Grey-Box Checking

Edith Elkind[1], Blaise Genest[1,2], Doron Peled[1], and Hongyang Qu[1,3]

[1] Department of Computer Science, Warwick, Coventry, CV4 7AL, UK
[2] CNRS & IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
[3] LIF, 39 rue Joliot Curie, 13453 Marseille Cedex 13, France

Abstract. There are many cases where we want to verify a system that does not have a usable formal model: the model may be missing, out of date, or simply too big to be used. A possible method is to analyze the system while learning the model (black box checking). However, learning may be an expensive task, thus it needs to be guided, e.g., using the checked property or an inaccurate model (adaptive model checking). In this paper, we consider the case where some of the system components are completely specified (white boxes), while others are unknown (black boxes), giving rise to a grey box system. We provide algorithms and lower bounds, as well as experimental results for this model.

1 Introduction

Tools for analyzing a system (e.g., model-checkers) usually require an accurate model of the system. However, such a model may be difficult to find: while some tools can perform the analysis based on a model constructed directly from the source code, there are few tools that can deal with a binary file or with a chip. A recent paper [12] proposed a method of checking black box systems, that is, systems for which we do not have a model. Later, it was extended to testing based on an approximately accurate model that can be automatically changed when discrepancies are found [9]. This approach is based on interactive learning of finite state systems [2] combined with conformance testing [14,6], and has many applications. For instance, [15] considers deriving a specification from observing a system, and [7,1] apply these techniques in order to guess an efficient property to be used as an interface in assume-guarantee reasoning.

In this paper, we extend the black box checking procedure of [12] to the case where some parts of the system in question are known. Specifically, we focus on the situation where we know the high level description of the system as well as some of its components, while the internal structure of the remaining components is unknown. We call such a system a *grey box*, and use the terms 'white box' and 'black box' to denote the known and the unknown parts, respectively. For instance, a component in a distributed system or a module in a hierarchical system can play the role of the (known) white box or the (unknown) black box. We propose the framework of grey box checking in a concurrent system, where several asynchronous components communicate with each other. We can easily extend our approach and get good complexity bounds for (sequential) hierarchical systems as well [8].

E. Najm et al. (Eds.): FORTE 2006, LNCS 4229, pp. 420–435, 2006.

In some settings, each component can be analyzed separately; in others, we can only test the system as a whole. In both cases, the information available about the white box can speed up the testing considerably. In the first case, the problem essentially reduces to learning the unknown components and thus its complexity does not depend on the size of the white box. For the more challenging case where all components have to be run together, we show that the complexity of checking the synchronous product $\mathcal{B} \times \mathcal{W}$ is substantially higher than the complexity of black box checking $\mathcal{B}$ (the increase in complexity is exponential in the size of the alphabet), but substantially lower than checking $\mathcal{B} \times \mathcal{W}$ as a black box (the savings are exponential in $|\mathcal{W}|$).

Our algorithms are based on the black box checking procedure of [12]. To decrease complexity, we use conformance testers that are better suited to our setup than the standard Vasilevskii-Chow algorithm [14,6]. Our first oracle relies on enumerating all finite automata up to a certain size and has an almost optimal worst case complexity. The second oracle combines the algorithm of [14,6] with ideas from partial order reduction and performs better in some of our experiments (see Section 6). Also, both algorithms use the information about the white box to speed up the learning algorithm of [2]. The experimental data provided by a new tool we are developing shows that the best compromise is to run both algorithms together, so that they help each other find discrepancies. This appears to speed up the process by several orders of magnitude.

While our goal is similar to that of adaptive model checking [9], we see our work as complementing the adaptive approach rather than replacing it. Indeed, adaptive model checking uses an inaccurate model to help the learner; here, we use partial but accurate information about the system being tested. The usefulness of the adaptive model checking has been argued by [9], which demonstrates that the learning algorithm is robust enough to deal with a partially wrong specification. However, there are small modifications to the system (e.g., adding a new state that separates two components of the system) that cannot be handled efficiently by this method. Our approach is likely to be successful if the changes can be limited to a small part of the system, which will then be treated as a black box. In particular, this applies to the case described above. Moreover, sometimes the two techniques can be combined. For instance, we may have an accurate model of a component, some old model of another component that was changed since the model was made, and another component that is totally unknown. Then we can use our approach for the product, and the adaptive approach when analyzing the second component.

2 Preliminaries

A *finite automaton* is a tuple $\mathcal{A} = (S, s_0, \Sigma, \rightarrow)$ where

- S is the finite set of states,
- s_0 is the initial state of $\mathcal{A}$.
- Σ is the finite set of letters (alphabet) of actions.
- $\rightarrow \subseteq S \times \Sigma \times S$ is the deterministic transition function, that is, for all $a \in \Sigma$ and $s, t, t' \in S$, if $s \xrightarrow{a} t$ and $s \xrightarrow{a} t'$ then $t' = t$.

We do not designate a set of accepting states; every state of $\mathcal{A}$ is considered to be accepting. A *run* ρ of $\mathcal{A}$ is a finite or infinite sequence of transitions $(v_i, a_i, v_{i+1}) \in \rightarrow$ with $v_0 = s_0$. An *experiment* is any sequence of labels in Σ^*. Since every automaton that we consider is deterministic, any experiment is associated with at most one run. Abusing notation, we will identify a run with the corresponding sequence of labels. The language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all maximal runs[1]. One can easily test whether an experiment u is a prefix of a run: it suffices to feed the sequence u to the system after a **reset**, letter by letter, and check that each time the next letter is enabled through executing a transition of the system.

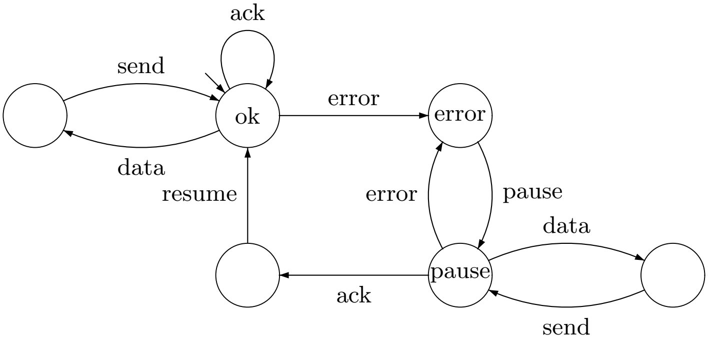

Fig. 1. The automaton *Interface*

2.1 The Black-Box Checking Procedure

Black-box checking was proposed in [12] for verification of (partially) unknown systems. It is based on interleaving learning and model checking. In what follows, we describe this approach in more detail.

The Learning Algorithm. In [2], Angluin describes an algorithm L^* for learning the minimal deterministic automaton that corresponds to a given black box $\mathcal{A}$.

Angluin's learning algorithm builds a candidate automaton $\mathcal{A}^*$ by making experiments on the system $\mathcal{A}$, i.e., invoking a procedure *test*(v) that returns 1 if v is executable in the black box after a **reset**, and 0 otherwise. Once it has obtained a candidate solution $\mathcal{A}^*$ that is consistent with all experiments run so far, it calls an oracle that checks whether $\mathcal{L}(\mathcal{A}^*) = \mathcal{L}(\mathcal{A})$. If not, the oracle gives a minimal-size experiment σ (discrepancy) distinguishing $\mathcal{A}^*$ from $\mathcal{A}$, i.e., a sequence σ that is either in $\mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{A}^*)$ or in $\mathcal{L}(\mathcal{A}^*) \setminus \mathcal{L}(\mathcal{A})$. The learning algorithm then uses σ to refine the current solution. Later, we show how to build this oracle using Vasilevskii-Chow [14,6] algorithm.

To construct a candidate automaton, the algorithm keeps two sets of sequences: a prefix-closed set of *access sequences* $V \subseteq \Sigma^*$ and a suffix-closed set of

[1] A run is *maximal* if it is not a prefix of another run.

distinguishing sequences $W \subseteq \Sigma^*$. Each sequence v in V corresponds to reaching a state of $\mathcal{A}^*$ by executing v from s_0. Different sequences may lead to the same state. Also, the algorithm keeps a table $T : (V \cup V.\Sigma) \times W \rightarrow \{0, 1\}$ such that for any $v \in V \cup V.\Sigma$ we have $T(v, w) = 1$ if and only if $vw \in \mathcal{L}(\mathcal{A})$.

We define the equivalence $\sim \subseteq V \times V$ as $v \sim v'$ if $T(v, w) = T(v', w)$ for every $w \in W$. For (V, W) to represent an automaton, it is necessary that T is *closed*, i.e., for every $v \in V$ and $a \in \Sigma$ s.t. $T(v, a) = 1$, there exists $v' \in V$ with $v' \sim va$. If T is not closed because a is executable after v, but $va \not\sim v'$ for all $v' \in V$, we add va to V. Also, we verify that the table T is *consistent*, i.e., for all $v \sim v'$, if $T(v, a) = 1$, then $T(va, w) = T(v'a, w)$ for all $w \in W$. If this is not the case, the sequence aw is added to W.

When the table T is closed and consistent, we set $\mathcal{A}^* = ([V/\sim], \epsilon, \Sigma, \delta)$, where the transition relation δ is defined as follows. Let $[v]$ be a $\sim$ equivalence class of v. Set $\delta([v], a) = [v']$ when $v' \sim va$. This relation is well defined when the table T is closed and consistent. We then invoke the oracle on $\mathcal{A}^*$. If the oracle returns a discrepancy σ, for each prefix v of σ that is not in V, we add v to V and update T accordingly.

Here is the formal description of one phase of the algorithm L^* after the oracle returned a discrepancy σ.

```
subroutine $L^*(V, W, T, \sigma)$ returns $(V, W, T)$ =
       if $T$ is empty then
            let $V := \{\epsilon\}$, $W := \Sigma$;
            for each $a \in \Sigma$, set $T(\epsilon, a)$ according to $test(a)$
       else
            for each prefix $v$ of $\sigma$ do
                 $add\_rows(v)$;
       while $T$ is inconsistent or not closed do
            if $T$ is inconsistent then
                 find $v_1, v_2 \in V$, $a \in \Sigma$, $w \in W$,
                      such that $v_1 \sim v_2$ and $T(v_1 a, w) \neq T(v_2 a, w)$
                 $add\_column(aw)$
            else
                 find $v \in V$, $a \in \Sigma$,
                      such that $va \notin [u]$ for any $u \in V$
                      $add\_rows(va)$
       end while
end $L^*$
```

Here, the procedure $add\_rows(v)$ checks if $v \in V$, and if not, adds v to V and fills the new rows in T, i.e., makes the experiments **reset** vaw for all $w \in W$ and all $a \in \Sigma \cup \{\epsilon\}$. Similarly, $add\_column(w)$ adds a new distinguishing sequence w to W and updates $T(v, w)$ for each $v \in V \cup V.\Sigma$ by making the experiment **reset** vw.

In our experiments, we use a modified version of L^* algorithm proposed by Rivest and Schapire [13]. The algorithm of [13] adds an appropriately chosen *suffix* of the discrepancy to W (instead of adding prefixes concatenated by a letter from Σ to V, as in Angluin's algorithm). Also, Rivest and Schapire noticed that consistency check is also performed by the conformance algorithm. In fact, this is exactly what is done by Vasilevskii-Chow algorithm when $l = 1$ (see next subsection). Therefore, in their version of the learning algorithm they omit the consistency check.

Let n be an upper bound on the number of states of the minimal deterministic automaton modeling the black box. Suppose that any counterexample returned by the oracle is of size $O(n)$ (this is indeed the case for all oracles considered in this paper). Then for the Rivest–Schapire version of the L^* algorithm we have the following result.

Proposition 1. *[13] The L^* algorithm makes $O(n^2|\Sigma|)$ membership queries and at most n calls to the oracle. Its running time is $O(n^3|\Sigma|)+T_{oracle}$, where T_{oracle} is the total time spent by the oracle.*

Vasilevskii-Chow Algorithm. The oracle is built using the Vasilevskii-Chow algorithm. This algorithm uses the sets V, W and a known upper bound n on the size of the minimal deterministic automaton modeling the black box. In order to check whether $\mathcal{A} = \mathcal{A}^*$, VC algorithm runs both automata on some sequences $y \in \Sigma^*$. We write check$(y) = 1$ if y is either in $\mathcal{L}(\mathcal{A}) \setminus \mathcal{L}(\mathcal{A}^*)$ or in $\mathcal{L}(\mathcal{A}^*) \setminus \mathcal{L}(\mathcal{A})$. The sequences that are tested are those of the form $y = vxw$ with v a selected representative per each equivalence class of $[V/\sim]$, $w \in W$ and $|x| \leq n - |[V\setminus \sim]|$. Intuitively, if two equivalent access sequences are not consistent, then one is not consistent with the actual black box and a new distinguishing sequence can be found.

```
VC(V,W,n):
  k = sizeof($[V/\sim]$);
  for $l =$ $1, \ldots, n-k$
     for each word $x$ of size $l$, $c \in [V/\sim]$, $w \in W$
        let $v$ be an arbitrary representative of $c$;
        if $check(vxw)$ then return $vxw$;
  return void;
```

Proposition 2. *[14,6] It is sufficient to test sequences of the form $y = vxw$ with v selected as representative for each equivalence class of $[V/\sim]$, $w \in W$ and $|x| \leq n - k$ in order to find a difference between $\mathcal{A}^*$ and $\mathcal{A}$, where k is the number of equivalent classes of $[V/\sim]$ and n is a bound on the number of states of $\mathcal{A}$. The algorithm makes $k^2|\Sigma|^{n-k+1}$ membership queries. Its time complexity is $O(nk^2|\Sigma|^{n-k+1})$.*

Observe that the L^* algorithm invokes Vasilevskii-Chow algorithm at most n times, and after each call the value of k increases by at least 1. Therefore, the total number of queries made by Vasilevskii-Chow algorithm during these calls is at most $|\Sigma|^n + 4|\Sigma|^{n-1} + \cdots + n^2|\Sigma| = O(n^2|\Sigma|^n)$, and the total time spent by Vasilevskii-Chow algorithm is $O(n^3|\Sigma|^n)$.

Black Box Checking. Finally, we describe the black box checking procedure [12], which is a way to test whether a given black box $\mathcal{A}$ satisfies a property φ. The property φ describes a set of allowed (or *good*) runs. We assume that it is written in some formal notation such as LTL or Büchi automata, Let $\mathcal{L}(\varphi)$ be the set of runs (the *language* of) the specification φ. We denote by $\mathcal{A} \models \varphi$ ($\mathcal{A}$ satisfies φ) the fact that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\varphi)$.

Suppose that we are given a (partially) unknown system $\mathcal{A}$. Our goal is to check whether there exists a run of $\mathcal{A}$ that does not satisfy φ. Such a run is called a *counterexample.* To do so, we infer an automaton $\mathcal{A}^*$ by running experiments on $\mathcal{A}$. We begin by using the learning algorithm initialized with $V = \epsilon$ and $W = \Sigma$. Then we feed the model checker with the candidate automaton $\mathcal{A}^*$. The model checker tests whether $\mathcal{A}^*$ satisfies φ. If not, it outputs a counterexample σ such that $\sigma \in \mathcal{L}(\mathcal{A}^*) \setminus \mathcal{L}(\varphi)$. We then test σ on $\mathcal{A}$. If $\sigma \in \mathcal{L}(\mathcal{A})$, we have found a genuine counterexample. Otherwise, σ is a discrepancy between $\mathcal{A}$ and $\mathcal{A}^*$ and can be used to change $\mathcal{A}^*$ so that it models $\mathcal{A}$ more accurately. If $\mathcal{A} \models \varphi$, we will have to repeat this procedure until $\mathcal{L}(\mathcal{A}^*) = \mathcal{L}(\mathcal{A})$. However, if $\mathcal{A} \not\models \varphi$, we may find a counterexample before we learn $\mathcal{A}$.

3 Our Model

We associate a set of components $(S^i, s_0^i, \Sigma^i, \rightarrow^i)$ with the automaton $\mathcal{G} = (\prod S^i, \prod s_0^i, \bigcup \Sigma^i, \rightarrow)$, where $\prod_{i=1,\ldots,n}(s_i) \xrightarrow{a} \prod_{i=1,\ldots,n}(t_i)$ iff for all i, either $a \notin \Sigma^i$ and $s_i = t_i$, or $a \in \Sigma^i$ and $s_i \xrightarrow{a} t_i$. We want to verify a property of the whole system $\mathcal{G}$, and we know the alphabet Σ^i used by every component (if not we take $\Sigma^i = \Sigma$).

As a running example, we consider a data acquisition system (DAS) similar to the one used in [16]. It consists of three components *Interface, Command, Sensor*, which communicate as follows. The *Command* can **request** the *Sensor* to send a **data** to the *Interface.* The *Sensor* can inform the *Interface* that an **error** occurred. Finally, the *Interface* can **stop** and **resume** the *Command*, and **send** the data it received to the environment, receiving **acknowledgement** from it. Assume that the *Interface* is given by the automaton in Figure 1; the other two components are unknown. In the beginning, we assume that both *Sensor* and *Command* can always perform each of their internal actions. Alternatively, if we only have an old specification of these components, we can use it to initialize these components, as is done in adaptive model checking [9]. We want to verify that between one **pause** and one **send**, the system $\mathcal{G}$ always performs a **resume**. Of course, a bad sequence of actions seems possible with this *Interface*, with the trace **error pause data send**, but this error may not be possible in the system with the actual *Command* and *Sensor.*

The algorithm that we use depends on whether the components can be analyzed separately, or only as a whole. The latter case may occur if, for instance, the communication is coded in a special way, or if the system is on a chip.

4 Independent Components

In this section, we assume that we can perform a test w on any black box $\mathcal{B}$. Our algorithm is a slight modification of the black box checking algorithm. Let $\mathcal{W}$ be the product of all white boxes. Our goal is to model check the system $\mathcal{G} = \mathcal{W} \times \prod_{i \leq l}(\mathcal{B}_i)$. Suppose $|\mathcal{W}| = m$, $|\mathcal{B}_i| \leq n$ for all $i = 1, \ldots, l$, i.e., $|\mathcal{G}| \leq mn^l$. We repeat the following steps until we find a counterexample or construct a product automaton $\mathcal{G}^*$ with $\mathcal{L}(\mathcal{G}^*) = \mathcal{L}(\mathcal{G})$.

- Execute the learning algorithm for each $\mathcal{B}_i$ separately to construct candidate automata $\mathcal{B}_i^*$.
- Model check the product $\mathcal{G}^* = \mathcal{W} \times \prod_{i \leq l}(\mathcal{B}_i^*)$.
- If no counterexample is found, call the conformance tester on every black box separately and feed the discrepancies to the learning algorithm.
- If a counterexample σ is found, then for all i, set $\sigma_i = \pi_{\Sigma_i}(\sigma)$, where $\pi_{\Sigma_i}(\sigma)$ is the projection of σ on the alphabet Σ_i, and test σ_i on the black box $\mathcal{B}_i$. If each of these tests passes, then the algorithm terminates and returns σ as a real counterexample. Otherwise, we have discrepancies (one per each component), which we then pass to the learner for each black box.

Proposition 3. *The maximal number of tests performed during the black box checking of a system $\mathcal{W} \times \prod_{i \leq l} \mathcal{B}_i$ is $O(l\, n^2\, |\Sigma|^n)$. The time complexity of this procedure is $O(l\, n^3\, |\Sigma|^n)$.*

Observe that the time complexity of running the black box testing procedure on $\mathcal{G}$ is $O(m^3 n^{3l}\, |\Sigma|^{mn^l})$. Thus, it is highly profitable to learn the components separately. For both algorithms, we can apply the method in an incremental way (increasing the size of the tested automata used by the Vasilevskii-Chow algorithm, up to n). In case that the checked system *does not* satisfy the specification, we typically find it much quicker than the worst case complexity (see [12]).

We now show how this algorithm behaves on the data acquisition example. We begin by model-checking the candidate system against our property (between one **pause** and one **send**, the system $\mathcal{G}$ always performs a **resume**), and find a first possible counterexample: **error pause data send**. We find out that *Sensor* never emits an **error** as its first execution (rather, it does nothing without receiving an action request). Thus, we learn that the current model for *Sensor* is wrong and we ask the learner to give a better approximation. The learner comes up with the following table (the rows contain the access sequences V, the columns contain the distinguishing sequences of W initialized with Σ). A $\surd$ in the table means that $w \in W$ is executable after $v \in V$. This table can be interpreted as the following automaton for *Sensor*:

Then, the model-checker verifies the new system with the new *Sensor*, and finds no errors since the action **error** is not allowed in the current model of *Sensor*. Hence, the conformance tester checks both the *Sensor* and the *Command*. For *Sensor*, the conformance tester comes up with the distinguishing sequence **request error** which is fed to the learner.

$T(v,w)$	req	data	error
ϵ	√	x	x
req	√	√	x
req,req	√	√	x
req,data	√	x	x

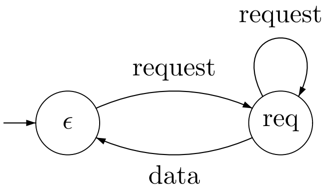

Fig. 2. First inferred black box *Sensor*: experiment Table and corresponding automaton

5 Testing a Grey Product

A more restrictive scenario is when we can only test whether $\sigma \in \mathcal{B} \times \mathcal{W}$. In what follows, we describe several new algorithms for this setting. Despite their simplicity, it turns out that our algorithms are almost optimal. We prove this by showing an (almost) matching lower bound. We focus on the case when there is one white box $\mathcal{W}$ of size m and one black box $\mathcal{B}$ of size n. If there are several black boxes that cannot be tested separately, we consider $\mathcal{B}$ to be their product. In some cases, $\mathcal{B}$ cannot be learned exactly. For instance, if b is in the intersection of both alphabets and $\mathcal{W}$ has no transition labeled by the letter b, then we cannot decide whether any state of $\mathcal{B}$ has a transition labeled by b. Therefore, our goal is to learn a black box $\mathcal{B}^*$ that satisfies $\mathcal{L}(\mathcal{B} \times \mathcal{W}) = \mathcal{L}(\mathcal{B}^* \times \mathcal{W})$. As $\mathcal{W}$ can be a machine that accepts every word of Σ^*, our problem is a generalization of black-box learning. This implies that one needs at least $n^2 \times |\Sigma|^n$ tests. We can also ignore what we know about $\mathcal{W}$ and treat $\mathcal{B} \times \mathcal{W}$ as a black box of size mn. This shows that it suffices to perform $O((mn)^2 \times |\Sigma|^{nm})$ tests of size nm. Clearly, if m is much bigger than n, this approach does not seem attractive.

5.1 Lower Bounds

We start by proving two new lower bounds. They imply that testing a black box combined with a known white box is much more difficult than testing the black box alone. In particular, unlike in black box checking, the number of tests may have to be exponential in the size of the alphabet.

Proposition 4. *For any $n \in \mathbb{N}$, $|\Sigma|$ even, and $x, y \notin \Sigma$, there exists a family of black boxes $\mathcal{F} = (\mathcal{B}_r)_{r \in R}$ and a white box $\mathcal{W}$ with $|\mathcal{B}_r| \leq n+1, |\mathcal{W}| \leq n|\Sigma|^2$ such that $2^{\Omega(n|\Sigma|)}$ tests of size $\Omega(n|\Sigma|)$ are needed to distinguish between $\mathcal{B}_r \times \mathcal{W}$ and $\mathcal{B}_{r'} \times \mathcal{W}$.*

Proof. The automata in $\mathcal{F}$ are constructed as follows. Any automaton in this family has $n+1$ states $s_0, \ldots, s_n$ and uses the alphabet $\Sigma \cup \{x, y\}$. For each $1 \leq i \leq n$, let Σ_i be a subalphabet of Σ of size $|\Sigma|/2$. There is a transition $s_i \xrightarrow{a} s_i$ for every $a \in \Sigma_i$ and a transition $s_i \xrightarrow{a} s_0$ for every $a \in \Sigma \setminus \Sigma_i$. Also, for $i = 1, \ldots, n-1$ there is a transition $s_i \xrightarrow{x} s_{i+1}$. The only transition labeled by y is $s_n \xrightarrow{y} s_0$. Finally, $s_0 \xrightarrow{a} s_0$ for every $a \neq y$. Every choice of subalphabets

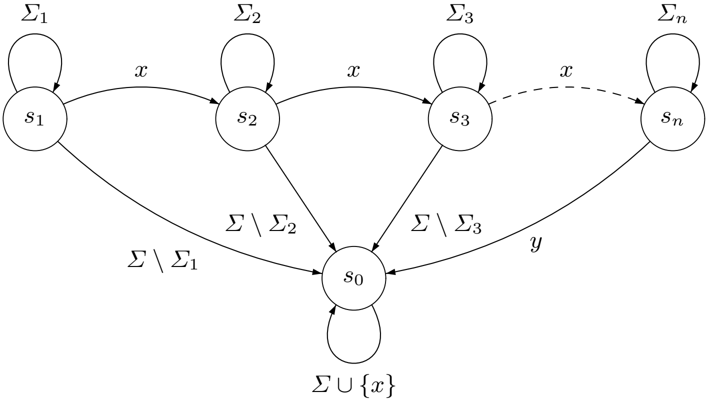

Fig. 3. A black box in $\mathcal{BB}$

$(\Sigma_1, \cdots, \Sigma_n)$ defines a black box in $\mathcal{F}$, which means that $|\mathcal{F}| = \binom{|\Sigma|}{|\Sigma|/2}^n$. Using Stirling's formula, we obtain $|\mathcal{F}| = 2^{\Omega(n|\Sigma|)}$.

To describe the white box $\mathcal{W}$, we fix a strict order $\prec \subseteq \Sigma \times \Sigma$ on letters. Intuitively, we want $\mathcal{W}$ to accept words that consist of n blocks of $|\Sigma|/2$ letters from Σ separated by x's, followed by a y; within each block, the letters should be ordered according to $\prec$. More formally, $\mathcal{W}$ is the minimal deterministic automaton that accepts prefixes of the words $w_1 \cdots w_t$, $t = n(|\Sigma|/2 + 1)$, that satisfy the following: $w_t = y$, $w_{i(|\Sigma|/2+1)} = x$ for all $i = 1, \ldots, n-1$, and finally, for any i such that $i \mod |\Sigma/2| + 1 \neq 0, 1$, we have $w_{i-1} \prec w_i$. It is not hard to see that $\mathcal{W}$ can be implemented using $n|\Sigma|^2$ states.

Clearly, any word of the form $w_1 \cdots w_t$ accepted by $\mathcal{W}$ is a word of the black box associated with $(\Sigma_1, \ldots, \Sigma_n)$, where Σ_i consists of the letters in the ith block of w. On the other hand, all other black boxes do not accept this word. This implies that we need at least $2^{\Omega(n|\Sigma|)}$ tests of size $\Omega(n|\Sigma|)$ each. □

Our second bound shows that the size of the counterexample cannot be bounded by a number lower than nm. Hence, the Vasilevskii-Chow approach of testing every sequence of a bounded size will require at least $|\Sigma|^{nm}$ tests.

Proposition 5. *Let $n \neq m$ be two prime numbers. There exists a white box $\mathcal{W}$ with $m + 1$ states and two black boxes $\mathcal{B}, \mathcal{B}'$ of size at most $n + 1$ such that a word of size nm is needed to distinguish between $\mathcal{B} \times \mathcal{W}$ and $\mathcal{B}' \times \mathcal{W}$.*

Proof. For all $r > 0$, consider an automaton $\mathcal{A}_r$ with $r + 1$ states $s_1, \cdots, s_{r+1}$ and transitions $s_i \xrightarrow{a} s_{i+1}$ for all $i < r$, $s_r \xrightarrow{a} s_1$, and $s_r \xrightarrow{b} s_{r+1}$. The regular language accepted by $\mathcal{A}_r$ is $a^* + (a^{r-1})(a^r)^* b$.

If $\mathcal{W} = \mathcal{A}_m$ and $\mathcal{B} = \mathcal{A}_n$, it is easy to see that the smallest word of $\mathcal{G}$ that contains b is $a^{mn-1}b$ because m and n are distinct primes. This is the smallest word that distinguishes $\mathcal{W} \times \mathcal{B}$ from $\mathcal{W} \times \mathcal{B}'$, where $\mathcal{B}'$ is the automaton with one state s and $s \xrightarrow{a} s$. □

5.2 An Almost Optimal Algorithm

It is not hard to identify the automaton that corresponds to the black box by considering all automata of size at most n.

Proposition 6. *Let $\mathcal{B}$ be a black box of size at most n and $\mathcal{W}$ a known automaton of size m. One can learn $\mathcal{B} \times \mathcal{W}$ with at most $2^{n \times |\Sigma| \times \log n}$ tests of size at most $2nm - 1$.*

Proof. Let $(\mathcal{B}_r)_{r \in \{0, \cdots, l\}}$ be the family of all deterministic finite automata of size at most n. For all $r < l$, if $\mathcal{B}_r \times \mathcal{W}$ and $\mathcal{B}_{r+1} \times \mathcal{W}$ agree on all words of size at most $2nm - 1$, they are equivalent. Otherwise, they have a distinguishing sequence, i.e., a word w of size at most $2nm - 1$ such that $w \in \mathcal{B}_r \times \mathcal{W}$ and $w \notin \mathcal{B}_{r+1} \times \mathcal{W}$ or vice versa. It suffices to test this word to make sure that $\mathcal{B} \neq \mathcal{B}_{r+1}$ or $\mathcal{B} \neq \mathcal{B}_r$. Observe that w can be chosen as the smallest sequence in $(\mathcal{L}(\overline{\mathcal{B}_r^*}) \cap \mathcal{L}(\mathcal{B}_{r+1}^*) \cap \mathcal{L}(\mathcal{W})) \cup (\mathcal{L}(\overline{\mathcal{B}_{r+1}^*}) \cap \mathcal{L}(\mathcal{B}_r^*) \cap \mathcal{L}(\mathcal{W}))$. Moreover, since $\mathcal{B}_r^*$ is deterministic, computing its complement $\overline{\mathcal{B}_r^*}$ is easy. Hence, we have to perform at most l tests of size at most $2nm - 1$ to find a $\mathcal{B}_r$ such that $\mathcal{B}_r \times \mathcal{W} = \mathcal{B} \times \mathcal{W}$. To finish the proof, note that the number of automata of size at most n is bounded by $2^{n \times |\Sigma| \times \log n}$. □

The worst-case running time of the algorithm described above is very close to the lower bound of Proposition 4. In other words, our algorithm is almost optimal. However, it is impractical since it has to test every possible automaton without learning anything before the very last test is performed. Thus, its average complexity is equal to its worst case complexity. On the other hand, if we apply the black box learning algorithm described in Section 2.1 to our grey box, the worst case complexity will be exponential in m, but the average complexity will be much lower. In what follows, we show how to combine the two approaches to construct an algorithm that needs at most $2^{n \times |\Sigma| \times \log n} + mn$ tests of size at most $2mn - 1$ in the worst case, but can be expected to do much better on average. The experiments in Section 6 show that this is indeed the case.

Our algorithm uses the learning algorithm L^* on the grey box $\mathcal{G} = \mathcal{B} \times \mathcal{W}$. Whenever L^* produces a candidate solution $\mathcal{G}^*$, we check that $\mathcal{G}^*$ does not accept sequences in $\overline{\mathcal{W}}$; any such sequence is a discrepancy and can be used to refine $\mathcal{G}^*$. Then, instead of using Vasilevskii-Chow algorithm for conformance testing, we use an oracle that only tests the candidate solution $\mathcal{G}^*$ proposed by L^* on the distinguishing sequences considered in Proposition 6. More precisely, we generate automata of size up to n one by one. Let $\mathcal{B}' = \mathcal{B}_i$ be the most recently generated such automaton. We compute a distinguishing sequence for $\mathcal{B}' \times \mathcal{W}$ and $\mathcal{G}^*$ and test it on the grey box. Clearly, either $\mathcal{B}' \times \mathcal{W}$ or $\mathcal{G}^*$ will behave differently from $\mathcal{G}$. If $\mathcal{B}' \times \mathcal{W} \neq \mathcal{G}$, we conclude that $\mathcal{B}' \neq \mathcal{B}$, so we set $\mathcal{B}' = \mathcal{B}_{i+1}$. If $\mathcal{G}^* \neq \mathcal{G}$, we have a discrepancy on $\mathcal{G}^*$ allowing refinement with L^*. Clearly, the first case can occur at most $2^{n \times |\Sigma| \times \log n}$ times, and the second case can occur at most mn times (each time, the size of $\mathcal{G}^*$ increases). Therefore, the total number of tests is at most $2^{n \times |\Sigma| \times \log n} + mn$. Moreover, as $|\mathcal{G}^*| \leq mn$, the length of each experiment is at most $2mn - 1$.

For this algorithm to be efficient, we need to eliminate *early* many automata. To do so, we use the information about $\mathcal{B}$ provided by tests made by L^*. Namely, if wa is executable in $\mathcal{B} \times \mathcal{W}$, then it is executable in $\mathcal{B}$. If wa is executable in $\mathcal{W}$, but not in $\mathcal{B} \times \mathcal{W}$, then we know that it is not executable in $\mathcal{B}$. To generate the automaton, for each state and label we choose the destination of the transition from this state with this label. The number of automata generated depends heavily on the order in which we generate the transitions. Worse yet, the best ordering may change a lot with the choice of transition. Hence, we decide not to impose this order statically but to determine it dynamically: the next transition chosen is the one that makes the largest number of tests progress, so that hopefully a contradiction is reached and every extension of the current automaton is eliminated.

Another technique to speed up the algorithm is to use information about the white box in order to lower the number of distinguishing sequences per state of the candidate solution. We denote by $white(v)$ the state of $\mathcal{W}$ reached after reading the sequence $v \in V$, where V is an access sequence for the grey box. We also denote by W_s the distinguishing sequences needed to distinguish the states in $\{v \in V \mid white(v) = s\}$, for each state s of $\mathcal{W}$. Recall that in Section 2.1 we defined an equivalence relation $\sim$ on $V \times V$. We can refine $\sim$ as follows: $v \sim v'$ iff $white(v) = white(v') = s$ and for all $w \in W_s$, $T(v, w) = T(v, w')$. Our closure and consistency checks are based on this new equivalence. The modified L^* algorithm only fills those lines $T(v, w)$ with $w \in W_{white(v)}$. Notice that $|W_s| \leq n$. It may be the case that $white(v) = white(v')$ with v and v' accepting the same language, and hence we may have to apply a minimization procedure to get a minimal automaton.

5.3 An Algorithm Based on Partial Order Reduction

Another way to reduce the number of experiments in the conformance step is to use the information about the alphabet. This approach is inspired by partial order reduction [4]. Suppose that we know an independence relation I given by $I = \Sigma^2 \setminus D$, with $(a, b) \in D$ iff $a, b \in \Sigma_i$ for some i. For instance, in the data acquisition example, $(\mathbf{request}, \mathbf{send}) \in I$.

Definition 1. *Let* $\sigma, \rho \in \Sigma^*$. *Define* $\sigma \overset{1}{\equiv} \rho$ *iff* $\sigma = uabv$ *and* $\rho = ubav$, *where* $u, v \in \Sigma^*$, *and* $a\, I\, b$.

That is, ρ is obtained from σ (or vice versa) by commuting an adjacent pair of letters.

Definition 2. *Let* $\sigma \equiv \rho$ *be the transitive closure of the relation* $\overset{1}{\equiv}$. *This relation is often called* trace equivalence *[10].*

For example, for $\Sigma = \{a, b\}$ and $I = \{(a, b), (b, a)\}$ we have $abbab \overset{1}{\equiv} ababb$ and $abbab \equiv bbbaa$.

Let $\ll$ be a total order on the alphabet Σ. We call it the *alphabetic order*. We extend $\ll$ in the standard alphabetical way to words, i.e., $v \ll vu$ and $vau \ll vbw$ for $v, u, w \in \Sigma^*$, $a, b \in \Sigma$ and $a \ll b$.

Definition 3. *Let $\sigma \in \Sigma^*$. Denote by $\tilde{\sigma}$ the least string under the relation $\ll$ that is trace equivalent to σ. If $\sigma = \tilde{\sigma}$, then we say that σ is in* lexicographic normal form *(LNF) [11].*

Our approach is based on using L^* algorithm together with Vasilevskii–Chow conformance oracle. However, instead of checking all sequences of the form vxw, we only check the ones where x is in LNF. Clearly, this preserves the correctness of our algorithm. In what follows, we show that by using appropriate data structures, we can ensure that the overhead due to generating sequences in LNF is not too big. This is also confirmed by our experimental results (see Section 6).

Denote by $\alpha(\sigma)$ the set of letters occurring in σ. Let $\prec_\sigma$ be a total order on the letters from $\alpha(\sigma)$ called the *summary* of σ. It is defined as follows:

Definition 4. *Define $a \prec_\sigma b$ if the last occurrence of a in σ precedes the last occurrence of b in σ. That is, $a \prec_\sigma b$ if and only if $\sigma = vaubw$, where $v \in \Sigma^*$, $u \in (\Sigma \setminus \{a\})^*$, $w \in (\Sigma \setminus \{a, b\})^*$.*

Lemma 1. *Let $\sigma \in \Sigma^*$ be in LNF, and $a \in \Sigma$. Then σa is* not *in LNF exactly when we can decompose $\sigma = vu$, such that (a) $vau \equiv vua$ and (b) $vau \ll vu$.*

Intuitively, this means that we can commute the last a in vua backwards over u to obtain a string that is smaller in the alphabetic order than vu. Note that it is not sufficient to check locally that a does not commute with the previous letter, i.e., the case with $|u| = 1$. Consider $\Sigma = \{a, b, c\}$ and $I = \{(a, b), (b, a), (b, c), (c, b)\}$. Then ca is in LNF, while $cab \equiv bca$, where $bca \ll ca$.

Proof. If the two conditions (a) and (b) hold, then obviously vua cannot be in LNF since it is not minimal under the alphabetic order among sequences equivalent to it.

Conversely, let ρ be the minimal string such that $\rho \equiv \sigma a$. Denote by *first*(v) the first letter of a nonempty string v. Let v be the maximal common prefix of ρ and σ (and thus also of σa). Write $\sigma = vu$ (as in (i)), and $\rho = vw$. Consider the following cases:

1. w starts with an a.
   (a) u_1 does not contain an a. Then $au \equiv ua$, satisfying (ii).
   (b) u contains a. Write $u = u_1 a u_2$, where u contains no a. Then $u = u_1 a u_2 \equiv a u_1 u_2$. Since $\rho = vw \ll vua$, we have that $a = \mathit{first}(w) \ll \mathit{first}(u_1) = \mathit{first}(u)$. Thus, $vau_1u_2 \ll vu_1au_2 = vu$, a contradiction to the fact that σ is in LNF.
2. Write $w = w_1 a w_2$, where w_2 does not contain an a. Then, $w = w_1 a w_2 \equiv w_1 w_2 a \equiv ua$ and thus $w_1 w_2 \equiv u$. Since $vw \ll vu$, we have that $\mathit{first}(w_1) = \mathit{first}(w) \ll \mathit{first}(u)$. Thus, $vw_1w_2 \ll vu = \sigma$ and $vw_1w_2 \equiv vu$. This contradicts the fact that σ is in LNF. □

The following Lemma shows how we can use a summary to decide whether σa is in LNF. Since $|\sigma|$ is usually quite larger than the size of the summary (essentially $|\Sigma|$), this makes the generation of normal forms much more efficient.

Lemma 2. *Let $\sigma \in \Sigma^*$ be in LNF with a summary $\prec_\sigma$ and $a \in \Sigma$. Then σa is not in LNF exactly when there is $b \in \alpha(\sigma)$ such that $a \ll b$ and for each c such that $b \preceq_\sigma c$, aIc.*

In words, this means that it is sufficient to check the commutativity of a with a suffix of the summary that commutes with a, and look among these letters for one that comes *after* a in the alphabetic order. This replaces a similar check for an actual suffix of σ.

Proof. Suppose that σ is in LNF and σa is not. Let u be the shortest suffix of σ according to the conditions of the previous lemma, i.e., $\sigma = vu$ and $vau \equiv vua$. Let b be the head of u. Then $a \ll b$. Let $C = \alpha(u)$. We have aIc for each $c \in C$, hence at least for each $b \preceq_\sigma c$.

Conversely, let $b \in \alpha(\sigma)$ a letter satisfying the conditions of the Lemma. Let u be the shortest suffix of σ that begins with b. Since $\prec_\sigma$ is the summary of σ, it follows that all the letters $c \in \alpha(u)$ satisfy $b \preceq_{\alpha(\sigma)} c$, hence aIc. This means that (a) and (b) from the previous lemma hold. □

For instance, assume we have **request** $\lessdot$ **data** $\lessdot$ **error** $\lessdot$ **resume** $\lessdot$ **pause** $\lessdot$ **send** $\lessdot$ **ack**. Then if the action **error** is seen, the new order $\lessdot$ will be **request** $\lessdot$ **data** $\lessdot$ **resume** $\lessdot$ **pause** $\lessdot$ **send** $\lessdot$ **ack** $\lessdot$ **error**.

Like in other partial order approaches, this algorithm can provide us with a reduction that is at most exponential in the number of concurrent (e.g., independent black box) components. Conversely, in other extreme cases, there can be no reduction at all. It is worth noting that the same idea can be used to improve the learning algorithm. Namely, two equivalent (with respect to commutation) states will never be distinguished, hence the tests for one are copied from the other one.

6 Experimental Results

Our implementation prototype for grey box checking is written in SML and includes roughly 6000 lines of code. We use three kinds of examples: an artificially pathological example simple_n with n components, DAS (data acquisition system) with 4 components from [16] with every event observable, and finally, a system in which the memory is incremented and decremented by two processes through a COMA coherency protocol with unobservable actions (COMA was already used in [9], though modeled differently). The two different versions of COMA correspond to different initializations of the memory. Notice that we only include the learner/conformance part, since the model checking part is the same for all algorithms considered. The algorithms are based on Rivest–Schapire's version of the learning algorithm L^*, but call different conformance testers: VC for the usual Vasilevskii-Chow algorithm, LNF for VC generating only sequences in LNF, GBC for Grey Box Checking, i.e., generating distinguishing sequences from the possible automata, and LNFGBC, which uses mainly LNF with calls to GBC when no short sequences were found by LNF.

For each example, we indicated the number of states of the product $\mathcal{G}$ to learn, the number of letters of the alphabet, and the size 'leng.' of the largest experiment needed to distinguish two different states. Then for each algorithm, we give the number of experiments needed to learn the whole system (M indicates millions). We also give an indicative value of the time needed in parentheses, in minutes (or seconds if 's' is specified). All tests were realized on a P-M@1.2Ghz with 256MB of dedicated memory. In Grey Box Checking, we consider only one component as known, the other components being black boxes that cannot be tested separately. In COMA, the black box $\mathcal{B}$ and the white box $\mathcal{W}$ are close in size. In simple_2, $\mathcal{W}$ is much bigger than $\mathcal{B}$. For DAS and simple_4, $\mathcal{B}$ is much bigger than $\mathcal{W}$.

example	states	letters	leng.	VC	LNF	GBC	LNFGBC
simple_2	19	2	18	$.5M$ (9)	$.5M$ (9)	388 (1s)	444 (1s)
simple_4	82	6	9	$7.2M$ (22)	$2.3M$ (3)	too long	$2.3M$ (4)
DAS	73	12	4	$.25M$ (13s)	$.13M$ (8s)	too long	$.13M$ (10s)
COMA(1)	48	8	6	$9.8M$ (33)	$5.7M$ (16)	1821 (120)	$.4M$ (2)
COMA(2)	48	8	7	$46M$ (190)	$25M$ (75)	1731 (170)	$.4M$ (2)

Partial Order

– The overhead in time due to the computation of the lexicographic normal form (LNF) is negligible in all the tests we did.
– Apart from simple_2, which has no commutation, partial order results in a speedup by a factor of 2 to 7. While the speedup in DAS is due to the equivalence relation that we consider on states (the length of the distinguishing sequences is too small), the longer the distinguishing sequences are, the more commutations can be found and the better the speedup is.

Grey Box Checking

– GBC tests very few distinguishing sequences compared with LNF. However, the time taken is not linear in the number of tests performed.
– simple_2 is the pathological case for VC, which explains why Grey Box Checking succeeds. One is, however, unlikely to find such cases in real life.
– In many cases, a pure Grey Box Checking approach is unpractical. However, a distinction should be made between two cases: In Simple_4, no information guides the generation of automata. Even generating all automata with 3 states takes hours, and is useless. On the other hand, in the more realistic DAS example, the initialization gives a lot of information. Although the number of letters is high, every automaton of size 8 respecting the information can be generated within 90 seconds. There are roughly 430,000 such (partial) automata, compared to about 8^{96} if no information was known.
– When VC is efficient ('leng.' is small), Grey Box Checking is useless (DAS).
– Using LNFGBC, i.e., combining Grey Box Checking and LNF can be much more efficient than any of them separately in non-artificial cases (COMA).

Moreover, the overhead of GBC as a helper of LNF is small even in the case where GBC is useless, and can lead to impressive speedup (100 times in COMA(2)).

- Many improvements are possible, e.g., using some of the tests realized by LNF as information to guide the generation of automata.

7 Conclusion

Black box checking [12] was suggested as a way to directly verify a system when its model is not given but a way of conducting experiments is provided. In this paper we studied an extension of this problem, where our system is decomposed into a known part (white box) and unknown part (black box, or a collection of concurrently operating black boxes).

In particular, one of the most interesting cases that we address here is that of an unknown system (i.e., a black box) that is connected to a device whose specification is given (a white box), where both components are coupled, i.e., we can perform the experiments only on the combined system. We prove that the complexity of verifying such a system is strictly in between that of verifying the properties of the black box alone and that of considering the complete structure as a big black box for which no specification is given. We provide algorithms and heuristic methods for verifying such systems.

We implemented the proposed algorithms and showed that this approach can be practical. We performed several experiments verifying that the overhead of these techniques is small, while in some real life cases, the speedup over the black box checking algorithm can be up to two orders of magnitude.

References

1. R. Alur, P. Madhusudan, and W. Nam. Symbolic Compositional Verification by Learning Assumptions. In *CAV'05*, LNCS, 2005.
2. D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75, 87-106 (1987).
3. R. Alur, R. Grosu and M. McDougall. Efficient Reachability Analysis of Hierarchical Reactive Machines In *CAV'00*, LNCS 1855, p.280-295, 2000.
4. E. M. Clarke, O. Grumberg, D. Peled, *Model Checking*, MIT Press, 1999.
5. E. Clarke, D. Long, K. McMillan. Compositional Model Checking. In *LICS'89*, IEEE , p.353-362, 1989.
6. T.S. Chow. Testing software design modeled by finite-states machines. In *IEEE transactions on software engineering*, SE-4, 1978, 178-187.
7. J. Cobleigh, D. Giannakopoulou, C. Pasareanu. Learning Assumptions for Compositional Verification. In *TACAS'03*, LNCS 2619, p.331-346, 2003.
8. E. Elkind, B. Genest, D. Peled and H. Qu. Grey-Box Checking. Internal Report, available at `http://www.crans.org/~genest/EGPQ.ps`.
9. A. Groce, D. Peled and M. Yannakakis. Adaptive Model Checking. In *TACAS'02*, LNCS 2280 , p.357-370, 2002.
10. A. Mazurkiewicz, Trace Semantics, Proceedings of Advances in Petri Nets, 1986, Bad Honnef, Lecture Notes in Computer Science, Springer Verlag, 279–324, 1987.

11. E. Ochmanski, Languages and Automata, in The Book of Traces, V. Diekert, G. Rozenberg (eds.), World Scientific, 167–204.
12. D. Peled, M. Vardi and M. Yannakakis. Black Box Checking. In *FORTE/PSTV'99*, 1999.
13. R. Rivest and R. Schapire. Inference of Finite Automata Using Homing Sequences. *Information and Computation*, 103(2), p.299-347, 1993.
14. M.P. Vasilevskii. Failure diagnosis of automata. *Kibertetika*, no 4, p.98-108, 1973.
15. W. Weimer and G. Necula Mining Temporal Specifications for Error Detection. In *TACAS'05*, LNCS 3440, p.461-476, 2005.
16. G. Xie and Z. Dang. Testing Systems of Concurrent Black-boxes - an Automata-Theoretic and Decompositional Approach. In *FATES'05*, LNCS, 2005.