Grey-Box Checking

Edith Elkind[1], Blaise Genest[1&2], Doron Peled[1], 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, F 13453 Marseille Cedex13, France

Abstract. There are many cases where we cannot deal with a formal model of a system: either it is missing, out of date, or simply too big. A possible method is to learn the behavior of the system to analyze it, e.g. to verify it. However, learning may be an expensive task, thus it needs to be guided, using the property to check (black box checking), or an out of date model for instance. In this paper, the high-level shape of a system is well specified and known (white box), but some of its components are unknown (black boxes), giving rise to a grey box system. Our main result focuses on the product of a known and of an unknown component (testing the unknown component directly being forbidden). Compared with adaptive model checking [8], we use the high level specification more accurately, which yields a provably better complexity.

1 Introduction

Tools to analyze 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 [10] 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 using an approximately accurate model that can be automatically changed when discrepancies are found [8]. This approach is based on interactive learning of finite state systems [2] combined with conformance testing [12, 6], and has many applications. For instance, [13] 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 [10] 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 extend our approach and get good complexity bounds for (sequential) hierarchical systems as well.

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. Even for the more challenging case where all components have to be run together, we provide algorithms that can learn a grey box system in time that is polynomial in the size of the white box (but still exponential in the size of the black box). On the other hand, we prove that such a system may be much harder to learn than a stand-alone black box. We give a lower bound showing that the increase in complexity can be exponential in the size of the alphabet.

Our algorithms are based on the black box checking procedure of [10]. To decrease complexity, we use conformance testers that are better suited to our setup than the standard Vasilevskii-Chow algorithm [12, 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 [12, 6] with ideas from partial order reduction and performs better in some of our experiments (see Section 7). 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 [8], 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 [8], 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_i 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 $t' = t$.

Notice that every state is considered as final. A run ρ of $\mathcal{A}$ is a sequence of transitions $(v_i, a_i, v_{i+1}) \in \rightarrow$ with $v_0 = s_0$. Notice that since every automaton that we consider is deterministic, a sequence of labels $(a_i) \in \Sigma^*$ is associated with at most one run: we will abusively denote as run a sequence of labels associated with a run. The language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of sequences of labels $u = (a_i) \in \Sigma^*$ that are runs. Notice that we can test easily whether an experiment is a run or not: it suffices to input the sequence u after a **reset** to the system until the system does not accept the input. Either the system accepts every input and we do have a run, or else the experiment is not a run.

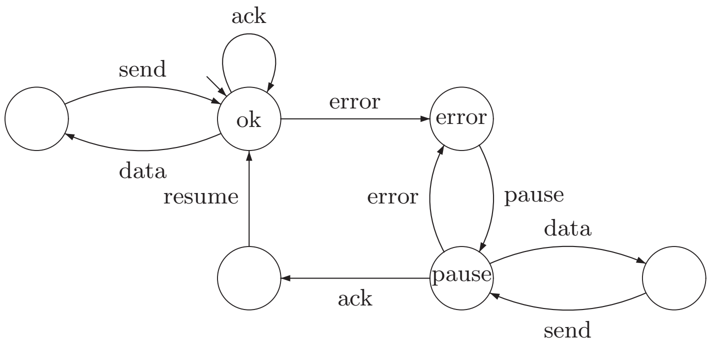

Fig. 1. The automaton *Interface*.

Every system $\mathcal{G}$ considered is (partially) unknown. We are also given a property φ that we want to verify on $\mathcal{G}$. More pragmatically, we want to find a run of $\mathcal{G}$ that does not respect φ, called a *counterexample*. To do so, we infer an automaton $\mathcal{G}^*$ by testing experiments on $\mathcal{G}$. If we find a counterexample for $\mathcal{G}^* \not\models \varphi$, either it is a run of $\mathcal{G}$ and we found a genuine counterexample. Else, it means we have found a *discrepancy* between $\mathcal{G}$ and $\mathcal{G}^*$ that allows to model $\mathcal{G}^*$ more accurately with respect to $\mathcal{G}$. While we may need to refine $\mathcal{G}^*$ such that $\mathcal{L}(\mathcal{G}^*) = \mathcal{L}(\mathcal{G})$ when $\mathcal{G} \models \varphi$, a counterexample is found with an approximated $\mathcal{G}^*$ when such a counterexample exists.

3 The Black-box Checking Procedure

We explain here the steps of the black-box checking procedure proposed in [10].

3.1 The Learning Algorithm

We use a slightly modified [11] version of the learning algorithm L^* from [2] to learn the minimal deterministic automaton corresponding to a black box.

The algorithm uses an oracle (later, we show how to build this oracle using Vasilevskii-Chow [12, 6] algorithm) that can tell for every candidate automaton $\mathcal{G}^*$ whether $\mathcal{L}(\mathcal{G}^*) = \mathcal{L}(\mathcal{G})$, and if not, give a minimal-size experiment u (discrepancy) distinguishing $\mathcal{A}^*$ from $\mathcal{G}$, i.e., a sequence u that is either in $\mathcal{L}(\mathcal{G}) \setminus \mathcal{L}(\mathcal{G}^*)$ or in $\mathcal{L}(\mathcal{G}^*) \setminus \mathcal{L}(\mathcal{G})$. Angluin's algorithm builds a candidate automaton $\mathcal{G}^*$ by making experiments on the system $\mathcal{G}$. More formally, it uses 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 $\mathcal{A}^*$, it calls the oracle. If $\mathcal{A}^*$ is not equivalent to $\mathcal{A}$, the oracle outputs a discrepancy; the algorithm uses this discrepancy to refine the candidate solution.

The algorithm keeps two sets of sequences $V, W \subseteq \Sigma^*$ of actions. The set V stands for the prefix closed set of *access sequences* allowing accessing the states from the initial state. Two access sequences of V may lead to the same state of $\mathcal{A}$. The set W is a suffix-closed set of *distinguishing sequences*. Moreover, the algorithm keeps a table $T : (V \cup V.\Sigma) \times W \rightarrow \{0, 1\}$ such that $T((v, a), w) = 1$ if and only if $vaw \in \mathcal{L}(\mathcal{A})$. We define the equivalence $\sim \subseteq V \times V$ as $v \sim v'$ if $T(v, w) = T(v', w)$ for every $w \in W$. The following condition is sufficient in order for V, W to represent an automaton: T is said to be closed if for every v and $a \in \Sigma$ s.t. $T(v, a) = 1$, there exists $v' \in V$ with $v' \sim va$; if this holds, we write $\delta(v, a) = v'$. If T is not closed because v has no successor with a in V but a is fireable after v, we add va to V. The original version of Angluin's algorithm also performed a consistency check, i.e., it verified that for all $v \sim v'$, if $T(v, a) = 1$, then $T(va, w) = T(v'a, w)$ for all $w \in W$, and if this was not the case, a new sequence was added to W. Rivest and Shapire noticed that this 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, we omit this step from our description of Angluin's algorithm.

When V, W is closed, a candidate (deterministic) automaton is built as $\mathcal{G}^* = ([V/\sim], \epsilon, \Sigma, \delta)$, that is, its states are the equivalence classes of $\sim$. If the oracle answers that w is a discrepancy, then w is added to W and the learning algorithm finds the next candidate automaton with the new sets V, W.

Proposition 1 *[11] The complexity of the L^* algorithm is $O(n^2 \cdot |\Sigma|)$, where n is the number of states of the minimal deterministic automaton modeling the black box.*

Here is the formal description of one phase of the algorithm L^* after the oracle returned a discrepancy τ. The initialization step sets $V = \epsilon, W = \Sigma$ and $T(\epsilon, a) = 1$ for all $a \in \Sigma$ executable in the black box after a **reset**.

```
Lstar(V,W,T,τ):
  W= W ∪ {τ};
  for each suffix τ' of τ, add_column(τ);
  While( V,W is not closed)
    find v,a with T(v,a) = 1, and va ∉ [v'] for any v'
    add_row(va);
  return (V,W,T);
```

Here, the procedure *add_column*(x) (resp. *add_rows*(x)) adds x to W (resp. V) and updates T accordingly.

3.2 Vasilevskii-Chow Algorithm

The Oracle is built from 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 test for equality, VC algorithm checks some sequences $y \in \Sigma^*$; we have $\text{check}(y) = 0$ if y is either in both $\mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{A}^*)$ or in neither of these sets. The sequences that are tested are those of the form $y = vxw$ with $v \in [V/\sim], w \in W$ 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,m):
  k =  sizeof([V/~]);
  for l =  1,...,n-k
     for each word x of size l, v ∈ [V/~],  w ∈ W
        if check(vxw) then return vxw;
  return void;
```

Proposition 2 *[12, 6] It is sufficient to test sequences of the form $y = vxw$ with $v \in [V/\sim], w \in W$ 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 a bound on the number of states of $\mathcal{A}$. There are $O(k^2|\Sigma|^{n-k+1})$ such sequences.*

We give here the proof of this property for sake of completeness, and also because we will need it in section 8.2.

Proof. If there exists a word u distinguishing $\mathcal{A}$ from $\mathcal{A}^*$, then taking $w = \epsilon$, we have a word uw distinguishing $\mathcal{A}$ from $\mathcal{A}^*$. Let us take u the smallest possible such that there exists $w \in W$ and uw distinguishes $\mathcal{A}$ from $\mathcal{A}^*$. First, if $|u| \leq n - |[V/\sim]|$, then $u' = u$ is fine with $v' = \epsilon, w' = w$.

Else, assume that $|u| > n - |[V/\sim]|$. Let s_i be the state of $\mathcal{A}$ seen after $u_1 \cdots u_i$ and s_i^* the state of $\mathcal{A}^*$ seen after $u_1 \cdots u_i$ for all i.

Assume by contradiction that there exist $i < j$ with $s_i = s_j$. First, if we have $s_i^* = s_j^*$, then we can delete $u_i \cdots u_j$ which is a loop for both $\mathcal{A}$ and $\mathcal{A}^*$ to obtain a run u' smaller than u with $u'w$ distinguishing $\mathcal{A}$ from $\mathcal{A}^*$, contradicting the minimality of u. Else, $s_i^* \neq s_j^*$, and there exists $w' \in W$ that distinguishes s_i^*

from s_j^*. But then both $u_1 \dots u_i w'$ and $u_1 \cdots u_j w'$ agrees for $\mathcal{A}$ since $s_i = s_j$, hence one distinguishes $\mathcal{A}$ from $\mathcal{A}^*$, and we again contradict the minimality of u. It means that (s_i) are pairwise distinct.

Since the set of states $[V/\sim]$ is a subset of set of states of $\mathcal{A}$, there is at least one i among the $n - |[V/\sim]|$ last ones such that $s_i \in [V/\sim]$.

By contradiction, assume that $s_i \neq s_i^*$. There exists $w' \in W$ which distinguishes s_i from s_i^*, and $u' = u_1 \cdots u_i w'$ distinguishes $\mathcal{A}$ from $\mathcal{A}^*$ with a smaller size than u, contradicting the minimality of u.

Thus $s_i = s_i^*$. Let $v' = s_i$, $w' = w$ and $u' = u_i \cdots u_n$. We have that $v'u'w'$ distinguishes $\mathcal{A}^*$ from $\mathcal{A}$, with $|u'| \leq |[V/\sim]|$. □

Remark 1. As we already mentioned, the learning algorithm is a slight modification of Angluin's original algorithm [2]. Actually, the original algorithm checks also that no sequence aw was a distinguishing sequence for the candidate automaton, with a letter $a \in \Sigma$, and $w \in W$. This task is already done by the VC algorithm, so there is no need to do it twice.

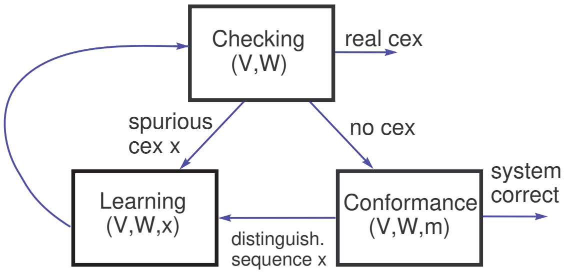

Fig. 2. The black box checking procedure

3.3 Black Box Checking

Finally, we describe the black box checking procedure [10], which is a way to test whether a given Black Box $\mathcal{A}$ with a bound m on its number of states satisfies a property φ. We assume that we can use some model-checker that depends on the formalism provided for φ.

We begin by using the learning algorithm initialized with $V = \epsilon$ and $W = \Sigma$. Then we feed the model checker with the candidate automaton. The model checker tests whether this model satisfies φ. If not, it obtains a counterexample x that it tests in the Black box. If the counterexample is a valid one, then the procedure can stop and report that it has found a problem x. Otherwise, x is spurious, and the learner is fed with this distinguishing sequence. If no counterexample is found, the model checker finds that the model provided satisfies φ, then the candidate models is given to the conformance test of Vasilevskii-Chow. This test either replies that $\mathcal{A}$ and $\mathcal{A}^*$ are equivalent, thus the property is satisfied by the black box, or it provides some distinguishing sequence that is fed to the learner.

3.4 Adaptive Model Checking

The idea behind adaptive model checking is to initialize the learner with a specification that is closely related to the black box being tested. For instance, if an old specification is known but some changes have been made to the system since, then this specification can be used to initialize the learner instead of initializing $V = \epsilon, W = \epsilon$. [8] showed that the learning algorithm is robust enough to handle a partly wrong specification and will make its way through the usable information and the unusable information. They also explain how to easily obtain the sets V and W corresponding to an automaton specification, by using a modified version of Hopcroft's minimization procedure. Experiments that they run show a potentially good improvement over initializing V, W as empty, but the worst case complexity cannot be lowered. More precisely, imagine a system S composed of two parts S_1, S_2. In the specification that we have (from an old design), the only way to go from S_1 to the part S_2 is in doing action a. However, the black box has been slightly modified since the old specification, and now both actions b then a has to be triggered to go from part S_1 to S_2. The set V initializing the learner will be $V = V_1 \cup V_2$ representing each of the part, where each state of V_2 is of the form vaw with $v \in S_1, w \in S_2$. The first action of the learner is to forget every states of V_2 since it does not correspond to any actual state of the black box, leaving the learning algorithm with the states of V_1 as an initialization. The initialization of the distinguishing sequences W is harder to understand though.

We want to use more accurate information about the structure to lower effectively the complexity. Such information may be available through an old specification, as the global (high level) structure of the system being tested is unlikely to change; only lower level components usually change. For instance, we may know that the two parts S_1, S_2 of S was unchanged, only the interface between them was changed, allowing us to keep every state from both S_1 and S_2.

4 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 $(s_i)_{i=1\cdots n} \xrightarrow{a} (t_i)_{i=1\cdots n}$ 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 will explain in the last section how to improve a sequential setting as well. We want to verify a property of the whole system $\mathcal{G}$, and we know the alphabet Σ^i used by every box (if not we take $\Sigma^i = \Sigma$). As a running example, we consider a data acquisition system (DAS) close to the one used in [14]. It consists of three components *Interface, Command,*

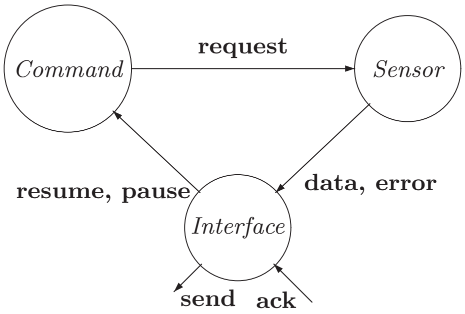

Fig. 3. An accurate high-level description of the system.

Sensor that 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 every of their internal actions; alternatively, if we had an old specification of these components, we could use it to initialize these components, as is done in adaptive model checking [8]. 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 what can be done with the system, namely, 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.

5 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 of figure 2. Let $\mathcal{W}$ be the product of all white boxes. Our goal is to model check the system $\mathcal{G}^* = \mathcal{W} \times \prod(\mathcal{B}_i^*)$.

- If no counter example is found, we call the learner/conformance tester on every black box separately to learn new black boxes $\mathcal{B}_i^*$; we then recompute $\mathcal{G}^*$ and loop.
- If a counter example w is found, then for all i, we test $w_i = \pi_{\Sigma_i}(w)$ on the black box $\mathcal{B}_i$. If every test passes, then the algorithm terminates and returns w as a real counterexample. Otherwise, we have discrepancies to feed the learner with for each black box, and we loop.

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 at most $O(l \times n^2 \times |\Sigma|^n)$, where n is a bound on the size of every black box $\mathcal{B}_i$, and l the number of black boxes.*

This complexity compares favorably with the complexity $O(m^2 n^{2l} \times |\Sigma|^{n^l})$ of running the black box testing procedure on $\mathcal{G}$. Note, however, that for both algorithms the practical instances are unlikely to exhibit this complexity since we use [12, 6] algorithm with decreasing complexity each time a new state is discovered.

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 finds a first possible counterexample: **error**, **pause**, **data**, **send**. We find out that *Sensor* never emits an **error** as its first execution (it rather 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 accessing sequences V, the columns contains the distinguishing sequences of W, initialized at Σ). A $\checkmark$ in the table means that $w \in W$ is an executable sequence after $v \in V$. This table can be interpreted as the following automaton for *Sensor*:

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

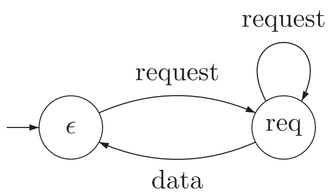

Fig. 4. First infered black box *Sensor*: experiment Table and corresponding automaton.

Then, the model-checker verifies the new system with the new *Sensor*, and finds no error since the action **error** is not allowed in the current model of *Sensor*. Hence, the conformance testing checks both the *Sensor* and the *Command*. The current model of *Command* overapproximate the real black box, so no real execution is found not to be part of the current model. For Sensor, the conformance testing comes up with the distinguishing sequence **request, error** which is fed to the learner.

$T(v,w)$	request	data	error	request error
ϵ	$\surd$	x	x	x
request	$\surd$	$\surd$	x	x
request,request	$\surd$	$\surd$	x	$\surd$
request,data	$\surd$	x	x	x
request,request,request	$\surd$	x	$\surd$	$\surd$
request,request,request,request	$\surd$	x	$\surd$	$\surd$
request,request,request,error	$\surd$	$\surd$	x	$\surd$

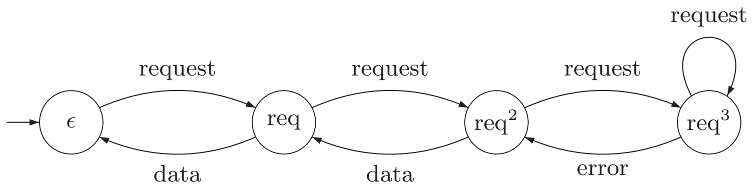

Fig. 5. The further infered black box *Sensor*.

Then the model-checker verifies the new system against the property, finds the counterexample **request request request error pause data send**. Since **request request request error data** is an execution of *Sensor* and **request request request pause** is an execution of *Command* (we test it to ensure this is correct), the grey box checker can safely answer that the model does not satisfy the property. Notice that the black box *Command* was not learnt at all and the black box *Sensor* needed only two learning steps.

6 Testing a Grey Product

A more restrictive scenario is when we can only test whether $w \in \mathcal{B} \times \mathcal{W}$, where $\mathcal{W}$ is the (known) white box and $\mathcal{B}$ is the black box. In what follows, we show how to modify our algorithm for this setting and provide 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 $\mathcal{W}$ 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 $\mathcal{B} \times \mathcal{W} = \mathcal{G}$ as a black box of size at most mn, thus obtaining a $(mn)^2 \times |\Sigma|^{nm}$ upper bound on the number of tests to perform; each of these tests may be of size nm. Clearly, if m is much bigger than n, this approach does not seem attractive.

6.1 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 testing, the number of tests may have to be exponential in the size of the alphabet.

Proposition 4 *For any $n \in \mathbb{N}$, $|\Sigma| = 2t$, 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 $2^{0(n|\Sigma|)}$ tests are needed to distinguish between $\mathcal{B}_r \times \mathcal{W}$ and $\mathcal{B}_{r'} \times \mathcal{W}$.*

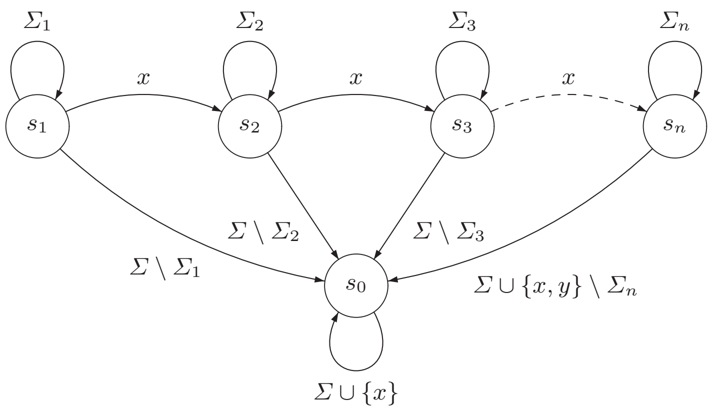

Fig. 6. A black box in $\mathcal{BB}$.

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, $s_n \xrightarrow{a} s_0$ and $s_0 \xrightarrow{a} s_0$ for every $a \neq y$. Every choice of subalphabets $(\Sigma_1, \cdots, \Sigma_n)$ 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, followed by 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 $w_0 \cdots w_t$, $t = n(|\Sigma|/2+1)+1$, that satisfy the following: $w_t = y$, $w_{i(|\Sigma|/2+1)} = x$ for all $i = 1, \ldots, n-1$, and finally, for any

$i \neq 0, 1 \mod |\Sigma/2| + 1$, we have $w_{i-1} \prec w_i$; it is not hard to see that $\mathcal{W}$ can be implemented using $O(n|\Sigma|^2)$ states. □

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 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 $a^* + (a^{r-1})(a^r)^*b$.

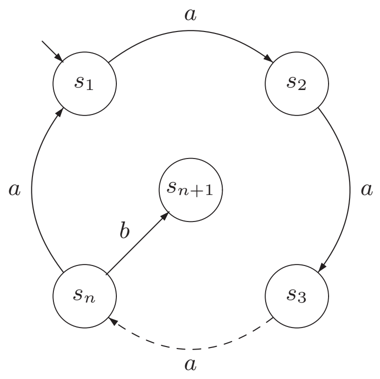

Fig. 7. The automaton $\mathcal{A}_n$.

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 $s \xrightarrow{a} s$. □

6.2 An Almost Optimal Algorithm

To obtain an upper bound that is close to the lower bound of Proposition 4, we consider all automata $\mathcal{B}_i^*$ of size at most n as candidates for the black box.

Proposition 6 *Let $\mathcal{B}$ be an automaton 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 nm.*

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

nm, they are equivalent. Otherwise, they have a distinguishing sequence, i.e., a word w of size at most nm 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 $[\overline{\mathcal{B}_r^*} \cap \mathcal{B}_{r+1}^* \cap \mathcal{W}] \cup [\overline{\mathcal{B}_{r+1}^*} \cap \mathcal{B}_r^* \cap \mathcal{W}]$; 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 nm 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 $2^{n \times |\Sigma| \times \log n}$. □

Although the algorithm of Proposition 6 is nearly optimal in the worst case, it is impractical since it has to test every possible automaton without learning anything before the very last test is performed, i.e., 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 3 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 whose worst case complexity is $O(2^{n \times |\Sigma| \times \log n})$, but the average complexity is likely to be better; the experiments in Section 7 show that this is indeed the case.

Our algorithm uses Angluin's learning algorithm L^* on the grey box; however, instead of using Vasilevskii-Chow algorithm for conformance testing, it uses an oracle that only tests the candidate solution $\mathcal{G}^*$ proposed by L^* on the distinguishing sequences considered in Proposition 6, i.e., those for pairs of automata $\mathcal{B}' \times \mathcal{W}$, $\mathcal{B}'' \times \mathcal{W}$, where $\mathcal{B}'$ is a generated candidate and $\mathcal{B}'' = \mathcal{G}^*$. As in Proposition 6, after each test we eliminate one of the candidates, so there can be at most $2^{n \times |\Sigma| \times \log n}$ such tests; moreover, if $\mathcal{G}^*$ fails some test, we have a discrepancy on $\mathcal{G}^*$ allowing refinement with L^*. Also, 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 $\mathcal{G}^*$.

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 uw is executable in $\mathcal{B} \times \mathcal{W}$, then it is executable in $\mathcal{B}$. If uw is executable in $\mathcal{B} \times \mathcal{W}$, but uwa is not executable in $\mathcal{W}$, then we do not know whether uwa is executable in $\mathcal{B}$. If uwa is executable in $\mathcal{W}$, but not in $\mathcal{B} \times \mathcal{W}$, then we know that it is not executable in $\mathcal{B}$. We generate an automaton by choosing for each state and label 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 $\mathcal{W}$. We define a new equivalence relation on V as $v \sim v'$ iff $white(v) = white(v') = s$ and for all $w \in W_s$, $T(v, w) = T(v, w')$; the closure property is inherited from 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. The complexity of this new L^* algorithm is $O(n^2 m|\Sigma|)$, thus saving us a factor of m.

We present the information known on $\mathcal{B}$ as a list $test = (w, a) :: tail$ with $a \in \Sigma \cup \{\epsilon\}$ and $w \in \Sigma^*$ a sequence with w executable in $\mathcal{B}$. Moreover, if $a \in \Sigma$, wa is not executable in $\mathcal{B}$. First, the generated automaton $\mathcal{A}$ is given by a transition table: $Trans(s, a) = k$ with s a state in $\{1, \cdots, n\}$, a a letter of Σ and $k \in \{0, \cdots, n+1\}$, with $Trans(s, a) = k$ meaning that the transition from s labeled with a leads to k if $k \in \{1, \ldots, n\}$, has not be chosen yet if $k = 0$, and that there are no such transition if $k = n + 1$. We also have a list $ord = (s, a) :: tail$ depicting the order in which we chose the transitions. These two structures are global.

Then, a local copy of a table $fly(s, a) = (j, todo)$ is kept, where $s \in \{1, \cdots, n\}$ is a state, $a \in \Sigma$ a letter and $j \in \mathbb{N}$ is the number of tests waiting for the transition s, a to go on. At last, $todo$ is the list of testing sequences v such that uav belongs to $test$, and u leads from the initial state 1 to the state s in the generated automaton $\mathcal{A}$ (where the transition (s, a) has not be chosen yet). Here is how we initialize fly:

```
function init()=
  fly=new list;
  for all r ∈ {1,...,n}, b ∈ Σ
      fly(r,b) = (0,ε);
  for all (w,a) ∈ test
      if (w == ε)
       {Trans(1,a) = n+1;
         ord = (1,a) :: ord;
       }
      else
       { bu = w;
         (j,todo) = fly(1,b);
         if (ua ≠ ε) todo = (u,a) :: todo;
         fly(1,b) = (++j,todo);
       }
  return fly;
```

The next transition that is chosen is defined by:

```
function choice(fly)=
(s,a) = none,  k = 0;
```

```
 for (r ∈ {0, ···, n}, b ∈ Σ)
    (j, todo) = fly(r, b);
    if j > k {
        (s, a) = (r, b);
        k = j;   }
 endfor
 if ((s, a) == none)
    let (s, a) ∉ ord;
 return (s, a);
```

Here is how we create a new table fly' after a new transition (s, a) is chosen.

```
function update(fly,s,a)=
 fly' = copy(fly);
 (j, todo) = fly(s, a);
 fly'(s, a) = (0, ε);
 if (Trans(s, a) == n + 1) then return fly';
 for every (w, b) ∈ todo
  {
    r = Trans(s, a);
    cu = w;
    while (c ≠ ε and Trans(r, c) ≠ 0)
        { r = Trans(r, c);
          if (r == n + 1) return error;
          cu = u;
        }
    if (c ≠ ε)
        { (j, todo) = fly'(r, c);
          if (ub ≠ ε) todo = (u, b) :: todo;
          fly'(r, c) = (++j, todo);
        }
    else if (b ≠ ε)
        {
          (j, todo) = fly'(r, b);
          if Trans(r, b) ∈ {1, ..., n} or (j > 0)
             return error;
          else
            { Trans(r, b) = n + 1;
              ord = (r, b) :: ord;
            }
        }
 }
return fly';
```

If the procedure returns an error, then we know that any automaton which respects the transitions already defined is distinguished from $\mathcal{B}$, hence we do not need to test any of them. Indeed, it is the case if some test enforces a transition not to exist whereas another test needs this transition to exist.

Here is the main recursive algorithm generate():

```
rec fun generate(fly)=
  if size(ord) = n |Σ|
    { distinguish(Trans);
       return; }
  %% every transition has been affected
  (s,a) = choice(fly);
  (j,todo) = fly(s,a);
  m=1;
  while (m ≤ n or (m == n+1 and j == 0))
    {
      ord = (s,a) :: ord;
      Trans(s,a) = m;
      fly' = update(fly,s,a);
      if (fly' ≠ error) generate(fly');
      loop
        head :: tail = ord;
        ord = tail;
      until (head == (s,a))
      m++;
    }
   Trans(s,a)=0;
   return;
```

6.3 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, $(\mathbf{request}, \mathbf{send}) \in I$. We write $u \equiv_I v$ if there exists a sequence of sequences $u_1, \ldots u_n$ with $u = u_1, v = u_n$ and for all $i < n$, $u_i = xaby$ and $u_{i+1} = xbay$ with $(a,b) \in I$. An equivalence class of $\equiv_I$ is called a *trace* [9]. An important property of the independence relation I is that $\mathcal{G}$ is *closed* under trace equivalence, i.e., $u \in \mathcal{G}$ iff $v \in \mathcal{G}$ for $u \equiv_I v$.

We propose to speed up the conformance algorithm by only considering one sequence in each equivalence class. A natural candidate is the sequence in lexicographic normal form (LNF, defined as follows. Fix some order $\prec$ on the alphabet Σ and extend it to sequences of length n by setting $(u_1 \cdots u_n) \prec (v_1 \cdots v_n)$ if $u_1 \prec v_1$ or $u_1 = v_1$ and $(u_2 \cdots u_n) \prec (v_2 \cdots v_n)$. Then u is said to be in lexicographic normal form if it is minimal among $v \equiv_I u$. It is worth noting that the same idea can be used to improve the learning algorithm; two equivalent (with respect to commutation) states will never be distinguished, hence the tests for one are copied from the other one.

Let LNF_k be the set of words of Σ^k in lexicographic normal form. We can generate LNF_k by performing depth first search on the set of strings in Σ^k, considered in alphabetical order, extending a string by a single letter only if the resulting string is in LNF. Note that for $v \in \Sigma^\star$, we have v is in LNF iff for all decompositions $v = xaybz$ with $ayb \equiv_I bay$, we have $a \prec b$. Hence, a sequence that is not in LNF cannot be a prefix of a sequence in LNF, and we do not need to continue the search from it.

Proposition 7 *Let $v \in \Sigma^\star$. We have v is in LNF iff for all decomposition $v = xaybz$ with $(ay, b) \in I$, we have $a \prec b$.*

Proof. First, if there is such a decomposition of $v = xaybz$, then one can commute ay and b to obtain a word equivalent to v and smaller for $\prec$.

For the converse, consider that v is not in LNF. Let $w \sim v$ be smaller for $\prec$. Let i be the first index such that $w_i \neq v_i$. Then $w_i \prec v_i$. Let j be the index after i such that v_j is the first letter of type w_i. Then $v = xv_iyw_iz$, with $(v_iy, w_i) \in I$. □

To check whether a given letter from Σ can extend a sequence in such a way that it remains in LNF, it suffices to remember the relative order $\lessdot$ of the last appearance of each letter in the current prefix, that is, a permutation of $|\Sigma|$ letters. 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**. We define the function $f(\lessdot, a)$ as $f(\lessdot, a) = 1$ if for all $b \in \Sigma$,either $b \prec a$ or $\exists c$ with $b \lessdot c$ (c can be b) and $(c, a) \notin I$. The property 7 implies that the sequence ua is in lexicographic normal form iff $f(\lessdot, a) = 1$ (notice that if a letter b as not yet been seen in u, then either $b \prec a$ or else $a \prec b$ which means that $b \lessdot a$).

```
LNF(u,⋖):
  If |u| = k then check u and return();
  else, for every b ∈ Σ do
          { if f(⋖,b) then LNF(ub, update(⋖,b)); }
```

Each step of the depth first search costs us $\mathcal{O}(|\Sigma|)$, and hence the set LNF_k can be generated in time $O(k \cdot |\Sigma| \cdot |LNF_k|)$. Since we have to test at least $O(k \cdot |LNF_k|)$ actions, our algorithm is almost optimal.

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. Similarly, in other extreme cases, there can be no reduction at all.

7 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 [14] 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 [8], 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 Angluin's 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, in units (nothing specified) or in millions (M). 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 (Banias) 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 $\mathcal{B}$ and white $\mathcal{W}$ component 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 from 2 to 7 times. 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 speed up 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 real-life 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, over possibly 8^{96} if no information was known.
– When VC is efficient ('leng.' is small), Grey Box Checking is useless (DAS).
– Using both Grey Box Checking and LNF in LNFGBC 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.

8 Checking Sequential Black-Box Components

We discuss in this section how to apply Black Box checking on a sequential system with components that are known and others that are not. More specifically, we consider that the system is given as an automaton with components $c \in C$, and we know a bound on the size n_c of each components $\mathcal{A}_c$. We also denote by m the size of the system $\mathcal{A}$.

Formaly, an automaton with components is a tuple $\mathcal{A} = (S, s_i, s_f, \Sigma, \rightarrow, \lambda)$ where

– S is the finite set of states,
– s_i and S_f are respectively the initial state and the set of final states 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 $t' = t$.
– $\lambda : S \rightarrow \{0, 1\}$ is the function which labels a state by 1 if and only if this state is a component.

We denote by $C = \lambda^{-1}(1)$ the set of components of an automaton with components. A component c of an automaton $\mathcal{A}$ with components is entirely defined by $(\mathcal{A}_c, \mu)$, where $\mathcal{A}_c = (S', s'_i, S'_f, \Sigma, \rightarrow')$ is an automaton and μ is a function which associates each transition $c \xrightarrow{a} t$ with a subset of S'_f that can use this transition. We denote by $\mathcal{A}^0$ the automaton obtained from an automaton $\mathcal{A}$ with components in replacing every component c by the corresponding $\mathcal{A}_c, \mu_c$. Replacing a component c by $(\mathcal{A}_c, \mu_c)$ with $\mathcal{A}_c = (S', s'_i, s'_f, \Sigma, \rightarrow')$ simply means that the set of states is increased by S', the set of transition is increased by $\rightarrow'$ and every transition $t \xrightarrow{a} c$ is replaced by $t \xrightarrow{a} s_i$, and every transition $c \xrightarrow{a} t$ is replaced by $s_f \xrightarrow{a} t$ for $s_f \in \mu_c(s \xrightarrow{a} t)$. Moreover, if c is an initial state of

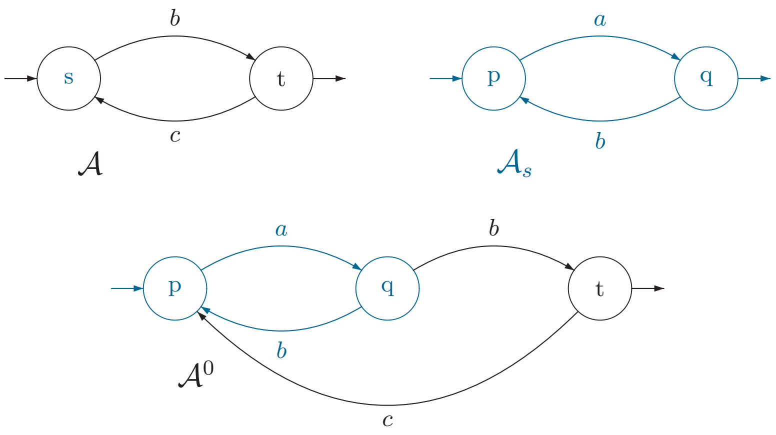

Fig. 8. An example of an automaton $\mathcal{A}$ with a component s, the automaton $\mathcal{A}_s$ associated with s and the automaton $\mathcal{A}^0$ obtained by replacing s by $\mathcal{A}_s$ in $\mathcal{A}$.

$\mathcal{A}$, then s'_i is the initial state of $\mathcal{A}^0$ and if s is a final state of $\mathcal{A}$, then S_f is added to the set of final states. We require that this replacement yields a *deterministic* automaton, that is if s' if a state of $\mathcal{A}_c$ with $s' \in \mu(c \xrightarrow{a} t)$, then there does not exists a transition $s' \xrightarrow{'} at'$ for any t' of $\mathcal{A}_c$.

We define the set of runs of $\mathcal{A}$ as being the set of runs of $\mathcal{A}^0$. In the same line, we define the language $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^0)$ of $\mathcal{A}$ as being the set of sequences $a_1 \cdots a_n$ such that there exists $s_0, \cdots s_n$ with $s_{i-1} \xrightarrow{a_i} s_i$ for all $i > 1$ and $s_o = s_i$. Notice that we do not require that the sequence ends in the final state: we assume that the language is prefix closed. Actually, we use a final state only to specify the end of a component.

Assume that we want to check whether $\mathcal{A}$ models some property φ (φ can be in any formalism for which the model checking is decidable). We want to learn an automaton $\mathcal{A}^*$ such that $\mathcal{L}(\mathcal{A}^*) = \mathcal{L}(\mathcal{A})$. Here, the strategy we adopt for learning $\mathcal{A}^*$ is to learn some automaton $\mathcal{A}^*_c$ for every component c. Implicitly, whenever $\mathcal{A}^*_c$ is defined for every $c \in C$, we call $\mathcal{A}^*$ the automaton obtained after replacing every component $c \in C$ by $\mathcal{A}^*_c$. In the following, we denote by n_c the bound on the size of the component $\mathcal{A}_c$, and by $n = \sum_{c \in \lambda^{-1}(1)}$ the sum of the bounds on the size of each components. We also denote by m the size of the whole system $\mathcal{A}^0$.

8.1 Independant Components

In the case where we can test any black-box component $\mathcal{A}_c$ independently of the rest of the system, it is easy to adapt the Black Box Checking algorithm with a good complexity, as in subsection 5). It suffices to run procedure shown in figure 2, considering that "Checking" runs on the whole system (the one where the black components $\mathcal{A}_c$ are replaced by the candidate automata $\mathcal{A}^*_c$ learned so far). If a correct counter example is found, then we have found a flaw in the design. If no counter example is found, we can still run the conformance test on the component alone. If a spurious counter example $u_1 \cdots u_n$ of the automaton $\mathcal{A} = (S, s_i, S_f, \Sigma, \rightarrow, \lambda)$ with components is found, then it is easy to see that there exist s, i, j with $i < j$ and $s_i \xrightarrow{u_1 \cdots u_i} s$ with $\lambda(s) = 1$, and for instance $u_{i+1} \cdots u_j \in \mathcal{A}_s^\star$ but $u_{i+1} \cdots u_j \notin \mathcal{A}_s$ (or the converse). That is, a spurious counter example means that $\mathcal{L}(\mathcal{A}_s^*) \neq \mathcal{L}(\mathcal{A}_s)$. We thus get a distinguishing sequence from $u_{i+1} \cdots u_j$ for exactly one $\mathcal{A}_s$.

The learning algorithm is thus of total complexity at most $O(\sum_{c \in C} n_c^2 \cdot |\Sigma|)$ and the conformance algorithm is of worst case complexity $O(\sum_{c \in C} n_c^2 \cdot |\Sigma|^{n_c})$.

To go further, the model-checking may also be done for the component and for rest of the system independently, using compositional model-checking [5] based on assume guarantee method. The idea is to break the property φ in two properties $\varphi_\mathcal{A}, \varphi_s$ such that $\mathcal{A} \models \varphi_\mathcal{A}$ and $\mathcal{A}_s \models \varphi_s$ implies that $\mathcal{A}^0 \models \varphi$. The major problem being to find the two properties, [7] propose a novel approach based on learning the property $\varphi_\mathcal{A}, \varphi_s$ (see also [1] for its symbolic implementation). This approach can be used here, since the assume part and the guarantee part are obtained by checking membership to $\mathcal{A}$ and $\mathcal{A}_s$, which we can do.

We can adapt this technique even to some systems where we cannot test each component independently. It suffices to be able to trace in which state the control (e.g. program counter of a program) is in $\mathcal{A}$. If it is in one substate of a compoenent c, then we only need to know that it is somewhere in the component c. We call traceable such a component. We begin by finding the smallest run u_c from the initial state s_i to each component $c \in C$. Notice first that if c is not reachable in our current candidate $\mathcal{A}^*$, then we never have to test c. Next, if we have a path to c (even going through other components from C), then we are sure it is a real path to c since we can trace in which state the ontrol is. We then apply the exact same method as above, except than for testing whether $v \in \mathcal{A}_c$, we test whether $u_c v \in \mathcal{A}$ and the control after $u_c v$ is still in the component c. Else we know that $v \notin \mathcal{A}_c$. We can also use the assume-guarantee technique of [7]. The learning algorithm is thus of total complexity at most $O(m \cdot n \cdot |\Sigma|)$ and the conformance algorithm is of worst case complexity $O(m \cdot n \cdot |\Sigma|^{\max_c(n_c)})$.

8.2 One non Traceable Component

The case where we cannot trace whether the control is in the component or not is more involved. Actually, we show that even in this case, we can use the learning algorithm to learn where the control is *in the case where there is only one component $C = \{c\}$*.

For every state t such that $c \xrightarrow{a} t$ for some $a \in \Sigma$, we add a special state out_t to V. Notice that there is at most $|\Sigma|$ such states out_t. Every membership query u from this state will be checked on the model rather than tested in the black box. That is, $test(out_t u) = 1$ if u is a run from t in $\mathcal{A}$ without accessing c. Moreover, $test(out_t u) = 0$ if v is a run from t in $\mathcal{A}$ without accessing c but vb

is not, with vb a prefix of u and $b \in \Sigma$. In the case where $u = vw$ with $t \xrightarrow{v} v$, then $test(out_t u) = test(w)$.

Notice that since we cannot test the component independently of the system, we still use the trick of the previous section to test a sequence after the shortest path leading from s_i to c. We never consider the successors of any out_a nor of any state equivalent to out_a, such that V never contain any other state of $\mathcal{A}$ than the successors t of c. We use the algorithm Lstar as a learner, but where (V, W) closed means that for every $v \notin \{out_t\}$ and $a \in \Sigma$ s.t. $T(v, a) = 1$, there exists $v' \in V$ with $v' \sim va$. The learning algorithm is of total complexity at most $O((n + |\Sigma|)^2 \cdot |\Sigma|)$ in the number of tests to perform.

The conformance testing is more involved than Vasilevskii-Chow but keeps its philosophy. We define $\mathcal{A}^*$ by replacing c by $\mathcal{A}^*_c$, where $s' \xrightarrow{a} t$ iff $(s'a) \sim out_t$. If $out_t \sim out_{t'}$, then we choose one of t, t'. First, if there exist t, t' with $out_t \sim out_{t'}$, then we compute (if it exists) a minimal sequence distinguishing t and t' in $\mathcal{A}^*$. This sequence is at most of size m. We test if necessary (that is if a factor is a run of the component) that it indeed distinguishes out_t and $out_{t'}$. If it doesnot, it means that we have another discrepency on the black box and can use Lstar to further refine it. In all cases, we have a distinguishing sequence that we add to W. Notice that for every state s of $\mathcal{A}$ not in the component, we can compute the value of $T(s, w)$ for all $w \in W$ without testing it, and hence we can extend $\sim_W$ to every state of $\mathcal{A}^*$. For all $s \sim_W out$, we compute a minimal distinguishing seqeunce in $\mathcal{A}^*$. Again, we test it to ensure that it is indeed distinguishing. If not, we have a discrepency on c that we can feed the learner with. Else, we add that distinguishing sequence to $\mathcal{W}'$ (plus its suffixes). The difference between $\mathcal{W}'$ and $\mathcal{W}$ is that the former is used only in VC while the later is used in both Lstar and VC. If no discrepency is found in this first step, then we look for discrepency between $\mathcal{A}$ and $\mathcal{A}^*$ of the form vuw with $v \in [V/\sim]$, $w \in W \cup W'$ and $|u| \leq n + |\{out_t\}| - |[V/\sim]|$. Compared with learning a black box alone, the exponent is incremented by the number of outputs minus the number of different classes of outputs. It thus gives the following worst case complexity:

Proposition 8 *We can find a discrepency between $\mathcal{A}$ and $\mathcal{A}^*$ in at most $O(m \cdot (n + |\Sigma|) \cdot |\Sigma|^{n+|\Sigma|-1})$ tests if such a discrepency exists.*

Proof. Let u_c leading to the component c in $\mathcal{A}$. We will denote the initial state of c as ϵ since it is represented as ϵ in V. We consider any out_t to be in the same time in c (as out_t) and outside of c as t.

Let us take a smallest word u such that $u_c uw$ is a discrepency between $\mathcal{A}^*$ and $\mathcal{A}$, for some $w \in W \cup W'$. If there is not such a discrepency, then $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^*)$. If the path of $\mathcal{A}$ labeled by u from ϵ is entirely in c, then we can apply VC and get a distinguishing sequence with vuw with $|u| \leq |u| \leq n + |out_t| - |[V/\sim]|$ and end the proof.

Assume by contradiction that it is not the case. We decompose $u = xyz$ with y non empty and z possibly empty if ϵ is never seen again along u. The path labeled by xy leads to the state s_y which is the last ϵ seen (or the last state). That is the path of $\mathcal{A}$ from s_x labeled with y stays outside of c. It is also the case in $\mathcal{A}^*$ since the behavior of $\mathcal{A}, \mathcal{A}^*$ outside of c is the same. In the same way, x leads to the state s_x which is the last out_t seen along u before s_y. We also denote by s_x^*, s_y^* the states reached from ϵ respectively after x and xy in $\mathcal{A}^*$.

If $s_x \not\sim_{W \cup W'} s_x^*$, then $u' = x$ contradicts the minimality of u, with a w' which separate s_x from s_x^*.

If $s_x = s_x^*$, and since yzw is a discrepency between $\mathcal{A}$ and $\mathcal{A}^*$ from s_x, then z is not empty, and $s_y^* = s_y$. Thus $u' = z$ contradicts the minimality of u, with $w = w'$.

Else, $s_x \sim_{W,W'} s_x^*$ but $s_x \neq s_x^*$. If s_x^* is in c, then $s_x^* = out_{t'}, t \neq t'$ is the only possibility. Hence we can consider that s_x^* is outside of c.

We first know that s_x and s_x^* are undistinguishable in $\mathcal{A}^*$ because they are equivalent (else a new sequence would have been added to W' since $s_x = out_t$). However, we know that there is a discrepency between s_x in $\mathcal{A}$ and s_x^* in $\mathcal{A}^*$.

Notice that we need to consider here the states reached from s_x according to the transition of $\mathcal{A}$ but also the one according to the transition of $\mathcal{A}^*$, and also the states reached from s_x^* according to the transition of $\mathcal{A}^*$ but also the one according to the transition of $\mathcal{A}$.

Second, the state reached from s_x after y is the same in $\mathcal{A}$ and $\mathcal{A}^*$. It is thus in both case s_y. Since u is minimal, in particular zw does not distinguish between $\mathcal{A}$ and $\mathcal{A}^*$ from $s_y = \epsilon$ (it is also the case if z is empty). Hence yzw does not distinguish $\mathcal{A}$ and $\mathcal{A}^*$ from s_x.

It means that yzw distinguishes $\mathcal{A}$ and $\mathcal{A}^*$ from s_x^*. Since s_x^* is not in c and the behavior of $\mathcal{A}$ and $\mathcal{A}^*$ is the same out of c, it means that there exists a decomposition $ef = yzw$ such that the path labeled by e from s_x^* leads to ϵ in both $\mathcal{A}$ and $\mathcal{A}^*$. We thus have a distinguishing sequence f smaller than u which contradicts the minimality of u (remember that every suffix of w are also added to $W \cup W'$). Every cases are a contradiction, hence there exists a distinguishing sequence that remains in c. □

9 Conclusion

Black box checking [10] was suggested as a way to directly verify a system when its model is not given but a way of conducting experiments is provided. A related idea, called adaptive model checking [8], allowed using an inaccurate and unupdated model to do the verification, while refining it during verification process. In this paper we studied a related 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 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), while both components are coupled. In this case, 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. A. Groce, D. Peled and M. Yannakakis. Adaptive Model Checking. In *TACAS'02*, LNCS 2280 , p.357-370, 2002.
9. A. Mazurkiewicz, Trace Semantics, Proceedings of Advances in Petri Nets, 1986, Bad Honnef, Lecture Notes in Computer Science, Springer Verlag, 279–324, 1987.
10. D. Peled, M. Vardi and M. Yannakakis. Black Box Checking. In *FORTE/PSTV'99*, 1999.
11. R. Rivest and R. Schapire. Inference of Finite Automata Using Homing Sequences. *Information and Computation*, 103(2), p.299-347, 1993.
12. M.P. Vasilevskii. Failure diagnosis of automata. *Kibertetika*, no 4, p.98-108, 1973.
13. W. Weimer and G. Necula Mining Temporal Specifications for Error Detection. In *TACAS'05*, LNCS 3440, p.461-476, 2005.
14. G. Xie and Z. Dang. Testing Systems of Concurrent Black-boxes - an Automata-Theoretic and Decompositional Approach. In *FATES'05*, LNCS, 2005.