Black Box Checking

Doron Peled
The University of Texas at Austin
Department of Electrical and Computer Engineering
Austin, TX 78712, USA

Moshe Y. Vardi*Rice University
Department of Computer Science
Houston, TX 77005, USA

Mihalis Yannakakis
Avaya Laboratories
233 Mount Airy Road
Basking Ridge, NJ 07920, USA

October 23, 2001

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.

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 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. 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. 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 often compare the system with some abstract design.

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 on the number of states. 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, 24], 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. There are two notions of time complexity for testing problems: the *testing* time complexity, i.e. the time spent testing the black box, corresponding to the length of the experiment performed, and the *computational* time complexity, i.e., the time spent in computing the test sequence (experiment) and interpreting the result.

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, 25, 26]. 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, 23], and surveys of the more recent results and related work on protocol testing can be found in [13, 21]. 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, 20].

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

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. Note that this version of the intersection is a simple case of Büchi automata intersection, given that $F^B = S^B$. 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 $\alpha_1 \alpha_2 \ldots \alpha_{k-1} \in (\Sigma \cup \{\mathbf{reset}\})^*$, 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. $\alpha_j = \mathbf{reset}$ 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*. (This should be contrasted with games against an adversarial environment, used, for example, in program synthesis [19].) Each such game consists of a nondeterministic machine with finitely many configurations[1] 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 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

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

[1] 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.

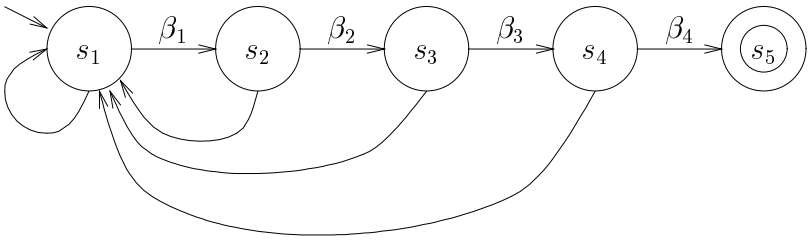

Figure 1: A combination lock automaton

configurations and labels, starting with an initial configuration. A *winning path* ends with a winning configuration. We define the *deterministic time complexity* as the length of the longest winning path in a deterministic 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. We define the *nondeterministic time complexity* of a as the length of the longest winning path in a nondeterministic strategy that ends with a configuration in W^+.

Combination Lock Automata

The following family of automata [15, 25] 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. Such an automaton is said to be 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 lock 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 first describe a simpler problem related to black box checking, which will be presented in the next section. 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 executing the transition corresponding to the label of the last move of the $\exists$-player from the state of the implementation automaton. The moves of the $\exists$-player are the possible input symbols, or a **reset** followed by a symbol.

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, by trying out all input labels without success.

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 (testing and computational) time $O(p^n)$.

Theorem 1 *The deterministic testing 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.

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, check if there is a sequence accepted by both P and B.*

Recall that 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 testing time complexity of black box checking is* $\Omega(p^{n-1})$.

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.[2]

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 [23]. 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 testing time complexity, is $O(np^{2n-1})$. If implemented in the straightforward way, the space complexity is also exponential (to record the tree of all the input strings and their output). However, the construction and minimization can be done instead together incrementally within polynomial space. The computational 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 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.

[2] 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.

4.2 An On-the-fly Strategy

We will present now a strategy that is guided by the checked property and can terminate before exploring the entire structure of the black box automaton.

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 is tested to allow executing the transitions of $\sigma_1\,\sigma_2^{n+1}$ after a **reset**.

The first path σ_1, when input to the automaton P, needs to terminate with some accepting state t. The second path σ_2, when starting in state t of P, needs to terminate with t as well. For such a 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.

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 $O(n^2 m)$.

A Deterministic Strategy

As in the nondeterministic case, the strategy finds a path consisting of the components σ_1 and σ_2^{n+1}, where $|\sigma_1|$ and $|\sigma_2|$ is smaller or equal to $n\,m$. This is done by running through the different possible paths in a systematic way.

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:

- The complexity of this strategy grows exponentially with the number of states m of P. However, m is typically 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, e.g., safety properties), 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 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.

The Conformance Testing Procedure

In our strategy, we will use as a procedure an algorithm for conformance testing of a given finite automaton M with a black box automaton B by Vasilevskii and Chow [4, 25]. The (testing and computational) time complexity of this algorithm is $O(k^2 n\, p^{n-k+1})$, where n is the assumed bound on the size of the black box automaton, $k \leq n$ is the size of the given automaton M, 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 nodes.

We give a brief description of the Vasilevskii-Chow algorithm. Assume without loss of generality that the given automaton M is minimized and has k states. Construct (1) a spanning tree T of the automaton M rooted at its initial state, and (2) a 'characterizing' set W of input strings that pairwise distinguish the states of M. That is, a set of strings with the property that for any two distinct states s, t of M, there is a string $w \in W$ which produces different output starting from states s and t (recall that outputs 0 and 1 correspond to disabled and enabled transitions). There is always a set W with at most k strings of length at most k, which can be constructed efficiently by the classical automata minimization algorithms. The Vasilevskii-Chow algorithm applies to the black box automaton B all the strings of the form $\alpha\beta\gamma$ reset, where α ranges over the input strings corresponding to the paths of T starting at the root, β ranges over all input strings of length $n-k+1$, and γ ranges over all strings of W. If B produces the same outputs as M on all these test strings, and B has at most n states, then B is equivalent to M. There are k choices for string α, at most $2p^{n-k+1}$ choices for β and at most k choices for γ. The length of each test string is $O(n)$, so the total length of the test is $O(k^2 n\, p^{n-k+1})$.

In general, the different test strings can be applied in arbitrary order as far as correctness is concerned. However, for efficiency purposes it is advantageous to apply them in increasing order of the substrings β, i.e. first the empty string for β combined with all α and γ, then all strings β of length 1 (i.e. all input symbols), then length 2 etc. In this manner, if the actual size of the black box B is $l \leq n$, and B does not conform with M, then a discrepancy will be discovered for some β of length at most $l - k + 1$, i.e. with a test of length at most $O(k^2\, l\, p^{l-k+1})$. (If $l < k$, then a discrepancy will be found with the length of β equal to 0, i.e., for some string $\alpha\gamma$, thus with test complexity $O(k^3)$.)

The Automata Learning Algorithm

Another procedure that we use in our strategy is an algorithm for learning an automaton B with **reset** using membership tests and questions to an oracle (a teacher) by Angluin [2]. In this 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.

The basic data structure of Angluin's algorithm [2] consists of two finite sets of finite strings V and W over the automaton alphabet Σ, and a table f. The set V is prefix closed (and contains thus in particular the empty string ε). The rows of the table f are the strings in $V \cup V.\Sigma$, while the columns are the strings in W. Let $f(v, w) = 1$ when the sequence of transitions vw is enabled in B (starting from B's initial state), and 0 otherwise. The entry $f(v, w)$ can be computed by performing the experiment vw on the automaton B. [3]

We define an equivalence relation $\equiv mod(W)$ over strings in Σ^* as follows: $v_1 \equiv v_2 \ mod(W)$ when for each $w \in W$, the sequence of transitions $v_1 w$ is enabled in B if and only if the sequence $v_2 w$ is enabled in B. Thus, two strings v_1 and v_2 corresponding to two rows of the table f are equivalent $mod(W)$ if the rows are the same. It is easy to check that this is indeed an equivalence relation. Denote by $[v]$ the equivalence class that includes v. A table f is *closed* if for each $va \in V.\Sigma$, either $f(v, \varepsilon) = 0$, or there is some $v' \in V$ such that $va \equiv v' \ mod(W)$. A table is *consistent* if for each $v_1, v_2 \in V$ such that $v_1 \equiv v_2 \ mod(W)$, either $f(v_1, \varepsilon) = f(v_2, \varepsilon) = 0$, or for each $a \in \Sigma$, we have that $v_1 a \equiv v_2 a \ mod(W)$. Notice that if the table is not consistent, then there are $v_1, v_2 \in V$, $a \in \Sigma$ and $w \in W$, such that $v_1 \equiv v_2 \ mod(W)$, and $v_1 aw \in \mathcal{L}(B)$ but $v_2 aw \notin \mathcal{L}(B)$ or vice-versa. This means that $f(v_1 a, w) \neq f(v_2 a, w)$, and thus we can add aw to W in order to separate v_1 from v_2.

Given a closed and consistent table f over the sets V and W, we construct a *proposed automaton* $M_f = (S, \{s_0\}, \Sigma, \delta, F)$ as follows:

- The set of states S is $\{[v] | v \in V, f(v, \varepsilon) \neq 0\}$.
- The initial state s_0 is $[\varepsilon]$ (where ε is the empty string).
- The transition relation δ is defined as follows: for $v \in V, a \in \Sigma$, the transition from $[v]$ on input a is enabled iff $f(v, a) = 1$ and in this case $\delta([v], a) = [va]$.
- $F = S$.

[3] By this definition, the entries of the table are 0-1. Alternatively we could have defined $f(v, w)$ to be the output sequence produced by B during the application of the string w starting from the state reached after v, i.e. a bit string indicating whether each transition of w is enabled at the corresponding state. This may result in a smaller set W, though the entries of the table will be longer.

The facts that the table f is closed and consistent guarantee that the transition relation is well defined. In particular, the transition relation is independent of which state v of the equivalence class $[v]$ we choose; if v, v' are two equivalent states in V, then for all $a \in \Sigma$ we have that $[va]$ coincides with $[v'a]$ (by consistency) and is equal to $[u]$ for some $u \in V$ (by closure).

There are two basic steps used in the learning algorithms for extending the table f:

add_rows(v) : Add v to V. Update the table by adding a row va for each $a \in \Sigma$ (if not already present), and by setting $f(va, w)$ for each $w \in W$ according to the result of the experiment vaw.

add_column(w) : Add w to W. Update the table f by adding the column w, i.e., set $f(v, w)$ for each $v \in V \cup V.\Sigma$, according the the experiment vw.

We define a subroutine *ANGLUIN* that accepts a data stucture as above consisting of two sets of strings and a table (corresponding to a previous guess of the automaton B), and a counter example σ, on which B and the current guess do not agree. Without loss of generality we can assume that the only disagreement occurs at the last symbol of σ (otherwise, we can truncate σ). The algorithm updates the sets V, W and the table to a new consistent and closed table, and returns the corresponding automaton, denoted *automaton*(V, W, f). The subroutine adds to the table rows corresponding to all the prefixes of the counter example σ, plus all the prefixes extended by a single letter from Σ. Then the subroutine adds rows and columns to the table f until it is both closed and consistent. The subroutine appears in Figure 2.

The Black Box Checking Algorithm

For the purpose of black box checking, 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 σ_1, $\sigma_2 \in \Sigma^*$ such that $\sigma_1 \sigma_2^n$ belongs to one of the checked automata, but not the other.

In either case, we truncate the counterexample string to the first disagreement between the two automata (the black box and the conjectured automaton).

We construct a sequence of automata M_1, M_2, ... that attempt to converge to 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 whether $\mathcal{L}(M_i) \cap \mathcal{L}(P) \neq \emptyset$. If the intersection is not empty, it must contain an ultimately periodic word of the form $\sigma_1 \sigma_2^\omega$ [22]. We input $\mathbf{reset}\, \sigma_1 \sigma_2^{n+1}$ 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 to the equivalence of B with M_i.

```
subroutine ANGLUIN((V, W, f), σ)
       if f is empty then
            let V := {ε}; W = {ε};
            add_rows(ε);
       else
            for each prefix v' of σ that is not in V do
                 add_rows(v');
       while (V, W, f) is inconsistent or not closed do
            if (V, W, f) is inconsistent then
                 find v1, v2 ∈ V, a ∈ Σ, w ∈ W, such that
                      v1 ≡ v2 mod(W) and f(v1a, w) ≠ f(v2a, w);
                 add_column(aw);
            else /* (V, W, f) is not closed */
                 find v ∈ V, a ∈ Σ,
                      such that va ∉ [u] for any u ∈ V;
                 add_rows(va);
       end while
       return automaton(V, W, f)
end ANGLUIN
```

Figure 2: The Angluin Algorithm

The special case of 'safety' properties, i.e. properties that depend only on finite computations, is a bit simpler. In this case P is an automaton on finite words, and we can find a witness finite string $\sigma \in \mathcal{L}(M_i) \cap \mathcal{L}(P)$; we input **reset** σ to the black box as above to check if B allows it, which signifies an error, or does not allow it, which provides a counterexample to the equivalence of B and M_i. If the experiment fails, we use the resulted counterexample σ or $\sigma_1 \sigma_2^{n+1}$ (truncated to the first discrepancy) 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 (VC) 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 $k+1$, $k+2$, ..., n. Note that, by our earlier description of the VC algorithm, we do not have to start in each round the algorithm from scratch, but simply extend it, by testing the strings of the form $\alpha\beta\gamma$ for increasing lengths of the middle substring β. If the bound n is reached without a discrepancy, we declare that the black box satisfies the checked property.

This strategy is described in Figure 3. The procedure TEST takes a black box automaton B along with a state bound n, and a property given in terms of an automaton P that specifies the erroneous computations. It determines whether B satisfies the property, provided that it has at most n states. The procedure call $VC(M, k)$ activates the Vasilevskii-Chow algorithm for conformance testing the current conjectured automaton M 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

```
proc TEST(B:black box; P:prop.  automaton; n:state bound)
       V := W := ∅;  f := empty table;
       M:=ANGLUIN(empty, −);
       while (M has at most n states) do
            X:=M × P;
            if L(X) = ∅ then
                 k:=number of states of M;
                 repeat
                      (conforms, σ):=VC(M, k);
                      k:=k + 1;
                 until k > n or ¬conforms;
                 if conforms then WIN(+);
            else
                 let σ1 , σ2 be strings such that  σ1 σ2^ω ∈ L(X);
                 if B allows reset σ1σ2^n then WIN(-)
                 else σ:=maximum prefix of σ1 σ2^n allowed by B;
            M:=ANGLUIN((V, W, f), σ);
       end while
       return 'the black box has more than n states';
end TEST
```

Figure 3: A strategy using learning and testing

(**fail**,σ), where σ is a (finite) word that is in one of the automata B or M but not the other. The procedure *ANGLUIN* accepts the current table, and a counterexample, and updates the table and returns a new attempted automaton with more states. In the first call to *ANGLUIN* in the strategy, it is executed with an empty table, and the second parameter (the counterexample) is ignored.

For simplicity, we show in the figure only the case of property automata on infinite strings. For property automata on finite strings one can take $\sigma_2 = \varepsilon$ in the counterexample of the Figure. There are three possible outcomes in the procedure TEST: (1) WIN(-) indicates that B does not satisfy the property (and a counterexample can be returned, if so desired), (2) WIN(+) indicates that if B has at most n states then it satisfies the property (and in this case, B is equivalent in fact to the last automaton M constructed by the algorithm), (3) the results of the test imply that B has definitely more than n states (this occurs if the outer loop completes normally and the last statement of the procedure is reached).

There are still several flexibilities in the implementation of the algorithm, not illustrated in Figure 3. We mention a few of these. For simplicity we have kept the calls to the VC and the Angluin algorithm separate. However the two subroutines compute a lot of common information, so in practice one would combine their data structures and eliminate the redundancies. Recall that the VC algorithm uses the paths of a spanning tree T to access the states of the given automaton, and a characterizing set W of strings to distinguish the states from each other. Angluin's algorithm also has a set of strings V to access the states and another set of strings W (corresponding to the

columns of the table) to distinguish the states. When we call the VC procedure in TEST, one choice is to use the set V of strings to access the states (one string of V for each equivalence class) instead of constructing a tree T, and to use the current set W of strings constructed from the calls to Angluin's procedure (corresponding to the columns of the table f). The advantage of doing this is that by construction of the conjectured automaton M, we know that $VC(M,k)$ will succeed for k equal to the number of states of M (all the relevant experiments were run in the construction of the current table f), and hence we can skip that call and continue from there. A possible disadvantage is that the strings in V and W may be longer than necessary.

The opposite choice is to replace the table f with a spanning tree T from the initial state (for example a shortest path spanning tree), and a new characterizing set W, with the objective of making it have as few and short strings as we can find. Furthermore, if this is not the first call to VC, then we can simply extend the tree T from the previous call to reach the new states and extend the characterizing set similarly. If the conformance test then succeeds for k (the number of states of M), and fails at a higher bound, we then have the option in the next Angluin call to ignore the old data structure and use instead the paths of T as the new set V and the characterizing set as the column set W; the entries of the table f are already known from the test executed during the VC(M,k) call. This may lead to shorter subsequent test sequences in the Angluin procedure.

Another variant of the algorithm is the following: Suppose that at some iteration the model checking determines that the conjectured automaton M does not satisfy the property and that it produces an example string that is not accepted by the black box B. If the string σ is too long, instead of feeding it to the *ANGLUIN* procedure, we may first try to find a much shorter counterexample to the equivalence of B and the conjectured automaton M by calling the VC procedure up to some bound on the number of states of B. If the VC procedure succeeds in finding a counterexample, and the first such counterexample that it finds is $\alpha\beta\gamma$, where α is a string corresponding to a path of the spanning tree T, β is a string of length $r \leq d$, and $\gamma \in W$, then it can be shown that all strings of the form $\alpha\beta'$, where β' is a prefix of β, lead to states that are inequivalent to each other and to all the other states that were previously found (i.e. the states of B reached by the paths of the tree T.) Thus, calling the *ANGLUIN* algorithm with the counterexample $\alpha\beta$ will result in at least r more states.

Complexity of the Strategy

We discuss now the testing complexity, i.e., the length of the test. The computational complexity is similar. Suppose that the minimum equivalent automaton of the black box B has $l \leq n$ states. Suppose first that the black box B has an error. Then this error will be discovered by the time the conjectured automaton M reaches size l, that is, at some iteration we will have a conjectured automaton M with at most l states (possibly the minimum automaton equivalent to B, but also possibly an inequivalent one with fewer states), and the model checking algorithm will produce a string in the intersection of M and P, which is allowed by B. Clearly the VC procedure is never called with a state bound greater than l. If the calls to VC are performed incrementally as explained earlier, i.e. by extending the same tree T and characterizing set W, reusing the previously performed test sequences, then the total time spent by all the VC calls is $O(l^3 p^l)$.

We analyze next the cost of the *ANGLUIN* calls. For simplicity and for the purposes of worst

case complexity, we assume that the VC and *ANGLUIN* algorithms use a common set of access and distinguishing strings. Further, we employ the variant mentioned above, in which if B does not allow a string produced by the model checking, then we do not call *ANGLUIN* directly with this counterexample string σ, but instead run the VC algorithm to find another short counterexample. In practice, we would do this only if the counterexample from model checking is too long (in the worst case it can have length l^2m). As mentioned above, if the VC algorithm finds a counterexample string of the form $\alpha\beta$, where α is a path of the tree T and β has length r, then all the r strings $\alpha\beta'$ for the prefixes β' of β lead to new inequivalent states, and we can extend the spanning tree T from the state reached by the path α by hanging a path of new states labeled by β. When we call the *ANGLUIN* procedure with the counterexample string $\alpha\beta$, we would first include in V the string $\alpha\beta$ and its prefixes; note that α and its prefixes are already present (because α is a path of the access tree T). The other strings $\alpha\beta'$ will give new inequivalent states. The same is true of any other strings that are added to V by *ANGLUIN* as a consequence of the table not being closed. Note that each of the latter strings has the form va where $v \in V$ and $a \in \Sigma$; we extend the tree T as we add va to V. From these remarks, it follows that the set V has at all times at most l elements, all of length at most l, and thus the *ANGLUIN* calls perform at most l^2p tests, each of length $O(l)$, for total cost $O(l^3p)$. Clearly, this is dominated by the bound of the VC calls.

Besides the VC and the *ANGLUIN* calls, the remaining tests on the black box apply the strings from the model checking to determine whether B can execute them. If the property is characterized by an automaton on finite strings with m states, then the counterexamples provided by the model checking algorithm have length at most ml. There are at most l such strings, for total length l^2m. Typically, m (the size of the property automaton) is a small constant, $m << l$, and therefore l^2m is dominated by l^3p^l even for $p = 2$. But in general, even if this is not the case (i.e. if m is large), for the purpose of bounding the worst case complexity, we can do the following trick to amortize all, except the last test from the model checking: to test if B can execute a string σ (provided by the model checking), apply succesively prefixes of σ, doubling the length each time, interleaved with tests from the VC algorithm that use approximately equal time. Then the time for these tests is dominated by the VC calls, except for the last test when the conjectured automaton may be the correct automaton. The cost of the last test is $O(lm)$. Thus, for properties that depend on finite executions, the total complexity of TEST in case of a faulty black box is $O(l^3p^l + lm)$. Note that this does not depend on the a priori bound n.

If the property is characterized by an automaton on infinite strings, then the model checking will give strings σ_1 and σ_2 of length at most lm. If B does not allow $\sigma_1\sigma_2^\omega$ then the first failure will occur within the first l repetitions of σ_2, because the black box must be at inequivalent states after σ_1 and after each repetition of σ_2. Thus, all these tests have length l^2m, except for the last one which B can execute completely (showing that B is faulty), which has length lmn. Again, regarding the tests before the last one, their cost is dominated by the VC calls - either because m typically is a small constant, or otherwise apply the trick described above. Consequently, in this case the total testing complexity of the algorithm is $O(l^3p^l + lmn)$. In this case the complexity depends on the bound n, but only linearly.

If the black box satisfies the property, then the strategy will generate the minimum equivalent automaton M in time $O(l^3p^l)$, but then it will have to spend $O(l^2\, n\, p^{n-l+1})$ more time to verify that the black box is indeed equivalent to M (by conformance checking M with B). It should be noted that unless the bound n is a good estimate of the actual size l (more precisely, if $n > 2l$), the

complexity is dominated by this last check that confirms the conformance of B with the conjectured automaton.

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 testing time*

- $O(l^3 p^l + lmn)$*, when there is an error, i.e., the intersection of B and P is nonempty (time $O(l^3 p^l + lm)$ if P is an automaton on finite strings), and*
- $O(l^3 p^l + l^2 n p^{n-l+1})$*, 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 $O(l^3 p^l + lm)$. 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 Büchi automaton that depends on infinite behaviors. Suppose further 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 $\mathbf{reset}\sigma_1$ 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. As described before, we can also interleave the application of successively longer prefixes of $\sigma_1 \sigma_2^\omega$ with running the VC algorithm for approximately equal amounts of time, until either we find a counterexample or 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 [23]. 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 Mealy machines with l states, if an error is present, it will be found after a test of length $O(lp\log^2 l + lmn)$ (respectively, length $O(lp\log^2 l + lm)$ for properties described by automata on finite strings).*

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 $\log_p \log_q l$ [23]. 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 in the worst case 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 [25, 4] showed a lower and upper bound of $O(l^2 n p^{n-l+1})$ 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 then in the worst case we cannot avoid the exponential complexity.

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 conformance testing and 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. Finally, as we pointed out, the exponential lower bound occurs because of certain pathological examples like the combination lock automaton, and in fact for most machines ("almost all" in a probabilistic sense), if there is an error, the strategy will find it almost surely in polynomial 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 [20] with the conformance testing algorithm of [26] for machines without a reset capability.

We have run several experiments of black box checking. An implementation was programmed by Alex Groce from CMU. We used it to verify communication protocols with up to two thousand states. We are currently experimenting with additional heuristics for increasing the number of states that we can handle.

References

[1] R. Alur, C. Courcoubetis, M. Yannakakis, Distinguishing tests for nondeterministic and probabilistic 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, 2000.

[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(1997), 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] A. Pnueli, R. Rosner, On the Synthesis of a Reactive Module, Proc. 16th ACM Symposium on Principles of Programming Languages, Austin, TX, 179–190, 1989

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

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

[22] W. Thomas, Automata on infinite objects, Handbook of Theoretical Computer Science, MIT Press, J. van Leeuwen (Ed.), 135–192, 1991.

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

[24] 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, 1986.

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

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