BLACK BOX CHECKING

DORON PELED
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974, USA

MOSHE Y. VARDI[1]
Rice University
Department of Computer Science
Houston, TX 77005, USA

MIHALIS YANNAKAKIS
Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974, USA

Abstract Two main approaches are used for increasing the quality of systems: in *model checking*, one checks properties of a known design of a system; in *testing*, one usually checks whether a given implementation, whose internal structure is often unknown, conforms with an abstract design. We are interested in the combination of these techniques. Namely, we would like to be able to test whether an implementation with unknown structure satisfies some given properties. We propose and formalize this problem of *black box checking* and suggest several algorithms. Since the input to black box checking is not given initially, as is the case in the classical model of computation, but is learned through experiments, we propose a computational model based on games with incomplete information. We use this model to analyze the complexity of the problem. We also address the more practical question of finding an approach that can detect errors in the implementation before completing an exhaustive search.

Keywords: Formal methods, Model checking, Specification, Testing, Verification.

1 INTRODUCTION

Model checking [5] and *testing* [16] are two complementary approaches for enhancing the reliability of systems. Model checking usually deals with checking whether the *design* of a finite state system satisfies some *properties* (e.g., mutual exclusion or responsiveness). On the other hand, testing is usually applied to the *actual system*, often

[1] Supported in part by NSF grants CCR-9628400 and CCR-9700061, and by a grant from the Intel Corporation. Part of this work was done when this author was a Varon Visiting Professor at the Weizmann Institute of Science.

without having access to, or knowledge of its internal structure. It checks whether a *system* (or an *implementation*) *conforms* with some design (i.e., informally, has the same behaviors). Even if access to the internal structure of the tested system is possible, it is not always a good idea to use it when performing tests, as this may lead to a bias in the testing process. Furthermore, the whole system may be very large (e.g., millions of lines of code), while we are interested only in specific aspects of it; think for example of a typical telephony switch and suppose we want to check that the implementation of a particular feature, such as call waiting, meets certain correctness properties. Extracting the part of the code that is relevant from the whole system, especially in the case of large legacy systems, is most probably infeasible (and is itself subject to errors). Suppose one is interested in checking specific properties of some system such as a communication switch or protocol system. Model checking would be appropriate for checking properties of a model of the system, but not checking the system itself. On the other hand, testing methods can compare the system with some abstract design, but usually are not used for checking specific properties.

One motivation for the current work is the case where acceptance tests need to be performed by a user who does not have access to the design, nor to the internal structure of the checked system. Our aim is thus to combine the two approaches, hence checking automatically *properties* of finite state systems whose *structure is unknown*. Of course, a completely hidden structure cannot be effectively checked. Thus, the following properties are assumed:

- A bound n on the number of states of the checked system is known.
- The tester can always **reset** the system to its (unique) initial state.
- The input alphabet Σ of the checked system is known.
- An experiment consists of repeatedly applying an input from Σ or a **reset** to the current state. An indication of whether the input was possible (enabled) from the current state is available.
- If an input α was possible from the current state, the system makes the move. Otherwise, it stays in the current state. No backtracking is available (but the tester can simulate backtracking by resetting and repeating the successful prefix of the experiment).
- The checked system is *deterministic* in the sense that from each state it can move with any given input to at most one successor state.

We do not assume that the size of the system is known precisely: n is only an upper bound. In particular, we would like to study the effect of the possibility that the bound on the number of states n may be much bigger than the actual number of states. This is the case when the number of states is only estimated. In practice, the system that is being checked may be large and have multiple functions, while the property may concern a specific aspect of the system. Although the system as a whole may be quite big, large parts of it may be irrelevant to the property, and the system may be equivalent to a much smaller finite state machine as far as checking the property is concerned. In this case, n should be taken to be an estimate on the logical complexity (the control structure) of the system with respect to the property at hand. Our methods

can be used also if no bound n is available, by running the algorithms to the extent that the available time and space resources allow; the guarantees in this case depend on the time spent.

Following the automata-theoretic approach to model-checking [12, 23], the (negation of the) checked property is directly given as, or translated into, a finite automaton on infinite words, usually a Büchi automaton [3]. Then, both the system and (the complement of) the checked property are represented using automata. An example of a system that is based on such principles is Spin [8, 10], where the specification is given by an automaton called a *never claim* that recognizes the *bad* (or *disallowed*) computations. In order to check whether the system under consideration satisfies the checked property, we intersect the automaton representing the system with the automaton representing the disallowed computations. Any sequence in the intersection is a counterexample for the checked property, while the absence of any counterexample means that the property is satisfied.

The problem we study here is a variant of the above model checking problem. We are given the automaton that represents the computations not allowed by the checked property. But the internal structure of the checked system is not revealed, and only some experiments, as described above, are allowed on it. Still, we want to check whether the system satisfies the given property. We call this problem *black box checking*. To simplify the discussion, we will not deal here with machines with output. Their treatment and the results are similar to the ones presented here. We will present on-the-fly algorithms that are aimed at quickly detecting errors in a checked system.

The choice of an appropriate computational model is central to the issue of black box checking. Unlike standard decision problems, the input is not given here at the beginning of the computation, but is learned through a sequence of experiments. We propose a computational model based on games with incomplete information, and use this model to analyze the complexity of the problem.

Our methods combine techniques from model checking, conformance testing and learning theory. All three areas have been actively pursued for a number of years and there is an extensive body of literature. Model checking has been a vibrant area of research for more than 15 years with the development of the theory and a number of software tools. Most tools check properties of finite state models expressed in some formal notation. One tool that is directed at the checking of software systems without a model is VeriSoft [7]: it is aimed at checking state invariants (assertions) of communicating processes, using partial order reduction methods for space exploration. For a recent book on model checking see [5].

The study of testing black box automata was initiated in Moore's classical paper from 1956 [15], where he defined and studied several problems including the machine identification problem (infer the state transition diagram of an unknown black box automaton). He also posed the fault detection or conformance testing problem (checking that the black box conforms to a specified design automaton). This problem has been studied in the subsequent years by many researchers, obtaining good bounds on the lengths of the tests needed, as well as efficient algorithms that check for conformance for different types of automata (machines with a distinguishing sequence or with reset, or in general without) [4, 9, 24, 25]. In the last 15 years, there has been a lot of work

on conformance testing in the protocols community, with a large number of papers, many of them based on the black box automaton testing models and methods. Early surveys of the work in the 50's and 60's can be found in e.g., [11, 22], and surveys of the more recent results and related work on protocol testing can be found in [13, 20]. Finally, there is substantial work in the learning community on the problem of learning finite automata (i.e. machine identification) with the help of a teacher. Efficient algorithms for learning different types of automata in this framework have been developed in [2, 14, 19].

2 PRELIMINARIES

AUTOMATA THEORETIC MODEL-CHECKING

A Büchi automaton is a quintuple $(S, S_0, \Sigma, \delta, F)$, where S is a finite set of states, $S_0 \subseteq S$ are the initial states, Σ is the finite alphabet, $\delta \subseteq S \times \Sigma \times S$ is the transition relation, and $F \subseteq S$ are the accepting states. A run over a word $a_1 a_2 \ldots \in \Sigma^\omega$ is an infinite sequence of states $s_1 s_2 s_3 \ldots$, with $s_1 \in S_0$, such that for each $i > 0$, $(s_i, \alpha_i, s_{i+1}) \in \delta$. A run is *accepting* if at least one accepting state occurs in it infinitely many times. A word is accepted by a Büchi automaton exactly when there exists a run accepting it. The *language* $\mathcal{L}(A)$ of a Büchi automaton A is the set of words that it accepts. Two automata are equivalent when they accept the same language.

An *implementation* automaton $B = (S^B, S_0^B, \Sigma, \delta^B, S^B)$ has several restrictions. We assume that the number of states $|S^B|$ is bounded by some value n, that S_0^B is a singleton $\{\iota\}$, and that $F^B = S^B$, namely, all the states are accepting.

We can view an implementation machine in our model as a Mealy machine: at each state v and for each input a, the machine outputs 0 if the transition is not enabled, and then remains in the same state, and 1 if it is enabled. Furthermore, we assume that the implementation automaton is deterministic, i.e., if $(s, a, t) \in \delta^B$ and $(s, a, t') \in \delta^B$, then $t = t'$.

For a *specification* automaton $P = (S^P, S_0^P, \Sigma, \delta^P, F^P)$, we will denote the number of states $|S^P|$ by m. Let the size of the alphabet Σ, common to the implementation and the specification, be p. As we mentioned in the introduction, we can easily extend the framework of this paper, and the results to implementation machines with arbitrary output (i.e., Mealy machines), and specification machines that describe the legal input-output behaviors.

The *intersection* (or *product*) $B \times P$ is $(S^B \times S^P, S_0^B \times S_0^P, \Sigma, \delta', S^B \times F^P)$, where

$$\delta' = \{((s, s'), \alpha, (t, t')) | (s, \alpha, t) \in \delta^B \wedge (s', \alpha, t') \in \delta^P\}.$$

Thus, the intersection contains (initial) states that are pairs of (initial, respectively) states of the individual automata. The transition relation relates such pairs following the two transition relations. The accepting states are pairs whose second component is an accepting state of P. We have that $\mathcal{L}(B \times P) = \mathcal{L}(B) \cap \mathcal{L}(P)$.

A **reset** is an additional symbol of S^B, not included in Σ, allowing a move from any state to the initial state. An *experiment* is a finite sequence $\alpha_1 \alpha_2 \ldots \alpha_{k-1} \in$

$(\Sigma \cup \{\textbf{reset}\})^*$, such that there exists a sequence of states $s_1 s_2 \ldots s_k$ of S^B, with $s_1 \in S_0^B$, and for each $1 \leq j < k$, either

1. $\alpha_j = \textbf{reset}$ and $s_{j+1} = \iota$ (a **reset** move), or
2. $(s_j, \alpha_j, s_{j+1}) \in \delta^B$ (an automaton move), or
3. there is no $t \in S^B$ such that $(s_j, \alpha_j, t) \in \delta^B$ and $s_{j+1} = s_j$ (a disabled move).

GAMES OF INCOMPLETE INFORMATION

The computation model for experiments on black box automata is not the standard one, in which the input is known from the beginning of the computation. Here, part of the input is hidden, and its structure is studied through experiments.

The relevant computational model is related to games of incomplete information [1, 18], where an $\exists$-*player* plays against a *deterministic environment* (representing a degenerate version of a $\forall$-*player*). Each such game consists of a nondeterministic machine with finitely many configurations[2] C, containing the following disjoint subsets: C_i are the *initial configurations*, W^+ and W^- are the positive and negative winning configurations for the $\exists$-player, respectively. Intuitively, since we want to check properties of systems, W^+ corresponds to finding an error, and W^- corresponds to concluding that there is no error.

Let $L_\exists$ and $L_\forall$ be sets of *labels* for the $\exists$-player and the environment, respectively. Then the sets of moves are $M_\exists \subseteq C \times L_\exists \times C$, and $M_\forall \in C \mapsto L_\forall \times C$, respectively. The $\exists$-player can have a choice of moves, thus $M_\exists$ is a relation, connecting the current configuration with all possible pairs of move-labels and resulted successor configurations. The moves of the environment $M_\forall$ are deterministic, and thus are defined as a function from the current configuration into the unique transition label and successor configuration. No move can originate in a winning configuration. Moreover, any two different moves from the same configuration must have different labels. The two players make moves in alternation, starting with the $\exists$-player, who makes the first move from an initial configuration. A *play* is a sequence from $(CL_\exists CL_\forall)^* C$, where each adjacent triple over $C(L_\exists \cup L_\forall)C$ conforms with a move of one of the players. A play is *winning* if it ends with a winning configuration in $W^+ \cup W^-$. There is no initial configurations starting both a play that ends with a configurations in W^+ and a play that ends with a configurations in W^-.

The incomplete information is stated by the partition of the configurations C into equivalence classes called *information sets*. The $\exists$-player cannot distinguish between configurations c_1 and c_2 that are in the same information set, denoted $c_1 \approx c_2$. Therefore, the move function $M_\exists$ must allow moves with the same labels for all the configurations that are in the same equivalence class. Furthermore, if $c_1 \approx c_2$ then $c_1 \in W^+$ ($c_1 \in W^-$, respectively) if and only if $c_2 \in W^+$ ($c_2 \in W^-$, respectively).

A *deterministic strategy* for the $\exists$-player is a function $st_\exists : C \times (L_\forall \cup \{\texttt{init}\}) \mapsto M_\exists$, such that

[2] Since the games that will be described later involve choosing an automaton and performing experiments on it, we choose to distinguish between a *configuration* of a game and a *state* of an automaton.

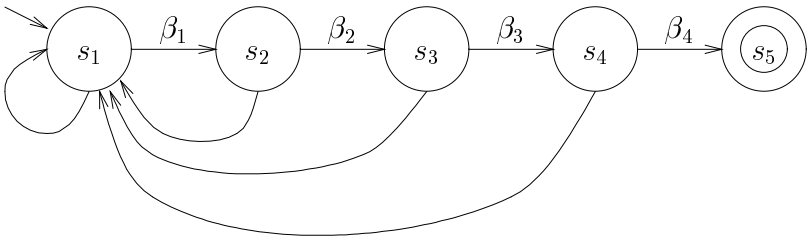

Figure 1: A combination lock automaton

1. If the $\exists$-player will keep playing $st_{\exists}(c, l)$ when it is his turn from configuration c and after the environment has played a move labeled with l, the sequence will end with a configuration in $W^+ \cup W^-$.
2. If $c \approx c'$, then the labels on $st_{\exists}(c, l)$ and $st_{\exists}(c', l)$ are the same.

The additional value `init` is paired with initial configurations from C_i (since there is no previous label for this configuration). A *path* played according to a strategy is an alternating sequence of configurations and labels, starting with an initial configuration. A *winning path* ends with a winning configuration. We will define the deterministic *time complexity* of a strategy as the length of the longest winning path in a strategy that ends with a configuration in $W^+ \cup W^-$.

We also define a *nondeterministic strategy* $nst_{\exists} : C \times (L_{\forall} \cup \{\texttt{init}\}) \mapsto M_{\exists}$ for the $\exists$-player. Let $c \in C_i$ be an arbitrary configuration such that there exists a play from c that ends with a configuration in W^+. Every play starting with c in which the $\exists$-player keeps playing his turn according to the $nst_{\exists}$ strategy will end with a configuration in W^+. The second constraint that was imposed on the deterministic strategy does not have a counterpart in the definition of the nondeterministic strategy. The intuition is that in the nondeterministic case, an $\exists$-player that is playing according to a nondeterministic strategy can make guesses that can distinguish between configurations that are in the same information set.

COMBINATION LOCK AUTOMATA

The following set of automata [15, 24] plays a major role in proving lower bounds on experiments with black box automata. A *combination lock* automaton [15] is a finite automaton such that there exists some complete order of the states $s_1, s_2, \ldots s_n$ with s_1 the initial state, and where the state s_n has no enabled transition. For each state s_i, $i < n$, there is a transition labeled with some $\beta_i \in \Sigma$ to s_{i+1}. For all other letters $\gamma \in \Sigma \setminus \{\beta_i\}$, there is a transition labeled with γ from s_i back to the initial state. It is called the combination automaton for $\beta_1\beta_2 \ldots \beta_{n-1}$. Figure 1 depicts a combination lock automaton for $n = 5$.

A sequence leading to a state without a successor (or even to a state where not all the letters are enabled) in a combination automaton must have a suffix of length $n - 1$

that is $\beta_1\beta_2\ldots\beta_{n-1}$. (This is a necessary, but not a sufficient condition. For example, the automaton in Figure 1 does not reach a deadlock state as a result of the sequence $\beta_1\beta_2\beta_1\beta_2\beta_3\beta_4$ when $\beta_1 \neq \beta_3$, since the second β_1 only causes it to return to its initial state.) Every path that does not contain the consecutive sequence $\beta_1\beta_2\ldots\beta_{n-1}$ is allowed (enabled) by the automaton.

3 BLACK BOX DEADLOCK DETECTION

In this section we describe a simpler problem related to black box checking. Given a deterministic finite state system B, with no more than n states, we want to check whether this machine deadlocks, namely reaches a state from which no input is possible.

In this problem, part of the model is unknown and is learned via experiments, which motivates modeling the problem as a game with incomplete information. We will also demonstrate that the deterministic and nondeterministic complexity do not have the same connections as in the standard model of computation that is based on Turing Machines.

For each implementation automaton with n or less states, there exists a single initial configuration. Each configuration in C in a play contains the same automaton as in the initial configuration, the current state of this automaton, as controlled by the moves of the $\exists$-player, and some information about the sequence of moves played so far. The current state of the automaton in an initial configuration is its initial state. The moves $M_\forall$ of the environment are labeled by `success` or `fail`. The label indicates whether the environment was successful or not in changing the state of the implementation automaton using the transition chosen according to the label of the last move of the $\exists$-player. The moves of the $\exists$-player are the possible input symbols, or a **reset** followed by a symbol (A **reset** is always successful.)

Projecting the labels of the moves of the $\exists$-player from a play ξ, we obtain an experiment over the implementation automaton in the initial configuration of ξ. If the configurations c_1 and c_2 are reachable using the prefixes of two plays ξ_1 and ξ_2 that correspond to the same experiment, then $c_1 \approx c_2$. The winning set W^+ contains only configurations that include an automaton that has a deadlock. Similarly, the winning set W^- contains only configurations that include an automaton without a deadlock.

A NONDETERMINISTIC STRATEGY

The $\exists$-player guesses in each move a label, forming a sequence of length smaller than n that brings the state of the selected machine from its initial state to some deadlock state. He then checks that this state has no enabled transitions.

Complexity: Nondeterministic time $O(n+p)$. The only information that is needed to be kept in each configuration is a counter from 0 to $n-1$ and a counter on the number of labels checked from the final state.

A DETERMINISTIC STRATEGY

The $\exists$-player checks systematically the possible sequences, up to length $n-1$, starting from the initial state. (Of course, there is no need to check sequences that include prefixes that have led to a failure.)

Complexity: Deterministic time $O(p^n)$.

Theorem 1 *The deterministic time complexity for black box deadlock detection is* $\Omega(p^{n-1})$.

Proof. Suppose that the initial configuration includes an implementation automaton B with n states that allows any input from any state. Consider a play ξ, played using a deterministic strategy for the $\exists$-player. Assume that ξ has less than p^{n-1} moves of the $\exists$-player, and terminates with a winning configuration c_w in W^-. Then at least one sequence $\beta_1\beta_2\ldots\beta_{n-1}$ does not appear consecutively in the experiment associated with ξ. If instead of the above automaton B, the environment would have chosen a combination lock automaton for $\beta_1\beta_2\ldots\beta_{n-1}$, the deterministic strategy would have resulted in a prefix of a play that has the same labels as ξ. Now we would have reached a configuration c' such that $c_w \approx c'$. Further, c_w is associated with an automaton without deadlocks, while c' is associated with an automaton with deadlocks. This contradicts the assumption that $c_w \in W^-$. ∎

In the standard complexity model, it is not known whether one can obtain a polynomial deterministic algorithm from a nondeterministic polynomial algorithm. Here, the (tight) lower deterministic bound is exponentially larger than the nondeterministic complexity. This justifies the use of games with incomplete information as an alternative for the common computational model of Turing machines.[3]

4 CHECKING PROPERTIES OF BLACK BOX FINITE STATE MACHINES

We address now the problem of black box checking. Namely, given a specification Büchi automaton P with m states, and a black box implementation automaton B with no more than n states, over a mutual alphabet Σ with p letters, we want to check if there is a sequence accepted by both P and B. The automaton P accepts the *bad* computations, i.e., those that are *not allowed*. Thus, if the property is given originally e.g., using a linear temporal logic (LTL) [17] property φ, then P is the automaton corresponding to $\neg\varphi$. For an efficient translation from LTL to automata, see e.g., [6]. The following simple theorem demonstrates that the current problem is at least exponential in time in the size of the automaton B.

Theorem 2 *The deterministic time complexity of black box checking is* $\Omega(p^{n-1})$.

[3] Another observation, connected with the *space complexity* of this model, which was not defined formally in this paper, is that the common space efficient strategy of binary search cannot be used here.

Proof. Similar to the proof of Theorem 1, we construct variants of a combination lock automata. The deadlock state is replaced now with a self loop labeled by γ. The symbol γ is disabled in the initial state. This removes at most half of the possible combinations (in the case where $p = 2$), so the complexity changes only by a constant factor. The property automaton P consists of two states: t_0 which is an initial state, and t_1 which is an accepting state. There is a self loop from t_0 to itself on each label from Σ, and from t_1 to itself on γ. There is also an edge labeled by γ from t_0 to t_1. Thus, the intersection is nonempty exactly if a state can be reached in the black box automaton, where γ^ω can be executed. The only such state of a combination lock black box is the state at the end of the path prescribed by the combination. ∎

4.1 AN OFF-LINE STRATEGY

A straightforward way to perform black box checking is to infer first the structure of the black box system, and then to apply model checking techniques to its newly revealed structure. The machine identification problem is a well studied problem. Typically, it is applied to automata that produce output, either at the states (Moore machines) or at the transitions (Mealy machines). As we mentioned, an implementation machine in our model can be viewed as a Mealy machine, where output 0 on a transition means that it is not enabled and output 1 means that it is enabled.[4]

It is well known that if two machines with n states are not equivalent, then there is an input of length at most $2n - 1$ that distinguishes them. This implies that any machine with at most n states is completely characterized by its output on all input strings of length $2n - 1$. That is, a black box is uniquely determined by applying all such p^{2n-1} input strings. A p-ary tree of depth $2n - 1$ can be constructed from the responses of the black box, and it can be minimized to produce the minimal machine M consistent with these outputs [22]. Then we can use model checking to check whether M satisfies the given property P. The length of the test sequence (or in terms of games with incomplete information, the length of the corresponding play), which gives us the time complexity, is $O(np^{2n-1})$. If implemented in the straightforward way, the space complexity is also exponential (to record the tree all the input strings and their output), but the minimization can be done incrementally in polynomial space. The time for the model checking is comparatively small, $O(pmn)$ where m is the size of the property automaton P (which is typically very small).

The complexity of this method is not that far off the lower bound, and in the worst case one may indeed need to identify in effect the black box machine in order to check a property. However, intuitively it is clear that in many cases this method can be wasteful in that it does not take advantage of the property to avoid doing a complete identification. For example, suppose that the property is that some error indication label γ never occurs. The property automaton P, representing the bad computations is

[4] By this convention, the output provides some partial information also on the next state, namely if the output is 0 then we know that the state does not change. This is not important for what follows (i.e. all the methods apply to any Mealy machine) but it can be used to do obvious optimizations on the tests. For example, if we apply input a and it is not enabled, then it is pointless to try again a until an enabled transition has been performed.

in this case a simple 2 state automaton. Obviously in this example, there is no reason to wait until we reconstruct the full black box automaton before we check the property. The sensible thing to do would be to check the assertion (i.e., try to see if γ is enabled in the current state) as we go along during the test, and if it gets violated at some point, then an error has been found and the check is complete.

In general, it would be obviously beneficial to use the property automaton on the fly to detect errors as early as possible and prune the test. Notably, if the estimate n on the number of states is much higher than the actual number of different states, or if it is accurate, but there is still a 'small' counterexample, i.e. there is a small set of states that exhibits the faulty behavior, we would like to be able to find the error without searching the whole space, if possible. This is not always as easy, especially in the case of properties that depend on the infinite behavior of the system, that is, in cases where the property automaton is a genuine Büchi automaton. We will investigate such methods in the following sections.

4.2 AN ON-THE-FLY STRATEGY

We will present now a strategy that can terminate quickly when the actual size of the automaton is much smaller than n, and an error is present (i.e., the intersection of B and P is nonempty).

A NONDETERMINISTIC STRATEGY

As before, we start with a nondeterministic version, in order to demonstrate the principle behind the on-the-fly black box checking.

According to the strategy, the $\exists$-player guesses a path σ of the automaton P, starting from an initial state, that can be partitioned into two subpaths σ_1 and σ_2, Each of which is of length smaller or equal to mn. Both subpaths end with the same accepting state t of P. Furthermore, the blackbox automaton must allow executing the transitions of $\sigma_1\,\sigma_2^{n+1}$ after a **reset**.

Correctness: Consider the unknown end state of the intersection automaton for each iteration of σ_2 in the experiment $\sigma_1\,\sigma_2^{n+1}$. Then at least one component state s of B must occur twice with the same accepting component state t of P (as there are no more than n states in the intersection that have the same component t). Thus, the path σ must include a cycle through an accepting state, which guarantees that an infinite accepting run exists in the intersection. Conversely, it is easy to see that if the intersection of B and P is nonempty, such a guess exists.

Complexity: Nondeterministic time $O(n^2m)$.

A DETERMINISTIC STRATEGY

The strategy finds a path as in the nondeterministic case, but in a systematic way. Now, a search for two paths of length bounded by mn is performed. The first path σ_1, when input to the automaton P needs to terminate with an accepting state t. The second path σ_2, when starting in state t of P, needs to terminate with t as well. For each such pair, we apply the second path n more times. That is, we try to execute the path $\sigma_1(\sigma_2)^{n+1}$.

If we succeed, this means that there is a cycle in the intersection through a state with a t (which is accepting) as the P component, since there are at most n ways to pair up t with a state of B. In this case, there is an infinite accepting path in the intersection.

Complexity: Deterministic time $O(n^2\, p^{2mn}\, m)$. This is because there are p^{2mn} choices of such paths. Each is of length bounded by mn, and we repeat it $n+1$ times.

The following comments should be noted:

- Unlike the off-line strategy, the complexity of this strategy depends on the number of states m of P. Typically however m is small, or even fixed, when talking about a fixed property.
- For properties that can be specified by automata on finite strings (i.e., depend essentially on finite computations), we need to search only for the first string σ_1 and the complexity is $O(n\, p^{mn}\, m)$.
- When searching for the strings σ_1, σ_2, we need only consider strings that can be extended to accepting strings of the property automaton. Furthermore, we can start by limiting the length of the subpaths that we explore and gradually increase that length as we proceed in the search. In this way, if the actual size of the the automaton B is much smaller than n, and an error occurs, it can be found much earlier than in the exhaustive strategy, as required above.

4.3 A STRATEGY BASED ON LEARNING AND TESTING

We show now that the factor m in the exponent can actually be removed. We provide below a strategy with complexity whose exponential term is $O(p^n)$. Furthermore, if the black box has an error, the time complexity will be exponential only in the actual size of the minimized version of the black box automaton.

In this algorithm, we will use an algorithm for conformance testing of a known finite automaton with a black box automaton by Vasilevskii and Chow [4, 24]. We will not repeat this algorithm here, but will use the result that its time complexity is $O(l^2\, n\, p^{n-l+1})$, where n is the assumed size of the black box automaton, $l \leq n$ is the actual size, and p is the alphabet size. Intuitively, the algorithm has to check the states and transition relation of the black box automaton, and that no error that follows a 'combination lock' occurs from any one of its node.

Another procedure that we use in our strategy is an algorithm for learning automata with **reset** using membership tests and questions to an oracle (a teacher) by Angluin [2]. In the learning algorithm, the teacher answers equivalence tests to a proposed machine and provides a counterexample in case of inequivalence. We will replace the teacher with experiments on the black box automaton. Starting from a trivial automaton, Angluin's algorithm generates successively larger candidate automata M_i, for $i = 1, 2, \ldots$ (the number of states in each conjectured automaton is monotonically increasing). It asks the teacher for equivalence. If equivalence does not hold, it uses a counterexample provided by the teacher, queries some more strings, and then generates the next conjectured automaton with more states, until it reaches the correct number of states. At this point the conjectured automaton is the correct one.

We modify Angluin's algorithm as follows. Our modification can use two kinds of counterexamples, provided by the teacher:

1. A *simple* counterexample of the form $\sigma \in \Sigma^*$, meaning that σ belongs to one of the checked automata, but not the other.
2. A pair of words $\sigma_1, \sigma_2 \in \Sigma^*$ such that $\sigma_1 \sigma_2^n$ belongs to one of the checked automata, but not the other.

We construct a sequence of automata $M_1, M_2, \ldots$ that attempt to converge into the black box automaton B. Membership queries are just experiments on the black box B. For equivalence queries, suppose we have a conjectured automaton M_i for the black box. First, we check if M_i generates a word accepted also by the specification automaton P, namely if $\mathcal{L}(M_i) \cap \mathcal{L}(P) \neq \emptyset$. If the intersection is not empty, it must contain an ultimately periodic word of the form $\sigma_1 \sigma_2^\omega$ [21]. We input **reset** $\sigma_1 \sigma_2^{n+1}$ to the black box B. If this experiment succeeds, then there is an error as $\mathcal{L}(B) \cap \mathcal{L}(P)$ contains $\sigma_1 \sigma_2^\omega$ and thus is not empty. If it fails, then this gives a counterexample for the equivalence of B with M_i. We use this in Angluin's algorithm to generate the next candidate automaton with more states.

If M_i does not generate any word accepted by P, we check whether M_i conforms with B. Let k be the number of states of M_i. We start the conformance test between M_i and B assuming B has k states and apply the Vasilevskii-Chow algorithm. If the conformance test fails, we use the counterexample in Angluin's learning algorithm to generate M_{i+1}. If the conformance test succeeds, we repeat it with $k+1, k+2, \ldots, n$. If n is reached, we declare that the black box satisfies the checked property.

This strategy is described in Figure 2. The procedure call $VC(M_i, k)$ calls the Vasilevskii and Chow algorithm for conformance testing M_i with the black box automaton B, assuming that B has no more than k states. VC returns (**true**,–) if the conformance test succeeds. If it fails, it returns (**fail**,σ), where σ is a word that is in one of the automata B or M_i but not the other. The procedures $ANGLUIN$ accepts the previous attempted automaton, and a counterexample, and returns a new attempted automaton. In the first call to $ANGLUIN$ in the strategy, it is executed with an empty automaton, and the second parameter (the counterexample) is ignored.

Complexity: Suppose that the minimum equivalent automaton M_i of the black box has l states. If the black box has an error, then the strategy will produce the automaton M_i in time $O(l^3 p^l)$, and declare the error. This is because some earlier conjectured automaton may have an empty intersection with P, in which case the strategy will need to do conformance testing until a string that distinguishes the conjectured automaton from B is found.

If the black box satisfies the property, then the strategy will generate M_i in time $O(l^3 p^l)$, but then it will have to spend $O(l^3 p^{n-l+1})$ more time to verify that the black box is indeed equivalent to M_i (by conformance checking M_i with B). It should be noted that the complexity of doing the exhaustive check for trying to distinguish the conjectured automaton from B is dominating over the complexity of the other tests prescribed by the above strategy. Moreover, since $O(p^n + p^{n-1}) = O(p^n)$, we need to consider only the last exhaustive check. Thus, we have the following theorem.

Theorem 3 *Black box checking, for a black box automaton B with l states, where l is unknown but is smaller than some bound n, and a property automaton P with m states can be done in time*

```
        M_1 := ANGLUIN(empty, −);
        i := 1;
learn:  X := M_i × P;
        if L(X) = ∅ then
             k := number of states of M_i
             loop
                  (conforms, σ) := VC(M_i, k);
                  k := k + 1
             until k > n or ¬conforms;
             if conforms then WIN(+);
        else
             let σ_1, σ_2 be s.t. σ_1 σ_2^ω ∈ L(X);
             if B allows reset σ_1 σ_2^n then WIN(-)
                  else σ := prefix of σ_1 σ_2^n not allowed by B;
        M_{i+1} := ANGLUIN(M_i, σ);
        i := i + 1;
        goto learn;
```

Figure 2: A strategy using learning

- $O(l^3\,p^l + l^2mn)$, *when there is an error (i.e., the intersection of* B *and* P *is nonempty), and*
- $O(l^3\,p^l + l^3\,p^{n-l+1} + l^2mn)$, *when there is no error.*

If we do not have a bound n on the number of states of the automaton B, we can run the algorithm as long as time permits. Consider first a property characterized by an automaton on finite strings (such as deadlock freedom and other safety properties). If we ever encounter an error, i.e. find a string σ accepted by both the black box and the property automaton P, then it is a true error and we can stop the test. If there is an error and B has size l (or the smallest counterexample has length l), we are sure to find the error within time $O(l^3\,p^l + l^2m)$. Conversely, if after the allocated time no error has been found, then this means that either the black box is correct, or else the smallest possible counterexample and the size of the black box must exceed a certain bound, which depends on the time spent on the test.

Suppose that we have a genuine Büchi automaton that depends on infinite behaviors. Suppose that at some point the conjectured black box automaton M_i has a nonempty intersection with the property automaton P, and let $\sigma_1\sigma_2^\omega$ be a string in the intersection. If the conjectured automaton M has l states at this point and P has m states, then the strings σ_1, σ_2 have length at most lm. We can input **reset**σ_1 to the black box followed by repeated applications of σ_2 until either the black box does not accept it or we run out of time. In the first case, we have found at least one new state and we continue the algorithm as before. In the second case, that is, if we run out of time after executing r repetitions of σ_2, then we can conclude that either there is an error, or the size of the implementation machine exceeds r.

Finally, if the conjectured automaton has an empty intersection with the property automaton then we perform conformance testing for increasingly larger values of the bound n on the black box. At the end we can place again a lower bound on the size of the black box or conclude that it is otherwise correct.

Let us comment finally on the exponential lower bound derived from the combination lock automata. Obviously these are rather pathological, worst case examples. The 'average' automata are much better behaved and do not exhibit this nasty performance bottleneck. This can be formalized by considering a probabilistic model for machines with output. Formally, there has been extensive work studying the properties of random machines [22]. The usual model of a random Mealy machine on l states is defined as follows. For each state and input symbol choose the next state and output uniformly at random. For the average machine, polynomial time will suffice to find an error. In the following statement, 'almost all machines' means that the probability tends to 1 as the size goes to infinity.

Theorem 4 *For almost all black box machines with l states, if an error is present, it will be found after a test of length $O(lp\log^2 l + l^2nm)$.*

This can be shown using the following two nice properties of almost all random machines: (1) if a state q can reach another state q' then it can reach it in $O(\log_p l)$ steps; (2) any two states can be distinguished by input strings of length $\log_p \log_q l$ [22]. Of course, if there is no error and we want to make sure that we do not have any other automaton at hand with at most n states, then we still would need to do the conformance testing (at a cost exponential in the difference $n-l$) in order to be certain of the correctness.

5 CONCLUSIONS

We defined the problem of black box checking, showed lower bounds and provided three strategies for solving the problem. The lower bound in Theorem 2 implies that the complexity of black box checking is exponential in the estimated size of the unknown automaton. For comparison, checking the emptiness of the intersection of the same automata (now both structures are given) is in NLOGSPACE-complete. In conformance testing, one checks whether a given known automaton P of length l is equivalent to a black box automaton B of length bounded by some $n \geq l$. Vasilevskii and Chow [24, 4] showed a lower and upper bound of $O(l^2np^{n-l+1})$ for conformance testing with reliable **reset**s. When $n = l$, namely the *actual* size of the black box automaton is known, this is a much more tractable complexity than that of black box checking. Thus, if a model (abstract design) is available or feasible to construct, then a good strategy for the developer of a system is to separately do a conformance test of an abstract design against the system, and then model check the design with respect to various properties. However, when a model is not available or n can be considerably bigger than l this approach does not help.

It is quite clear that the off-line strategy is suboptimal as it does not take advantage of the property at hand. On the other hand, the on-the-fly strategies, while still exponential, may work in practice in some important cases. One case is when an error

exists and the estimate n is much higher than the actual size of the checked system or the size of a portion of the system that provides a counterexample. Another case is when the specification automaton P limits the possible bad executions considerably. An example for a "helpful" specification will be that P specifies sequences of the form $\alpha^*(\beta + \gamma)$. An example of an "unhelpful specification" is $X^*\alpha$, where X allows any letter of Σ except α.

The last strategy, based on learning, uses the property P while trying to learn the structure of B. Thus, an error may be found before completing the construction of a minimized automaton equivalent to B. It is also possible that no explicit bound is given on the size of the black box automaton. In this case, we can use the strategy as long as we are willing to spend time.

There is a number of issues in black box checking that deserve further investigation. Some open problems are finding strategies for partially specified automata, or known automata where the actual implementation deviates from the known design in no more than k changes ('implementation errors'). Another problem is to develop an algorithm for black box checking when reliable **reset** moves are not available. It is possible that similar techniques can be used by combining the learning algorithm of [19] with the conformance testing algorithm of [25] for machines without reset.

References

[1] R. Alur, C. Courcoubetis, M. Yannakakis, Distinguishing tests for nondeterministic and probablistic machines, Symposium on Theory of Computer Science, 1995, ACM, 363–372.

[2] D. Angluin, Learning Regular Sets from Queries and Counterexamples, Information and Computation, 75, 87–106 (1978).

[3] J. R. Büchi, On a decision method in restricted second order arithmetic, Proceedings of International Congress on Logic, Methodology and Philosophy of Science, Palo Alto, CA, USA, 1960, 1–11.

[4] T. S. Chow, Testing software design modeled by finite-state machines, IEEE transactions on software engineering, SE-4, 3, 1978, 178–187.

[5] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, to appear 1999.

[6] R. Gerth, D. Peled, M. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, Protocol Specification Testing and Verification, 1995, Chapman & Hall, 3–18, Warsaw, Poland.

[7] P. Godefroid, Model checking for programming languages using VeriSoft, Proc. 24th ACM Symp. on Progr. Lang. and Sys., 174-186, 1996.

[8] G. J. Holzmann, The model checker SPIN, IEEE transactions on Software Engineering, 23(5):279–295.

[9] F. C. Hennie, Fault detecting experiments for sequential circuits, Proc. 5th Ann. Symp. Switching Circuit Theory and Logical Design, 95-110, 1964.

[10] G. J. Holzmann, D. Peled, The State of Spin, 8th International Conference on Computer Aided Verification, Springer Verlag, LNCS, 1102, 385-389, 1996, New Brunswick, NJ, USA.

[11] Z. Kohavi, Switching and Finite Automata Theory, 1978, McGraw Hill.

[12] R. P. Kurshan, Computer-Aided Verification of Coordinating Processes : The Automata-Theoretic Approach, Princeton University Press, 1995.

[13] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines - a survey, Proceedings of the IEEE, 84(8), 1090–1126, 1996.

[14] O. Maler, A. Pnueli, On the learnability of infinitary regular sets, Information and Computation 118 (1995), 316-326.

[15] E. F. Moore, Gedanken-experiments on sequential machines, Automata Studies, Princeton University Press, 1956, 129–153.

[16] G. J. Myers, The Art of Software Testing, Wiley International, 1979.

[17] A. Pnueli, The temporal logic of programs, $18th$ IEEE symposium on Foundation of Computer Science, 1977, 46–57.

[18] J.H. Reif, The complexity of two-player games of incomplete information, Journal of computer and system sciences, 29, 1984, 274–301.

[19] R. L. Rivest, R. E. Schapire, Inference of finite automata using homing sequences, Information and Computation 103, 299-347, 1993.

[20] D. P. Sidhu, T. K. Leung, Formal methods for protocol testing: a detailed study, IEEE Trans. Sw Eng., 15, 413-426, 1989.

[21] W. Thomas, Automata on infinite objects, Handbook of Theoretical Computer Science, MIT Press, J. van Leeuwen (Ed.), 135–192.

[22] B. A. Trakhtenbrot, Y. M. Barzdin, Finite Automata: Behavior and Synthesis, North Holland, 1973.

[23] M. Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, Proceedings of the First Symposium on Logic in Computer Science, Cambridge, UK, 322-331.

[24] M. P. Vasilevskii, Failure diagnosis of automata, Kibertetika, no 4, 1973, 98–108.

[25] M. Yannakakis, D. Lee, Testing finite state machines: fault detection, J. Computer and Syst. Sci., 50, 209-227, 1995.