Testability of Oracle Automata

(extended abstract)

Gaoyan Xie, Cheng Li, and Zhe Dang*

School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164, USA

Abstract. In this paper, we introduce oracle (finite) automata that are finite/Buchi automata augmented with oracles in some classes of formal languages. We present some testability results for the emptiness problem of oracle automata associated with various classes of oracles. Moreover, we show that some important verification problems (such as reachability, safety, LTL model-checking, etc.) for oracle automata can be reduced to testing the emptiness of oracle automata. Our theory results on the testability of oracle automata can find applications in the verification of a system containing unspecified/partially specified components.

1 Introduction

In automata theory, the emptiness problem is to decide whether an automaton accepts the empty language. Algorithmic solutions to the emptiness problem for various classes of automata have become a cornerstone for solving various problems in many areas of computer science, especially in model-checking [6] that seeks algorithmic procedures to check whether a system satisfies a given temporal property through state exploration. In traditional automata theory, the automaton in the emptiness problem must be *fully* specified. As shown later in this section, however, there is a need (motivated by model-checking problems for component-based systems) to investigate the emptiness problem for automata that are only partially specified. This latter emptiness problem, however, has not been well-studied.

In this paper, we study a form of partially specified automata, called *oracle finite automata* (OFAs), and their emptiness problem. An *oracle* is a language in a class $\mathcal{O}$ of languages. The name of "oracle" comes from the fact that we only know that the oracle is an element in the class but we do not know which one it is. However, one may obtain a truth value from a query "$w \in O$?" to the oracle O for a word w, where w is called a query string. An OFA is a finite automaton (FA) augmented with finitely many query tapes, each of which is unbounded, one-way, and writable. When the OFA is associated with an array of oracles in a language class, the OFA works exactly as the FA except when a write transition, a query transition, or a reset transition wrt some i-th query tape is fired. On executing the write transition, a symbol is appended to the end of the specified query tape. On executing the query transition, a truth value is returned after querying the i-th oracle with the content of the i-th query tape as the query string; the truth value depends on whether the query string is in the oracle. On executing the reset transition, the content of the i-th query tape is completely erased.

Obviously, for the OFA defined above, its emptiness problem can not be solved by simply looking at its transition graph, since results of queries to the oracles may affect the firability of a transition. Our approach to solving the problem is to compute a number, called a *query bound*, from the specification of the OFA and the language class $\mathcal{O}$ to which the oracles in the OFA belong such that testing query strings not longer than the query bound is sufficient to answer the emptiness problem. Once the query bound is computable, we say that the emptiness problem is solvable or, more accurately, testable. The main body of this paper focuses on establishing conditions on the language class $\mathcal{O}$ and on the OFA such that the emptiness problem is testable. We will study cases when $\mathcal{O}$ is the class of regular languages accepted by (non)deterministic finite automata with n states, when $\mathcal{O}$ is the class of context-free languages accepted by nondeterministic pushdown automata with n states, and when $\mathcal{O}$ is the class of commutative semilinear languages with characteristic n (defined in the paper). Most of our testability results also demonstrate respective query bounds explicitly.

* Corresponding author (zdang@eecs.wsu.edu)

Our studies on OFAs and their testability can find applications in model-checking. It is well-known that, in many cases, model-checking finite-state/infinite-state systems can be reduced to solving, for instance, the emptiness problem for various classes of automata. In these so called automata-theoretic approaches, a *goal automaton* is constructed from an instance of the model-checking problem in consideration such that the (language) emptiness of the automaton is equivalent to the answer to the model-checking query on the instance. Indeed, this is evidenced by classical work on LTL model-checking [20] as well as more recent work on model-checking restricted infinite-state systems, e.g., [5, 1, 13, 12]. Notice that the goal automata constructed in the existing automata-theoretic approaches are all fully specified.

However, a real-world verification problem may concern a system containing some unspecified components (i.e., they are not completely specified). For instance, a component-based system may involve some externally obtained components whose internal design specifications are not completely available to the system designers because of, e.g., patent or copyright reasons. On the other hand, a component is considered to be unspecified because that, even detailed specification for the component is given, algorithmic analysis for the component still may not be possible; e.g., components may have an infinite state space, components may contain analog circuits etc.. These unspecified components are usually treated as *black-box*es in the sense that their behaviors can only be determined by observing (i.e., testing) their input/out sequences. For systems with such unspecified components, their verification problems could not be solved using traditional automata-theoretic approaches, since the goal automata obtained by those the approaches would *not* be fully specified..

We claim that OFAs and their testability results can be used to solve some important verification problems for a class of systems with unspecified components. Consider a simple system $Sys = \langle A, X \rangle$ consisting of a specified component A and an unspecified component X. The specified component A keeps receiving messages from the outside environment and then transmits the message through the unspecified component X. The communications between A and X are synchronized via pairs of input/output symbols of X(communications in this example is one-way, i.e., always initiated by A). The unspecified component X has two input symbols $send$ and ack, and two output symbols yes and no. The transition graph of A is depicted in Figure 1, where we use a suffix ? to denote events from the outside environment (e.g., msg?), and use an infix / to denote communications of A with X (e.g., $send/yes$). Suppose that the property to be verified is as follows. Starting from s_0, state s_4 is not reachable on any path of A. Notice that the system reachability really depends on the communications between A and X. For instance, even though s_4 is reachable in the figure, it may not be so in reality since, e.g., X may never response with a no when receiving a $send$.

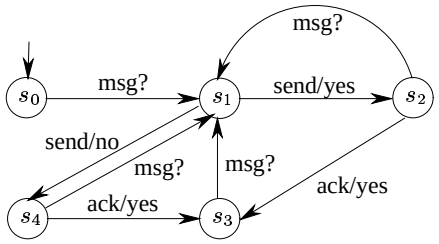

Fig. 1. A simple communication system

To address the verification problem for the system Sys, we construct an OFA M associated with an oracle O (that is the language of all the observable behaviors (the input/output sequences) of the unspecified component X). M works as follows. Starting from s_0, it nondeterministically follows the transition graph shown in the figure, without interacting with the unspecified component. Whenever a transition labeled with "$msg?$" is fired, M reads a symbol "msg" from its own input tape. Whenever a transition labeled with "$send/yes$" (resp. "$send/no$", "ack/yes") is fired, M writes a symbol "$send/yes$" (resp. "$send/no$", "ack/yes") on its query tape (there is only one query tape in M). When M enters state s_4, it accepts the word on its own input tape if a query to the oracle O with the current oracle tape content returns "yes". Clearly, M accepts an empty language iff the property holds for Sys. Notice that querying the oracle O with a query string w is equivalent to running a test-case w on the unspecified component X (i.e., see whether the test-case is a sequence of observable (input/output) behaviors of X. Suppose that a query bound can be

computed through some partial information about O using our testability results, then the property on Sys can be verified through black-box testing on the unspecified component X using test-cases not longer than the query bound.

Sometimes, a stronger testability result can be obtained when one restricts the behavior of an OFA or provides additional information on the oracles. We say that the OFA is positive if a no answer on a query always makes the OFA crash. Clearly, the OFA M in the above example is positive (and in fact *1-query* since only one query is necessary on acceptance). We say that the OFA is memoryless if a query transition on a query tape is immediately followed by a reset transition on the same query tape. A memoryless OFA is useful when, e.g., the unspecified component X runs in sessions. The end of a session is triggered by a special input symbol "reset" that brings X back to its internal initial state. Hence, two consecutive sessions of X are unrelated. An oracle could be *prefix-closed* in the sense that if a query string is in the oracle then all prefixes of the query string are all in the oracle. Testing the oracle can thus be sped up by this feature — once a query string w is not in the oracle, we have already known that any query string with w as a prefix is not in the oracle either. In the paper, we will also study the testability results for these restricted forms of OFAs. Moreover, later in the paper, we generalize the testability results concerning the emptiness problem to the LTL model-checking problem. This is interesting, since LTL concerns infinite behavior of an OFA. Testability of the LTL model-checking problem implies that testing query strings with a bounded length is enough to conclude some infinite behaviors of an OFA.

Oracle (Turing) machines are a classic concept in the theory of computation, and have been quite useful in studying, e.g., relativized complexity classes [2]. However, as far as we know, studying oracle finite automata in the context of model-checking is new. In particular, our technical approaches of doing model-checking through testing fit nicely into the current trend of integrating model-checking with testing [18]. The traditional work of black-box-testing [15] that checks conformance between two Mealy machines through input-out sequence testing is considered a different research problem from our work. The inference of query bounds is also loosely related to bounded model-checking that performs LTL model-checking through finite executions [4] as well as the bounding box techniques for some infinite state systems [21]. The oracle automata studied in this paper is also related to some existing work on model-checking with an incomplete model [3, 9]. But the most related work is probably from Peled, Vardi, and Yannakakis [19] who studied the problem of testing finite state systems with unknown structures against an LTL property. Their work, called Black-box Checking (BBC), is different from ours because in our work an unknown component (whose observable behavior could be irregular) is hooked up with a completely known finite state systems. Additionally, their proof techniques are completely different from ours. Kupferman and Vardi [14] investigated module checking by considering the problem of checking an open finite-state system under *all* possible environments. Module checking is different from our work in the sense that an oracle understood as an environment in [14] is a specific one.

The rest of the paper is organized as follows. Section 2 introduces basic definitions used in this paper. Section 3 starts with a formal definition of oracle finite automata and their testability. Then in Subsection 3.1, some results on the testability of the emptiness problem for various classes of oracle finite automata are presented. These results are further extended to oracle Buchi automata in Subsection 3.2. In Subsections 3.3, the connection between the testability results and some important verification problems of oracle finite/Buchi automata is established. Section 4 is a brief conclusion.

All of the proofs can be found in the Appendix, which may be read by the PC members at their discretion.

2 Preliminaries

Throughout this paper, Σ is any fixed alphabet. A *finite automaton* (FA) A consists of finitely many transitions, each of which makes the automaton move from one state to another while reading an input symbol (in Σ). In the description of A, we also designate an initial state and a number of accepting states. A sequence of input symbols or an input word $w \in \Sigma^*$ is *accepted* by A if, from the initial state of A, A reaches an accepting state after reading the entire word w. As usual, $L(A)$ stands for the language accepted by A. In general, a FA is nondeterministic; so we use DFA to denote a deterministic FA. A pushdown automaton (PDA) can be obtained by augmenting an FA with a pushdown stack (without loss of generality, we assume that the stack alphabet is the same as the input alphabet Σ and each time, the PDA pushes/pops at most one symbol). Similarly, DPDA is used to denote a deterministic PDA. We further use FA(n) (resp. DFA(n), PDA(n), DPDA(n)) to denote an FA (resp. DFA, PDA, DPDA) with at most $n \geq 1$ states. In

this paper, these notations of automata are also abused to represent languages accepted by the automata. For instance, FA(n) is the class of regular languages (on alphabet Σ) accepted by finite automata with at most n states.

Next, we recall the definitions of (semi)linear sets and their connection to counter machines. Let $\mathbf{N}$ be the set of nonnegative integers and $\Sigma = \{a_1, a_2, \ldots, a_k\}$ for some positive k. A subset S of $\mathbf{N}^k$ is a *linear set* if there exist vectors $v_0, v_1, \ldots, v_t$ in $\mathbf{N}^k$ such that $S = \{v \mid v = v_0 + b_1 v_1 + \ldots + b_t v_t,\ b_i \in \mathbf{N}\}$. The set $S \subseteq \mathbf{N}^k$ is *semilinear* if it is a finite union of linear sets. semilinear sets are precisely the sets definable by Presburger formulas [8]. For each word w in Σ^*, define the *Parikh map* of w to be $\psi(w) = (|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_k})$, where $|w|_{a_i}$ denotes the number of symbol a_i's in word w, $1 \leq i \leq k$. For a language $L \subseteq \Sigma^*$, the *Parikh map* of L is $\psi(L) = \{\psi(w) \mid w \in L\}$. The language L is *semilinear* if $\psi(L)$ is a semilinear set [17]. L is a *semilinear commutative* language if L is semilinear and, for all w_1, w_2 with $\psi(w_1) = \psi(w_2)$, $w_1 \in L$ iff $w_2 \in L$. That is, only the counts information $(|w|_{a_1}, |w|_{a_2}, \ldots, |w|_{a_k})$ is sufficient to decide whether $w \in L$. For instance, $\{w : |w|_a - |w|_b > 2|w|_c \wedge |w_a| < 5|w|_c\}$ is a commutative semilinear language over alphabet $\{a, b, c\}$.

Let c be a nonnegative integer. A c-counter machine is an FA augmented with c counters, each of which can be incremented by 1, decremented by 1, and tested for zero. We assume, w.l.o.g., that each counter can only store a nonnegative integer (since the sign can be stored in the states). Let r be a nonnegative integer and let NCM(c,r) denote the class of c-counter machines where each counter is r *reversal-bounded* [10]; i.e., each counter makes at most r alternations between nondecreasing and non-increasing modes in any computation. For instance, a counter whose values change according to the pattern 0 1 1 2 3 4 4 3 2 1 0 1 1 0 is 3-reversal, where the reversals are underlined. We use DCM(c,r) to denote the deterministic machines in NCM(c,r). From a result in [10], a semilinear commutative language L can be recognized by a DCM(c,r) M for some c and r if M's input is equipped with an end marker. In particular, it can be shown from [11] that there is a constant d such that, $L \neq \emptyset$ iff there is a word $|w| \leq d^{crm}$ in L, where m is the number of states in M. This result remains even when M is nondeterministic. From now on, we use M to characterize L and $\mathrm{LIN}(n)$ to denote those semilinear commutative languages that can be accepted by a DCM(c,r) with m states, where $n = d^{crm}$. With this definition, when $L \in \mathrm{LIN}(n)$ with $n \geq 1$, we say that L, as well as the M, has characteristic n. We use LIN to denote the class of all semilinear commutative languages.

3 Oracle Finite Automata and Testability

Recall that $\mathcal{O}$ is a class of languages over alphabet Σ and O, called an oracle, is a language in $\mathcal{O}$. Formally, M, an *oracle finite automaton* (OFA) with t oracles is a tuple

$$\langle t, \Sigma, S, R, s_{\mathrm{init}}, F \rangle, \tag{1}$$

where Σ is the given (input/query tape) alphabet, S is a finite set of *states* with s_{init} being the *initial state* and $F \subseteq S$ being a set of *accepting states*. R is a (finite) set of *transitions*, each of which is in one of the following five forms:

- (a read-input transition) $s \xrightarrow{a} s'$, which makes M move from state s to state s' after reading an input symbol a;
- (a write transition) $s \xrightarrow{\mathtt{write}(i,a)} s'$, which makes M move from state s to state s' after appending a symbol a to the end of the i-th query tape;
- (a positive query transition) $s \xrightarrow{\mathtt{query}(i)} s'$, which makes M move from state s to state s' when querying the i-th oracle (with the i-th query tape content as the query string) returns a "yes" answer;
- (a negative query transition) $s \xrightarrow{\neg\mathtt{query}(i)} s'$, which makes M move from state s to state s' when $\mathtt{query}(i)$ returns a "no" answer;
- (a reset transition) $s \xrightarrow{\mathtt{reset}(i)} s'$, which makes M move from state s to state s' and resets the i-th query tape content to be empty;

where $s, s' \in S$, $a \in \Sigma$, and $1 \leq i \leq t$. When $t = 1$, M is called a *single* OFA. Notice that the syntactical definition of M involves neither any description of $\mathcal{O}$ nor O. When M is associated with an array $O_1, \ldots, O_t$ of t oracles in $\mathcal{O}$, we use $M(O_1, \ldots, O_t)$ to denote the association. The semantics of M is defined as follows. Let $M(O_1, \ldots, O_t)$ be an association. A *configuration* is a tuple $\langle s, w_1, \ldots, w_t \rangle$ of a state s and t query tape contents $w_1, \ldots, w_t \in \Sigma^*$. The configuration is initial if the state is the initial state and the query tape contents are all empty.

The configuration is accepting if s is an accepting state. A one-step transition between two configurations is written as $\langle s, w_1, \ldots, w_t \rangle \xrightarrow{\alpha} \langle s', w'_1, \ldots, w'_t \rangle$ when one of the following conditions is satisfied:

- α is a, $s \xrightarrow{a} s'$ is a read-input transition in R, and each $w'_j = w_j$;
- α is $\mathtt{write}(i, a)$, $s \xrightarrow{\mathtt{write}(i,a)} s'$ is a write transition in R, and for each $j \neq i$, $w'_j = w_j$ and $w'_i = w_i a$;
- α is $\mathtt{query}(i)$, $s \xrightarrow{\mathtt{query}(i)} s'$ is a positive query transition in R, query string w_i is in O_i, and each $w'_j = w_j$;
- α is $\neg\mathtt{query}(i)$, $s \xrightarrow{\neg\mathtt{query}(i)} s'$ is a negative query transition in R, query string w_i is not in O_i, and each $w'_j = w_j$;
- α is $\mathtt{reset}(i)$, $s \xrightarrow{\mathtt{reset}(i)} s'$ is a reset transition in R and, for each $j \neq i$ $w'_j = w_j$, and $w'_i = \Lambda$ (the empty string).

A run of $M(O_1, \ldots, O_t)$ is a sequence

$$C_0 \xrightarrow{\alpha_1} C_1 \ldots C_{n-1} \xrightarrow{\alpha_n} C_n, \tag{2}$$

such that

- for each $j < n$, $C_{j-1} \xrightarrow{\alpha_j} C_j$ is a one-step transition, and,
- C_0 is the initial configuration.

The run is an accepting run if C_n is an accepting configuration. Let w be the result of deleting elements not in Σ from the sequence $\alpha_1 \ldots \alpha_n$. Then we say that the run in (2) is a run on input word w. A word w is *accepted* by $M(O_1, \ldots, O_t)$ if there is an accepting run on w. The language accepted by $M(O_1, \ldots, O_t)$, written

$$L(M(O_1, \ldots, O_t)),$$

is the set of all words accepted by $M(O_1, \ldots, O_t)$. Obviously, when associated with a different array of oracles, a query may return a different result and hence M may behave differently. Therefore, M can be thought of a template with t places to be filled in with oracles. To emphasize the fact that oracles are drawn from $\mathcal{O}$, we sometimes use $M^{\mathcal{O}}$ to denote the oracle finite automaton M and further use $M^{\mathcal{O}}(O_1, \ldots, O_t)$ to denote the specific association of the oracles $O_1, \ldots, O_t \in \mathcal{O}$ with M.

Various restrictions can be placed on query behaviors of an oracle finite automaton M. In this paper, we will focus on the following four forms of restrictions. M is a *prefix-closed* OFA if M is only associated with prefix-closed oracles[1]. M is a *k-query* OFA if, during any run, the oracles are queried for at most k times. M is a *positive* OFA if, in M, each query must return a "yes" answer (i.e., M does not have negative query transitions). M is a *memoryless* OFA if for each i, the i-th query tape content is erased (by a $\mathtt{reset}(i)$ transition) immediately after each query $\mathtt{query}(i)$. Therefore, during any run of a memoryless OFA, each query string sent to an oracle was "freshly written" since the previous query to the same oracle.

A *B-bounded testing script* $\mathcal{T}$ (with t oracles) is a deterministic Turing machine equipped with two tapes:

- the first tape, called the query tape, is a two-way readable and writable Turing tape whose length is B,
- the second tape, called the working tape, is an ordinary unbounded Turing tape,

and is further augmented with *query instructions*. Each query instruction allows the script to query an oracle with a query string that is the content of the portion of the query tape between the first cell and the current cell under the query tape head. A state transition is made upon the query result. We assume that $\mathcal{T}$ starts with both tapes blank and always halts, when associated with any array of t oracles. $\mathcal{T}$ is *successful* (resp. *unsuccessful*) on $O_1, \ldots, O_t$, if, when associated with $O_1, \ldots, O_t$, $\mathcal{T}$ halts with an accepting (resp. rejecting) state. The name of a "testing script" comes from the fact that, when $\mathcal{T}$ runs, the oracles are tested (queried) with query strings not longer than B. When $\mathcal{T}$ halts, the testing is finished and an answer of either "successful" or "unsuccessful" is given. Of course, the answer may be different when $\mathcal{T}$ is associated with another array of oracles.

A testing script is used to solve problems concerning an OFA. Let X be one of the language classes FA,DFA,PDA, DPDA, and LIN defined earlier. We use $\mathrm{OFA}^{\mathrm{X}}$ to denote the class of OFAs whose oracles are drawn only from X.

[1] A language on the alphabet is *prefix-closed* if the following condition is satisfied: for any word w, if w is in the language, then so is every prefix of w.

Let $M^{\mathrm{X}(n)}$ be one such automaton in $\mathrm{OFA}^{\mathrm{X}}$ with $n \geq 1$. Then a *problem* of M is a predicate over some oracles $O_1, \ldots, O_t$ in $\mathrm{X}(n)$. For instance, the *emptiness problem* of $M^{\mathrm{X}(n)}$ is to decide whether $M^{\mathrm{X}(n)}(O_1, \ldots, O_t)$ accepts an empty language. This problem can be characterized by the predicate $\mathcal{P}_{M^{\mathrm{X}(n)}}(O_1, \ldots, O_t)$, which is true iff $L(M^{\mathrm{X}(n)}(O_1, \ldots, O_t)) = \emptyset$. A problem $\mathcal{P}$ of $M^{\mathrm{X}(n)}$ is *testable* if there is an algorithm such that from the description of M and n, one can compute a number $B(M, n)$ and a $B(M, n)$-bounded testing script $\mathcal{T}$ satisfying the following condition: for each $O_1, \ldots, O_t \in \mathrm{X}(n)$, $\mathcal{P}(O_1, \ldots, O_t)$ is true (resp. false) iff $\mathcal{T}$ is successful (resp. unsuccessful) on $O_1, \ldots, O_t$. In this case, we also say that the $\mathcal{P}$ problem of the oracle finite automaton $M^{\mathrm{X}(n)}$ is $B(M, n)$-*testable*. That is, the $\mathcal{P}$ problem of $M^{\mathrm{X}(n)}$ can be decided by running the test script which queries the oracle with query strings not longer than $B(M, n)$.

3.1 Testing Emptiness for Oracle Finite Automata

How to figure out whether an oracle finite automaton M^{X} is testable for the emptiness problem? Notice that $M^{\mathrm{X}(n)}$, when associated with oracles $O_1, \ldots, O_t$, runs on some input, during which the oracles are queried. On a specific run, one may record the maximal length of all query strings sent to the oracles. Assume that the maximal length is uniformly bounded by a number B, called a *query bound*, among all the possible input words, runs, and associations of oracles from $\mathrm{X}(n)$. Under this assumption, checking whether $L(M^{\mathrm{X}(n)}(O_1, \ldots, O_t) = \emptyset$ becomes easier. This is because, for the purpose of emptiness, one can make each oracle to be finite (the number of elements is bounded by $|\Sigma|^B$) by dropping any word longer than B from the oracle. In this way, the finite oracle can be "recovered" through a finite number of queries. Even though the assumption in general does not hold, one can effectively build an approximated version $M_B^{\mathrm{X}(n)}$ from $M^{\mathrm{X}(n)}$ that satisfies the assumption for any number B, by forcing $M^{\mathrm{X}(n)}$ to crash whenever it tries to query the oracle with a query string longer than B. Then we refine the definition of *query bound*: B is a *query bound* of $M^{\mathrm{X}(n)}$ if, for any $O_1, \ldots, O_t \in \mathrm{X}(n)$,

$$L(M^{\mathrm{X}(n)}(O_1, \ldots, O_t) = \emptyset \text{ iff } L(M_B^{\mathrm{X}(n)}(O_1, \ldots, O_t) = \emptyset.$$

Once the query bound is identified, a B-bounded testing script $\mathcal{T}$ can be easily constructed to answer the emptiness of $M^{\mathrm{X}(n)}$:

1. For each oracle, $\mathcal{T}$ enumerates each string not longer than B, queries the oracle with the string, and stores the result on the working tape;
2. On the working tape, $\mathcal{T}$ also constructs a finite automaton (without any oracles) that simulates $M_B^{\mathrm{X}(n)}$ where each query is answered by retrieving the stored results;
3. $\mathcal{T}$ returns "successful" or "unsuccessful" according whether the finite automaton accepts an empty language (this can be decided by running a standard algorithm on the finite automaton).

Hence, in proving that the class $\mathrm{OFA}^{\mathrm{X}}$ is testable, we only need to demonstrate that a computable query bound exists for every M^{X} in $\mathrm{OFA}^{\mathrm{X}}$. This is the fundamental approach we will use to study some testable classes of oracle finite automata.

Studies on black-box testing [15] have shown that the structure of a finite automata with n states can be completed recovered by test sequences with length not longer that a bound $BT(n)$. This result can be immediately used to establish the testability of OFAs with regular oracles in FA(n). However, there are reasons that new techniques are needed. First, for emptiness testing, one does not need to recover the complete information of the oracle, and a smaller query bound than BT(n) may exist. Second, as shown in below, complete information is not recoverable for some practically useful but irregular oracles, e.g., $\mathrm{PDA}(n)$. That is, $BT(n)$ is not computable (from n) for oracles in $\mathrm{PDA}(n)$. In other words, context-free languages are not black-box testable.

Theorem 1. *Context-free languages are not black-box testable.*

In the rest of this section, we will present some results concerning the testability of the emptiness problem for various classes of OFAs. Recall that an oracle finite automaton M is associated with an array of t oracles. Let $|M|$ denote the number of states in M and we start with the case when M's oracles are regular.

Theorem 2. *(a). The emptiness problem for oracle finite automata $M^{\mathrm{DFA}(n)}$ is $O(n^t \cdot |M|)$-testable. (b). The emptiness problem for oracle finite automata $M^{\mathrm{FA}(n)}$ is $O(2^{nt} \cdot |M|)$-testable.*

We now turn to the case when M's oracles are drawn from context-free languages. Unfortunately, the emptiness problem for M is testable only under some special conditions. The proof of Theorem 3(a) uses a reduction to the halting problem of two-counter machines. For the testable cases, Theorem 3(b) involves PDA constructions.

Theorem 3. *(a). The emptiness problem for oracle finite automata in $\mathrm{OFA}^{\mathrm{PDA}}$ is not testable. The result remains in each of the following restricted cases:*

(a.1) the automata are 1-query and single,
(a.2) the automata are 2-query, positive, and single,
(a.3) the automata are 2-query, positive, and in $\mathrm{OFA}^{\mathrm{DPDA}}$.

(b). The emptiness problem for oracle finite automata $M^{\mathrm{PDA}(n)}$ is $2^{O(|M|^2 \cdot n^2 \cdot |\Sigma|)}$-testable, under each of the following conditions:

(b.1) $M^{\mathrm{PDA}(n)}$ is positive, single, and prefix-closed.
(b.2) $M^{\mathrm{PDA}(n)}$ is positive, single, and 1-query.
(b.3) $M^{\mathrm{PDA}(n)}$ is positive, single, and memoryless.
(b.4) $M^{\mathrm{PDA}(n)}$ is $M^{\mathrm{DPDA}(n)}$ and single.

Finally, we consider the case when M's oracles are drawn from semilinear commutative languages. As we are going to show, even though, in general, OFAs with such oracles are in general not testable for emptiness, under some restrictions, the problem becomes testable. The proof of Theorem 4(a) is a complex reduction to the halting problem of two-counter machines. In showing Theorem 4(b), properties over reversal-bounded NCMs are used.

Theorem 4. *(a). The emptiness problem for oracle finite automata in $\mathrm{OFA}^{\mathrm{LIN}}$ is not testable. The result remains in each of the following restricted cases:*

(a.1) the automata are single and positive,
(a.2) the automata are memoryless, positive, and have two oracles (i.e., $t = 2$).

(b). The emptiness problem for oracle finite automata $M^{\mathrm{LIN}(n)}$ is testable, under each of the following conditions:

(b.1) $M^{\mathrm{LIN}(n)}$ is k-query. In this case, it is $O(n^{k \cdot |M|^k})$-testable,
(b.2) $M^{\mathrm{LIN}(n)}$ is prefix-closed,
(b.3) $M^{\mathrm{LIN}(n)}$ is memoryless and single. In this case, it is $O(n^{|M|})$-testable.

3.2 Testing Emptiness for Oracle Buchi Automata

Syntactically, an *oracle Buchi automaton* (ω-OFA) M_ω is an oracle finite automaton M in (1). The difference is that, M_ω accepts only ω-runs (i.e., infinite runs). We write $M_\omega^{\mathcal{O}}$ for M_ω when its oracles are drawn from $\mathcal{O}$. Let $M_\omega^{\mathcal{O}}(O_1, \ldots, O_t)$ be an association of $M_\omega^{\mathcal{O}}$ with oracles $O_1, \ldots, O_t$ in $\mathcal{O}$. An *ω-run* of $M_\omega^{\mathcal{O}}(O_1, \ldots, O_t)$ is an infinite sequence

$$C_0 \stackrel{\alpha_1}{\to} C_1 \ldots C_{n-1} \stackrel{\alpha_n}{\to} C_n \ldots, \tag{3}$$

such that each prefix $C_0 \stackrel{\alpha_1}{\to} C_1 \ldots C_{n-1} \stackrel{\alpha_n}{\to} C_n$ is a run of $M^{\mathcal{O}}(O_1, \ldots, O_t)$, and for all m there is an $n > m$ with $\alpha_n \in \Sigma$. This latter requirement ensures that an ω-run reads an infinite number of input symbols. The ω-run is accepting if some accepting state in F appears infinitely often on the run. An ω-word τ is *accepted* by $M_\omega^{\mathcal{O}}(O_1, \ldots, O_t)$ if there is an accepting run in the form of (3) such that the word w_n (the result of deleting elements not in Σ from the sequence $\alpha_1 \ldots \alpha_n$) is a prefix of τ, for each n. We use $L^\omega(M_\omega^{\mathcal{O}}(O_1, \ldots, O_t))$ to denote the ω-language accepted by $M_\omega^{\mathcal{O}}(O_1, \ldots, O_t)$.

Completely analogous to oracle finite automata, we use ω-$\mathrm{OFA}^{\mathrm{X}}$ to denote the set of all ω-OFA $M_\omega^{\mathrm{X}(n)}$, for $\mathrm{X} \in \{\mathrm{FA}, \mathrm{DFA}, \mathrm{PDA}, \mathrm{DPDA}, \mathrm{LIN}\}$. We also follow a similar definition for prefix-closed, k-query, memoryless, and positive ω-OFAs.

The emptiness problem (for oracle Buchi automata) is to decide whether $M_\omega^{\mathrm{X}(n)}(O_1, \ldots, O_t)$ accepts an empty ω-language. For each class C of oracle finite automata considered in Theorems 3(a) and 4(a) whose emptiness is not testable, one can easily conclude that the emptiness problem for its corresponding class of oracle Buchi automata is not testable either. This is because from an oracle finite automaton M, one can build an oracle Buchi automaton M' as follows. M' behaves in the exactly same way as M, except that when M enters an accepting state, M' nondeterministically enters a special state (that is the only accepting state of M') and keeps staying in the state forever. Clearly, on associating with any oracles, M accepts an empty language iff M' accepts an empty ω-language. Notice that if M belongs to the class C of oracle finite automata mentioned earlier, M' belongs to the same class of oracle Buchi automata too. Therefore,

Theorem 5. *(a). The emptiness problem for oracle Buchi automata in ω-$\mathrm{OFA}^{\mathrm{PDA}}$ is not testable. The result remains even when each of the restrictions stated in Theorem 3(a) is applied.*

(b). The emptiness problem for oracle Buchi automata in ω-$\mathrm{OFA}^{\mathrm{LIN}}$ is not testable. The result remains even when each of the restrictions stated in Theorem 3(a) is applied.

By definition, when the emptiness problem for $M_\omega^{\mathrm{X}(n)}$ is testable, one can compute a $B(M, n)$-bounded testing script and, after running the script, the emptiness can be decided. The basic technique in showing testability is to reduce the emptiness problem of an oracle Buchi automaton in a class into the emptiness problem of an oracle finite automaton in the same class through loop analysis. Although loop analysis is a general technique, such a reduction does not always exist. For instance, currently, we do not know whether a positive, single and prefix-closed $M_\omega^{\mathrm{PDA}(n)}$ is testable or not for emptiness. However, according to Theorem 3 (b.1), a positive, single and prefix-closed $M'^{\mathrm{PDA}(n)}$ is testable for emptiness.

Theorem 6. *(a). The emptiness problem for oracle Buchi automata $M_\omega^{\mathrm{DFA}(n)}$ is $O(n^{2t} \cdot |M|)$-testable. The emptiness problem for oracle Buchi automata $M_\omega^{\mathrm{FA}(n)}$ is $O(2^{2nt} \cdot |M|)$-testable.*

(b). The emptiness problem for oracle Buchi automata $M_\omega^{\mathrm{PDA}(n)}$ is $2^{O(|M|^2 \cdot n^2 \cdot |\Sigma|)}$-testable, under each of the following conditions:

(b.1) $M_\omega^{\mathrm{PDA}(n)}$ is positive, single and 1-query.
(b.2) $M_\omega^{\mathrm{PDA}(n)}$ is positive, single and memoryless.
(b.3) $M_\omega^{\mathrm{PDA}(n)}$ is $M_\omega^{\mathrm{DPDA}(n)}$ and single.

(c). The emptiness problem for oracle Buchi automata $M_\omega^{\mathrm{LIN}(n)}$ is testable, under each of the following conditions:

(c.1) $M_\omega^{\mathrm{LIN}(n)}$ is k-query. In this case, it is $O(n^{k \cdot |M|^k})$-testable.
(c.2) $M_\omega^{\mathrm{LIN}(n)}$ is prefix-closed.
(c.3) $M_\omega^{\mathrm{LIN}(n)}$ is memoryless and single. In this case, it is $O(n^{|M|})$-testable.

3.3 Model-checking Through Testing

The emptiness problems we have investigated so far are closely related to some verification problems of oracle finite/Buchi automata. In this subsection, we will elaborate on this relationship.

Let $M^{\mathrm{X}(n)}(O_1, \ldots, O_t)$ be an OFA in a class $\mathrm{OFA}^{\mathrm{X}}$ associated with oracles $O_1, \ldots, O_t$ in $\mathrm{X}(n)$. The *reachability* problem is to decide whether there is a run of the OFA that ends up with a state in a given set Bad of states; i.e., Bad is reachable. (In practice, Bad specifies "bad" states that are not supposed to reach.)

Clearly, one may regard Bad as the accepting states in M and establish that Bad is not reachable iff

$$M^{\mathrm{X}(n)}(O_1, \ldots, O_t)$$

accepts an empty language. Let $|M|$ denote the state number in M, then we have,

Theorem 7. *The reachability problem for oracle finite automata $M^{\mathrm{X}(n)}$ is testable iff the emptiness problem for them is testable. In particular, if the emptiness problem is $B(|M|, n)$-testable, then so is the reachability problem.*

An input word accepted by the automaton is called an behavior. The *safety* problem is to decide whether every behavior of $M^{\mathrm{X}(n)}(O_1, \ldots, O_t)$ is contained in a given regular language R. We assume that the complement of R can be accepted by an FA with k states and $\bar{M}$ be the Cartesian product of the complement FA and M. Notice that $\bar{M}$ has $k \cdot |M|$ states. Clearly, the safety problem of $M^{\mathrm{X}(n)}$ is equivalent to the emptiness of $\bar{M}^{\mathrm{X}(n)}$. Hence,

Theorem 8. *The safety problem for oracle finite automata $M^{\mathrm{X}(n)}$ is testable iff the emptiness problem for them is testable. In particular, if the emptiness is $B(|M|, n)$-testable, then the safety wrt a regular language R is $B(k \cdot |M|, n)$-testable, where k is the state number of an FA accepting $\neg R$.*

Certainly, one can develop a result similar to Theorem 8 for oracle Buchi automata by requiring the property R to be "ω-regular" instead of "regular". However, concerning verification problems for automata on infinite words, LTL model-checking is a technique that has already been widely used. Therefore, it is also worthwhile to investigate the LTL model-checking problem for oracle Buchi automata.

The linear-time temporal logic (LTL) views the behaviors of a finite-state system as a set of paths, i.e., infinite words on alphabet Σ. LTL formulas are interpreted as sets of paths, which are defined as follows:

$$\phi ::= a \mid \neg\phi \mid \phi \wedge \phi \mid \circ\phi \mid \phi \mathbf{U} \phi,$$

where $a \in \Sigma$ is an *atomic proposition*. $\circ$ is the *next* operator, and $\mathbf{U}$ is the *until* operator. We interpret each atomic proposition a as the singleton set $\{a\}$. Intuitively, a path σ satisfies an atomic proposition a if the first symbol in σ is symbol a. A path σ satisfies $\circ\phi$ if σ^1 (by deleting the first symbol in σ) satisfies ϕ. σ satisfies $\phi \mathbf{U} \psi$ if there is a suffix σ^i (by deleting the first i symbols) of σ such that (1). the suffix satisfies ψ and (2). ϕ is consistently satisfied on each σ^j with $0 \leq j < i$. Notice that our treatment of atomic propositions here is essentially equivalent to a standard LTL definition [7] (though the appearance of ours is a little different). LTL is also capable of expressing many interesting properties of a reactive system. For instance, the property "the pump is on for infinitely many times" can be expressed as $\Box\Diamond\mathtt{pumpOn}$ (where $\Diamond\phi$ (eventually ϕ) is an abbreviation for $\mathtt{true}\ \mathbf{U}\ \phi$, and $\Box\phi$ (always ϕ) stands for $\neg\Diamond\neg\phi$.). We use $[f]$ to denote the set of ω-words that satisfy f. It is known that $[f]$ can be accepted by a Buchi automaton (an ω-OFA without the query tapes) with $O(2^{|f|})$ number of states, where $|f|$ is the length of f).

The *LTL model-checking* problem is to decide whether every ω-behavior of an oracle Buchi automaton

$$M_\omega^{\mathrm{X}(n)}(O_1, \ldots, O_t)$$

satisfies a given LTL formula f; i.e., every ω-word accepted by $M_\omega^{\mathrm{X}(n)}(O_1, \ldots, O_t)$ satisfies f. Similar to the standard LTL model-checking approach [20], we define $\bar{M}_\omega$ to be the Cartesian product of M_ω and the Buchi automaton that accepts $[\neg f]$. Clearly, M_ω is an oracle Buchi automaton with states $O(|M| \cdot 2^{|f|})$. Observe that the LTL model-checking problem is equivalent to check the emptiness of $\bar{M}_\omega^{\mathrm{X}(n)}(O_1, \ldots, O_t)$. Hence, we have the following result.

Theorem 9. *The LTL model-checking problem for oracle Buchi automata $M^{\mathrm{X}(n)}$ is testable iff the emptiness problem for them is testable. In particular, if the emptiness problem is $B(|M|, n)$-testable, then the LTL model-checking problem (wrt an LTL formula f) is $B(|M| \cdot 2^{|f|}, n)$-testable.*

Combining the three theorems obtained in this section with the testability results for the emptiness problem in the previous subsections, we immediately establish some testability results on verifying oracle finite/Buchi automata. For instance, the LTL model-checking problem is not testable for oracle Buchi automata in $\omega\text{-OFA}^{\mathrm{LIN}}$ (using Theorem 5). However, when memoryless and single oracle Buchi automata are considered, the problem becomes testable (using Theorem 6 (c)).

4 Conclusions

In this paper, we introduced oracle (finite) automata that are finite/Buchi automata augmented with oracles in some classes of formal languages. We presented some testability results for the emptiness problem of oracle automata associated with various classes of oracles. Moreover, we showed that some important verification problems (such as reachability, safety, LTL model-checking, etc.) for oracle automata can be reduced to testing the emptiness of oracle

automata. Our theory results on the testability of oracle automata can find applications in the verification of a system containing unspecified/partially specified components.

There are several possible directions for future work. For instance, the naive bounded testing script presented at the beginning of Section 3.1 may not be efficient. In a future paper, we will present a more efficient testing script using symbolic state exploration. On the other hand, one may also investigate oracle infinite-state automata instead of oracle finite automata. That is to study testability of designs with unbounded variables that interact with partially specified environments. One other possible work is to find ways to obtain a smaller query bound using, e.g., structural information of the transition graph of an oracle finite automaton. We will also look at the possibility of hooking up a real-world tester with a component-based design and performing model-checking through testing, using some of the algorithms in this paper. The (worst-case) query bounds obtained in the paper are large. However, the worst-cases may not show up in a specific application. In particular, even an internally complex environment may only have a very simple pattern of observable (by the design or the OFA) behavior, which will significantly bring down the query bounds.

References

1. P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.
2. J. L. Balcazar, J. Diaz, and J. Gabarro. *Structural Complexity II*. Springer-Verlag, 1990.
3. Sergey Berezin, Armin Biere, Edmund M. Clarke, and Yunshan Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In *Formal Methods in Computer-Aided Design*, pages 369–386, 1998.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.
5. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: application to model-checking. In *CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 1997.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.
7. E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 997–1072. Elsevier, 1990.
8. S. Ginsburg and E. Spanier. Semigroups, presburger formulas, and languages. *Pacific J. of Mathematics*, 16:285–296, 1966.
9. Patrice Godefroid, Robert S. Hanmer, and Lalita Jategaonkar Jagadeesan. Model checking without a model: an analysis of the heart-beat monitor of a telephone switch using verisoft. In *ISSTA'98*, pages 124–133. ACM Press, 1998.
10. O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25(1):116–133, January 1978.
11. O. H. Ibarra and J. Su. On the containment and equivalence of database queries with linear constraints. In *PODS'97*, pages 32–43.
12. O. Kupferman, N. Piterman, and M. Y. Vardi. Pushdown specifications. In *LPAR'02*, volume 2514 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002.
13. O. Kupferman and M. Y. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *CAV'00*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.
14. O. Kupferman and M.Y. Vardi. Module checking revisited. In *CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 1997.
15. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
16. M. Minsky. Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines. *Ann. of Math.*, 74:437–455, 1961.
17. R. Parikh. On context-free languages. *Journal of the ACM*, 13:570–581, 1966.
18. Doron Peled. Model checking and testing combined. In *ICALP'03*, volume 2719 of *Lecture Notes in Computer Science*, pages 47–63. Springer, 2003.
19. Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In *FORTE/PSTV'99*, pages 225–240. Kluwer, 1999.
20. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS'86*, pages 332–344. IEEE Computer Society, 1986.
21. G. Xie, C. Li, and Z. Dang. New Complexity Results for Some Linear Counting Problems Using Minimal Solutions to Linear Diophantine Equations. In *CIAA'03*, volume 2759 of *Lecture Notes in Computer Science*, pages 163–175. Springer, 2003.

Appendix: Proofs not presented in the paper

Proof of Theorem 1.

Proof. Assume that $BT(n)$ is computable. Let M_0 be a fixed PDA that accepts language Σ^*. Suppose that M_0 has n_0 states. Now, we are going to solve the following totalness problem:

Given: a PDA M,

Question: $L(M) = \Sigma^*$?

Assume that M has n states and without loss of generality, $n \geq n_0$ (otherwise one can add dummy states into M). From this n, one computes $BT(n)$ and makes sure that $w \in L(M)$ for each $w \in \Sigma^*$ that is not longer than $BT(n)$ (there are finitely many such w's). If this is true, then **Question** returns yes. Otherwise, **Question** returns no. According to the definition of BT, this indeed gives an algorithm to solve the totalness problem. This is a contradiction, since the totalness problem is known undecidable. The theorem follows. ∎

Proof of Theorem 2.

Proof. (a). Let $O_1, \ldots, O_t$ be any t oracles in $\mathrm{DFA}(n)$. For each $1 \leq i \leq t$, we use A_i to denote a $\mathrm{DFA}(n)$ that recognizes O_i. Now, we construct a finite automaton A that simulates $M^{\mathrm{DFA}(n)}(O_1, \ldots, O_t)$ as follows. A does not have an input. Whenever M reads an input symbol a, A guess an input symbol and makes sure that it is a. Whenever M writes a symbol a to the i-th query tape, A runs A_i on this symbol. Whenever M queries the current content of the i-th query tape, A answers the query by checking whether A_i is in an accepting state (notice that since A_i is a DFA, a negative query can be faithfully answered). Whenever M executes a transition that resets the i-th query tape, A brings A_i directly to A_i's initial state. In each case, A performs the same state transition as in M. Initially, A stays at the initial state of M and each A_i also stays at its own initial state. Clearly, $M^{\mathrm{DFA}(n)}(O_1, \ldots, O_t)$ accepts a nonempty language iff A has a run that ends with an accepting state of M. Moreover, the run can be further restricted to be not longer than the number $|A|$ of states in A. Notice that within this length of the run, M can not query an oracle with query strings longer than the length. Hence, the query bound is at most $|A|$, which can be easily calculated as $O(n^t \cdot |M|)$. Notice that this query bound does not depend on the particular choices of O_i's and hence A_i's. Therefore, the $M^{\mathrm{DFA}(n)}$ is $O(n^t \cdot |M|)$-testable.

(b). The result follows from the fact that a language in $\mathrm{FA}(n)$ is contained in $\mathrm{DFA}(2^n)$. ∎

Proof of Theorem 3.

Proof. (a.1). Let M be an OFA that is single (i.e., $t = 1$) and is associated with an oracle $O \in \mathrm{PDA}$. M first guesses a query string and writes it on the query tape. On a negative query result, M enters the accepting state. Clearly, M is 1-query, and, M accepts a nonempty language iff $O \neq \Sigma^*$. Now M is not testable since, otherwise, the totalness problem for context-free languages would become decidable.

(a.2). Let A be a deterministic two-counter machine where x and y are the counters and S is the set of states in A. Recall that a transition in A leads from s to s' while updating the counters (i.e., incrementing/decrementing the counters by 1 or testing for 0). We may use a string C

$$\#s\#1^x\#1^y\#$$

to denote a configuration of A where s is the state and x, y are the counter values. Notice that in the string, 1^x is the unary representation of value x (x number of 1's). For the same configuration, we use $\bar{C}$ to denote the *reverse configuration*

$$\$s\$1^y\$1^x\$.$$

A string is *even-valid* if it is in one of the following two forms for some m:

$$C_0\bar{C}_1 \ldots C_m\bar{C}_{m+1} \tag{4}$$

or

$$C_0\bar{C}_1 \ldots \bar{C}_mC_{m+1}, \tag{5}$$

such that each C_i is a configuration with C_0 being the initial configuration (with the state being the initial state and the counters being 0). Additionally, for each even $i \leq m$, $C_i \to C_{i+1}$; i.e., there is a transition in A that leads from C_i to C_{i+1}. The string is *odd-valid* if the additional condition is changed to be: for each odd $i \leq m$, $C_i \to C_{i+1}$. Now, we use L_{even} (resp. L_{odd}) to denote the set of even-valid (resp. odd-valid) strings. It is left to the reader to verify that both L_{even} and L_{odd} are in DPDA. Now, we define O to be the union of the following two languages: L_{even} and $\{w\# : w \in L_{\mathrm{odd}}\}$. Notice that, because of the additional suffix $\#$ appended after each odd-valid string, these two languages are disjoint. Obviously, O is in PDA. We now construct a single OFA M that works as follows. M keeps guessing a configuration and writes it to the query tape. We use C_i to denote the result of the i-th guess (i starts from 0). In fact, when i is odd, M writes the reverse configuration $\bar{C}_i$ instead of C_i to the tape. Nondeterministically, M decides to enter the accepting state. Before it does this, M first makes sure that the most recently written configuration is an accepting configuration of A (i.e., the configuration contains the accepting state of A). Then, M makes a positive query to O. On a yes answer, M writes an additional symbol $\#$ to the query tape, and performs one more positive query to O. A yes answer on this latter query leads M to accept. Notice that, on M's accepting, the content of the query tape forms a halting execution sequence of configurations of A. Therefore, M accepts a nonempty language iff A enters the accepting state (i.e., halts). From here, the result follows, since it is undecidable to decide whether a two counter machine halts [16], and, clearly, M is positive, 2-query, and single.

(a.3). Still, we let A be the two-counter machine in (a.2). We use O_1 (resp. O_2) to denote L_{even} (resp. L_{odd}). Notice that both O_1 and O_2 are in DPDA. Now, we construct an OFA M that has two query tapes and works similarly as in (a.2): M keeps guessing a configuration and writes it to both tapes (for the i-th guess with i being odd, a reverse configuration is written). Nondeterministically, M decides to enter the accepting state. To do this, M performs two positive queries: querying the first tape to oracle O_1 and querying the second tape to oracle O_2. At this time, M also makes sure that the most recently written configuration is an accepting configuration of A. Similar to (a.2), M accepts a nonempty language iff A halts. The result follows, noticing that M is 2-query, positive, and the oracles are from DPDA.

(b.1). Let O be any prefix-closed oracle in $\mathrm{PDA}(n)$. We use A' to denote a $\mathrm{PDA}(n)$ that accepts O. Now we construct a pushdown automaton A to simulate $M^{\mathrm{PDA}(n)}(O)$ as follows. Whenever M reads an input symbol a, A guess a symbol and makes sure that it is a. Whenever M writes a symbol a to the query tape, A runs A' on this symbol. Whenever M performs a query, A guesses and checks later one of the following two cases:

> CASE 1. the current query is the last query before the next reset transition or before M accepts. In this case, A makes sure that A' is in an accepting state (i.e., the query is positive),
>
> CASE 2. the current query is not the last query before the next reset transition or before M accepts. In this case, A assume that the query returns yes. This is valid, recalling that the oracle is prefix-closed.

Whenever M executes a reset transition, A brings A' back to its initial state (and also cleans up the stack). On each transition of M, A performs the same state transition, and additionally, A reads an input symbol $\mathtt{a}$ from A's input tape. Initially, A stays at the initial state of M and each A' also stays at its own initial state (with an empty stack). Notice that A is indeed a pushdown automaton that only accepts unary words in the form of $\mathtt{a}^B$ for some $B \geq 0$. Clearly, $M^{\mathrm{DFA}(n)}(O)$ accepts a nonempty language iff A does. If A accepts a nonempty language, then what is the length of the shortest word in the language? The length can be calculated as follows. The number of states in A is $O(|M| \cdot n)$. One may use a textbook technique to translate A into a context-free grammar G in Chomsky-Normal Form and to calculate the desired length, which is bounded by $2^{O(|M|^2 \cdot n^2 \cdot |\Sigma|)}$.[2] Notice that A accepts a unary word $\mathtt{a}^B$ iff M has an accepting run on some input word where the length of the run is exactly B. Obviously, during the run, M does not query the oracle with query strings longer than B. From here, we may conclude that $2^{O(|M|^2 \cdot n^2 \cdot |\Sigma|)}$ is the query bound for M. Since the bound is independent of the choice of O, the result follows.

(b.2). The proof of (b.1) still works here since M, being 1-query, need not worry about CASE 2 in the proof of (b.1).

(b.3). Similar to (b.2), one need not CASE 2 in the proof of (b.1).

[2] Recall that a PDA each time pushes/pops at most one symbol.

(b.4). Let O be any oracle in $\mathrm{DPDA}(n)$. We use A' to denote a $\mathrm{DPDA}(n)$ that accepts O. Now we construct a pushdown automaton A to simulate $M^{\mathrm{DPDA}(n)}(O)$. A works exactly as in (b.1) except when M performs a query. In this case, A obtains the query result by inspecting whether the deterministic A' is in an accepting state. The result follows after the exact query bound analysis that was done in (b.1). ∎

Proof of Theorem 4.

Proof. (a.1). Let A be a deterministic two-counter machine specified in the proof of Theorem 3 (a.2). That is, A has two counters x and y, and, we let $s_0, \ldots, s_m$, for some m, are the states in A. Without loss of generality, we assume that s_0 is the initial state and it is not an accepting state. In particular, we define, for each state s_i, $[s_i]$ to be the index i. For the purpose of describing the OFA to be built, we introduce an alphabet Σ that contains the following symbols:
θ^+, θ^-,
$\dot{\theta}^+, \dot{\theta}^-$,
a^+, a^-, b^+, b^-,
$\dot{a}^+, \dot{a}^-, \dot{b}^+, \dot{b}^-$.
A word w in Σ^* corresponds to a pair of configurations, called the pre-configuration C_w and the post-configuration C'_w, as follows. In the pre-configuration C_w,

- the state is s_i, where the index i is $|w|_{\theta^+} - |w|_{\theta^-}$ (recalling that $|w|_{\theta^+}$ is the number of symbols θ^+ appearing in w);
- the value for counter x is $|w|_{a^+} - |w|_{a^-}$;
- the value for counter y is $|w|_{b^+} - |w|_{b^-}$.

In the post-configuration C'_w,

- the state is $s_{i'}$, where the index i' is $|w|_{\dot{\theta}^+} - |w|_{\dot{\theta}^-}$;
- the value for counter x is $|w|_{\dot{a}^+} - |w|_{\dot{a}^-}$;
- the value for counter y is $|w|_{\dot{b}^+} - |w|_{\dot{b}^-}$.

Not every w will make two legal configurations; one has to further restrict that indexes i and i' are in the range of $0..m$, counter values in both of the configurations are nonnegative. We call this restriction as $\mathrm{R}_{\mathrm{conf}}$.

Each instruction in A is to increment/decrement a counter by 1 or test for 0. For example, with respect to counter x, an instruction can be in one of the following three forms:
(1) $s : x := x + 1$, goto s';
(2) $s : x := x - 1$, goto s';
(3) s : if $x = 0$ then goto s' else goto s''.
Instructions for counter y can be defined similarly. For each instruction I, one can formulate a restriction, called R_I, on w such that the pre-configuration C_w reaches the post-configuration C'_w after firing I. For instance, when I is in Form (1), we require that

- both C_w and C'_w are legal configurations; i.e., $\mathrm{R}_{\mathrm{conf}}$ is satisfied;
- the state in C_w is s; i.e., $|w|_{\theta^+} - |w|_{\theta^-} = [s]$. The state in C'_w is s'; i.e., $|w|_{\dot{\theta}^+} - |w|_{\dot{\theta}^-} = [s']$;
- the value for counter x in C'_w is equal to the value for counter x in C_w plus 1; i.e., $|w|_{a^+} - |w|_{a^-} + 1 = |w|_{\dot{a}^+} - |w|_{\dot{a}^-}$;
- the value for counter y does not change; i.e., $|w|_{b^+} - |w|_{b^-} = |w|_{\dot{b}^+} - |w|_{\dot{b}^-}$.

When I in the other forms, similar R_I can be defined.

Clearly, each R_I defines a semilinear commutative language over Σ. Let R be $\cup_{I \in \mathcal{I}} \mathrm{R}_I$, where $\mathcal{I}$ is the set of all instructions in A. R is also a semilinear commutative language.

Now, we are ready to build the single OFA M. M is associated with the semilinear commutative oracle R and works in rounds. We first sketch the ideas behind the following construction. At the beginning of each round, the content w of M's current query tape already encodes the pre-configuration C_w and the post-configuration C'_w such that $C_w \to C'_w$. That is, C_w reaches C'_w by firing some instruction in A; i.e. $w \in \mathrm{R}$. The job of the round is to change the tape content from w to w'. The new content w' also encodes $C_{w'}$ and $C'_{w'}$ such that

– $C_{w'} \to C'_{w'}$ and,
– $C_{w'}$ is exactly C'_w.

The first item can be ensured by performing a positive query to the oracle R. The difficulty is how to ensure the second item, which essentially creates an execution chain of configurations in A. Fortunately, the difference between C_w and C'_w can be remembered by M and hence can be used to update C_w to C'_w. Below is the formal construction of M. There are four phases in each round. In the first phase of the r-th round (r starts from 1), M guesses a state s^r. Keep in mind that the states in C_w and C'_w are s^{r-1} and s^{r-2}, respectively, where w is the current tape content. The job of the second phase is to change the current tape content such that the states in the pre-configuration and the post-configuration encoded by the new content are s^r and s^{r-1}, respectively. Formally, in the second phase, the following activities are performed, assuming that s^0 and s^{-1} are defined to be s_0:

– if $[s^r] \geq [s^{r-1}]$, then M writes q number of symbols $\dot{\theta}^+$ to the query tape, where $q = [s^r] - [s^{r-1}]$;
– if $[s^r] < [s^{r-1}]$, then M writes q number of symbols $\dot{\theta}^-$ to the query tape, where $q = [s^{r-1}] - [s^r]$;
– if $[s^{r-1}] \geq [s^{r-2}]$, then M writes q number of symbols θ^+ to the query tape, where $q = [s^{r-1}] - [s^{r-2}]$;
– if $[s^{r-1}] < [s^{r-2}]$, then M writes q number of symbols θ^- to the query tape, where $q = [s^{r-2}] - [s^{r-1}]$.

An *update* Δ is a pair (Δ_x, Δ_y) where $\Delta_x, \Delta_y \in \{1, 0, -1\}$. It indicates that, after the update, the amount of the change to counter x (resp. y) is Δ_x (resp. Δ_y). The job of the third phase is to change the current tape content such that the new counter values in the post-configuration encoded by the new content are the result of a guessed update on the old counter values in the post-configuration encoded by the old content. Formally, in the third phase, M guesses an update Δ^r and does the following:

– if $\Delta^r_x = 1$, then M writes a symbol $\dot{a}^+$ to the query tape;
– if $\Delta^r_x = -1$, then M writes a symbol $\dot{a}^-$ to the query tape;
– if $\Delta^r_x = 0$, then M writes nothing.

The job of the fourth phase is to change the current tape content such that the new counter values in the pre-configuration encoded by the new content are the result of the update performed in the last round on the old counter values in the pre-configuration encoded by the old content. Formally, in the fourth phase, (we define $\Delta^0 = (0, 0)$)

– if $\Delta^{r-1}_x = 1$, then M writes a symbol a^+ to the query tape;
– if $\Delta^{r-1}_x = -1$, then M writes a symbol a^- to the query tape;
– if $\Delta^{r-1}_x = 0$, then M writes nothing.

At the end of the fourth phase, M makes a positive query to the oracle with the current tape content and then starts a new round. Nondeterministically at the end of some round, M guesses that A halts. M accepts after making sure that the state s^r guessed in the round is the accepting state of A.

Since A is deterministic, for any word w, $w \in \mathrm{R}$ implies that there is a unique $I \in \mathcal{I}$ satisfying $w \in \mathrm{R}_I$. From this property, it is not hard to show that M accepts a nonempty language iff A has a halting execution; i.e., A halts. The result follows.

(a.2). Still, let A be a deterministic two-counter machine specified in the proof of (a.1). Similar to what we have mentioned in the proof of Theorem 3 (a.2), a configuration C of A can be specified as a string, denoted by $[C]$,

$$\$\theta^i\$a^x\$b^y\$.$$

In above, θ^i is to encode the state s_i, $1 \leq i \leq m$, and, a^x and b^y are for the counter values. Additionally, we may use the following string, denoted by $\langle C \rangle$,

$$\#\dot{\theta}^i\#\dot{a}^x\#\dot{b}^y\#$$

to represent the same configuration C. Similar to what we have in (a.1), one can construct from A a semilinear commutative language R (over alphabet $\{\$, \#, \theta, \dot{\theta}, a, \dot{a}, b, \dot{b}\}$) such that, for any two configurations C and C', the string $[C]\langle C' \rangle \in \mathrm{R}$ iff $C \to C'$.

We now construct an OFA M associated with two oracles (both are R) to simulate A. Initially, M writes the initial configuration of A to the first query tape, in the form of $[C_0]$. Then, M works in rounds. The r-th round (r starts from 1) is to perform the following two items:

– M guesses a configuration C_r. Then M writes $\langle C_r \rangle$ to the first query tape. In parallel to this, M also writes $[C_r]$ to the second query tape.
– M performs a positive query with the content of the first query tape and right after this, M erases the first tape (so M is memoryless).

The above description only works when r is odd. In the case when r is even, one needs to replace "first" with "second" (and vice versa) in the description of the two items. Nondeterministically at the end of some r-th round, M guesses that it is the time to accept. Then M makes sure that the state encoded in C_r is the accepting state of A. Clearly, M accepts a nonempty language iff A has a halting execution; i.e., A halts. The result follows.

(b.1). Let M be a k-query $\mathrm{OFA}^{\mathrm{LIN}(n)}$. Without loss of generality, we assume that M makes exactly k queries in an accepting run. Also, we assume that the queries are made to oracles $O_1, \ldots, O_k$, respectively. Let $A_1, \ldots, A_k$ be reversal-bounded DCMs (whose input has an end marker) with characteristic n and recognizing $O_1, \ldots, O_k$, respectively. We now build another reversal-bounded NCM A to simulate M. A starts with the initial state of M and simulates M's moves. When M reads the input tape, A does nothing to its own input. When M write a symbol to a blank query tape, A makes a guess on one of the following two cases:

– there is a query to oracle O_i that will be performed on the tape before the next reset (if any) happens. In this case, A starts running A_i on every symbol that is written on the tape subsequently until M indeed queries. For each such write, A reads a symbol a from its own input tape. At the time of querying, checking whether A_i enters an accepting state gives the query answer.
– there will not be a query to oracle O_i that will be performed on the tape before the next reset (if any) happens. In this case, A does nothing on every write to this tape until the tape is reset.

Notice that, on every move of M, A faithfully simulates M's state transitions. A accepts when M enters an accepting state. Clearly, A accepts a nonempty language iff M does. In particular, A only accepts a unary language. A word a^B is accepted by A iff M has a successful run on some input word where query strings are not longer than B. Since A is an NCM, to estimate B, it is sufficient for us to calculate a characteristic for A, which is a product of $A_1, \ldots, A_k$, along with the finite-state transition graph of M. One can show that a characteristic of A, and hence a query bound for M, is $O(n^{k \cdot |M|^k})$.

(b.2). Suppose $\Sigma = \{a_1, \ldots, a_k\}$. Let O be a prefix closed language in $\mathrm{LIN}(n)$ and accepted by a reversal-bounded DCM A with characteristic n. Observe that, from the description of A, one can effectively compute a finite number of "corner points"

$$(x_1, \ldots, x_k)$$

such that each x_i is in $\{0, \ldots, \infty\}$, and O is the union of all $\{w : |w|_{a_i} \leq x_i, 1 \leq i \leq k\}$. This gives the fact that O is regular. Hence, the result follows from Theorem 2. However, since we currently are unable to give a good estimation of the sizes for the corner points, the exact query bound for (b.2) is unknown.

(b.3). Since M is memoryless and single, M resets the query tape after each query. Now, we define another M' that is exactly as M but starts from a state s (in M) with blank query tape and ended with a reset right after a query (this is the only query that M' performs). Clearly, the maximal query bound for this M' (among all s) governs the desired query bound for M. Notice that M' is 1-query, the result follows from (b.1). ∎

Proof of Theorem 6.

Proof. (a). We need only to prove the first part of (a). Let $O_1, \ldots, O_t$ be any t oracles in $\mathrm{DFA}(n)$. For each $1 \leq i \leq t$, we use A_i to denote a $\mathrm{DFA}(n)$ that recognizes O_i. Now we construct an FA A that simulates $M^{\mathrm{DFA}(n)}(O_1, \ldots, O_t)$. Let s be any fixed state of M. A works almost the same as the A in the proof of Theorem 2 (a). The difference is that, during A's simulation on M, A nondeterministically remembers point when M is at state s. Then, A continues the simulation and makes sure that, after the point, M has read at least one input symbol and has passed the accepting state for at least once. At this time, A still continues the simulation and, nondeterministically, it guesses that it is time to accept. At this moment, it makes sure that the current states of M and all A_i's are exactly the same as those at the remembered point. It is not hard to show that A accepts a nonempty language for some s iff $M_\omega^{\mathrm{DFA}(n)}(O_1, \ldots, O_t)$ does. The result follows immediately, since A has $O(n^{2t} \cdot |M|)$ states.

(b.1). Since $M_\omega^{\mathrm{PDA}(n)}$ is positive, single and 1-query, on an accepting ω-run, M_ω does not perform any queries after certain point when an accepting state is reached. Before the point, M_ω behaves like the 1-query OFA $M^{\mathrm{PDA}(n)}$. After the point, M_ω behaves like a Buchi automaton (without accessing to the oracle). Hence, it suffices to consider the query bound for testing the emptiness of the 1-query OFA, shown in Theorem 3 (b.2).

(b.2). Without loss of generality, we assume that an accepting ω-run of M_ω queries the oracle for infinitely many times. One can also show that, on the run, there are two points such that at both points M_ω is at the same state and is right after a reset (resulting from a query since M_ω is memoryless). Furthermore, in between these two points, the run passes an accepting state and consumes at least one input symbol. In fact, the existence of the two points is the iff-condition on whether the ω-run is an accepting run. Checking the existence can be reduced to the case of Theorem 3 (b.3). The result follows.

(b.3) Let O be an oracle in $\mathrm{DPDA}(n)$ and M_ω be associated with O. On an accepting ω-run of the M_ω, there are two cases to consider:

Case 1. there are infinite number of reset transitions on the ω-run. Recall that each reset makes the query tape blank. The existence of such an ω-run can be fully decided by answering the following question for each pair of states s and s' in M: Can M start from state s with blank query tape and end with state s' also with blank query tape during which at least one input symbol is read and an accepting state is passed? The questions can be answered with a query bound

$$2^{O(|M|^2 \cdot n^2 \cdot |\Sigma|)} \tag{6}$$

using Theorem 3 (b.4).

Case 2. there are a finite number of reset transitions on the ω-run. The ω-run can be split into two parts. The first part is from the initial state s_0 to a state, say s, right after the last reset transition. The second part, starting from s and with a blank query tape, is the suffix of the ω-run right after the last reset transition. The existence of the first part is testable shown in (6) using Theorem 3 (b.4). Notice that M_ω does not reset on the second part, denoted by τ. We further assume that on τ, M_ω writes and queries infinitely many times on the query tape. Otherwise, it is easy to show that the existence of τ is testable in (6). Without loss of generality, we let s be s_0. With these assumptions, τ is essentially an accepting ω-run of M_ω (with oracle O) on which the query tape grows to infinity and an accepting state, say s_f, repeats infinitely often. As the result of τ, we use α to denote the ω-word that occupies the query tape. Let A be a $\mathrm{DPDA}(n)$ that accepts O. Since A is deterministic, we may run A along with the infinitely many write transitions performed during τ: a query can be faithfully answered by looking at whether A is at its accepting state. One can also observe the stack behavior during this infinite run of A and pick infinitely many points on the run where the stack stays lowest (i.e., the stack height beyond the point is not lower). In particular, we have the following Property:

There must be two points p_1 and p_2 on τ such that all of the following conditions are satisfied:
- At the two points, the states of M are the same;
- From point p_1 to point p_2 on τ, M has passed s_f at least once, has read at least one input symbol, has queried the oracle for at least once, and has written at least one symbol on the query tape;
- The state (resp. top symbol of the stack) of A at point p_1 is the same as the state (resp. top symbol of the stack) of A at point p_2; (recalling that A runs along M)
- From point p_1 to point p_2, A does not pop the stack content underneath the top symbol at point p_1.

In fact, one can also show that the Property implies the existence of τ since the segment from p_1 to p_2 forms a loop. Therefore, testing the existence of τ is equivalent to testing the Property. In the Property, the query tape content up to point p_2 can be accepted by a PDA with $O(|M| \cdot n)$ states by composing A with M properly. Using the technique presenting in the proof of Theorem 3 (b.1), one can show that the Property as well as the existence of τ is testable shown in (6).

The result follows by combining Case 1 and Case 2.

(c.1). Since M_ω is k-query, after certain point on an ω-run, M does not perform queries anymore. The result follows easily from Theorem 4 (b.1).

(c.2). The result follows from a similar argument made in the proof of Theorem 4 (b.2) and then from Theorem 6 (a).

(c.3). Similar to (b.2) except that we use Theorem 4 (b.3) instead of Theorem 3 (b.3).

∎