 1995.
17. Martin Kutrib, Andreas Malcher, Carlo Mereghetti and Beatrice Palano. Deterministic and nondeterministic iterated uniform finite-state transducers: Computational and descriptional power. In *Beyond the Horizon of Computability*, volume 12098, pages 87–99, 2020.
18. Zeyong Li. Complexity of linear languages and its closures and exploring automatic functions as models of computation. Undergraduate Research Opportunities Programme (UROP) Project Report, National University of Singapore, 2018/2019.
19. Anthony W. Lin and Philipp RÃijmmer. Regular model checking revisited. Technical Report, 2020.
20. John C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the Association of Computing Machinery*, 10:217–255, 1963.
21. Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, third edition, 2013.
22. Frank Stephan. Automatic structures – recent results and open questions. In *Third International Conference on Science and Engineering in Mathematics, Chemistry and Physics*, volume 622/1 (Paper 012013) of *Journal of Physics: Conference Series*, 2015.
23. Frank Stephan. *Methods and Theory of Automata and Languages*. School of Computing, National University of Singapore, 2016.