Adaptive Model Checking

Alex Groce[1] and Doron Peled[2] and Mihalis Yannakakis[3]

[1] Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213
[2] Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX, 78712, USA
[3] Avaya Laboratories
233 Mount Airy Road
Basking Ridge, NJ 07920, USA

Abstract. We consider the case where inconsistencies are present between a system and its corresponding model, used for automatic verification. Such inconsistencies can be the result of modeling errors or recent modifications of the system. Despite such discrepancies we can still attempt to perform automatic verification. In fact, as we show, we can sometimes exploit the verification results to assist in automatically learning the required updates to the model. In a related previous work, we have suggested the idea of *black box checking*, where verification starts without any model, and the model is obtained while repeated verification attempts are performed. Under the current assumptions, an existing inaccurate (but not completely obsolete) model is used to expedite the updates. We use techniques from black box testing and machine learning. We present an implementation of the proposed methodology called AMC (for Adaptive Model Checking). We discuss some experimental results, comparing various tactics of updating a model while trying to perform model checking.

Keywords: Automatic Verification, Black Box Testing, Learning Algorithms.

1 Introduction

The automatic verification of systems, also called *model checking*, is increasingly gaining popularity as an important tool for enhancing system reliability. A major effort is to find new and more efficient algorithms. One typical assumption is that a detailed model, which correctly reflects the properties of the original system to be checked, is given. The verification is then performed with respect to this model. Because of the possibility of modeling errors, when a counterexample is found, it still needs to be compared against the actual system. If the counterexample does not reflect an actual execution of the system, the model needs to be refined, and the automatic verification is repeated. A similar iterative process in the framework of abstracted models of systems has been used as a technique for combatting the state space explosion problem [4]. Our technique is substantially different in that rather than using counterexamples to iteratively refine the abstraction used (exposing variables or adding predicates for instance), we use counterexamples to modify incorrect models which are not abstractions of the real system.

Although there are several tools for obtaining automatic translation from various notations to modeling languages, such translations are used only in a small minority of cases, as they are syntax-specific. The modeling process and the refinement of the model are largely manual processes. Most noticeably, they depend on the skills of the person who is performing the modeling, and his experience.

In this paper, we deal with the problem of model checking in the presence of an inaccurate model. We suggest a methodology in which model checking is performed on some preliminary model. Then, if a counterexample is found, it is compared with the actual system. This results in either the conclusion that the system does not satisfy a property, or an automatic refinement of the model. We adapt a learning algorithm [1], to help us with the updating of the model. We employ a testing algorithm [3, 10] to help us compare the model with the actual system, through experiments.

Our adaptive model checking approach can be used in several cases.

- When the model includes a modeling error.
- After some previously occurring bug in the system was corrected.
- When a new version of the system is presented.
- When a new feature is added to the system.

We present an implementation of Adaptive Model Checking (AMC) and experimental results. In the limit, this approach is akin to Black Box Checking [9] (BBC), where initially no model is given. The current implementation serves also as a testbed for the black box checking approach, and we present our experimental results.

The black box checking approach [9] is a strategy to verify a system without a model. According to this strategy, illustrated in Figure 1, we alternate between incremental learning of the system, according to Angluin's algorithm [1], and the black box testing of the learned model against the actual system, using the Vasilevskii-Chou (VC) algorithm [3, 10].

At any stage we have a model that approximates the actual system. We apply model checking to this model. In our case we use the nested depth-first search algorithm to check for emptiness of the product of the system with a Büchi automaton [6]. We can thus handle general LTL properties, under the assumption that the upper bound on the size of the real system is correct. Because the complexity of learning is dependent on the length of the counterexamples generated, we apply iterative deepening to the nested depth-first search. If we find a counterexample for the checked property, we compare it with the actual system. If it turns out to be a false negative, we feed this example to the learning algorithm, since this is an example of the difference between the model and the system. This allows us, through the learning algorithm, to improve the accuracy of the model. If we do not find a counterexample (recall that we use an approximation model for model-checking, and not the system directly), we apply the VC algorithm, looking for a discrepancy between the current approximation model and the system. Again, if we find a sequence that distinguishes the behavior of the system from the model, we feed it to the learning algorithm, in order to improve the approximated model.

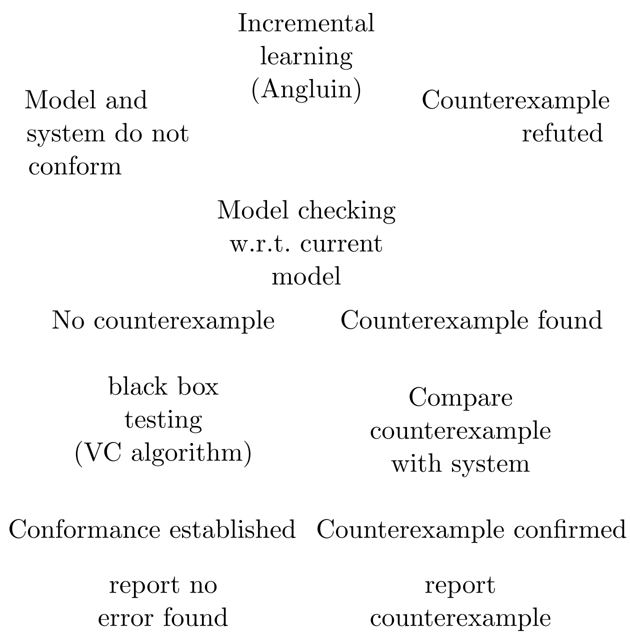

Fig. 1. The black box checking strategy

In this paper, we consider a variant case, in which a model for the tested system is provided, but is inaccurate, due to modeling errors or new updates in the system. Abandoning the model and applying the black box checking approach may not be an efficient strategy due to the inherently high complexity of the black box testing involved. Instead, we attempt to exploit the existing model in order to learn the changes and verify the system. Specifically, we try to diminish the need for the repeated call to the VC algorithm by providing the learning algorithm with initial information taken from the given model. This is in line with our goal of adapting an existing model, as opposed to performing model checking without a model being initially present. We present experimental data that compares the different cases.

2 Preliminaries

A Model and a System

A *model* is a finite automaton $M = \langle S, \iota, \Sigma, \delta \rangle$, where S is the (finite) set of states, $\iota \in S$ is the initial state, Σ is the set of *inputs*, and $\delta \subseteq S \times \Sigma \times S$ is a deterministic *transition relation*. That is, if $(s, a, r), (s, a, r') \in \delta$, then $r = r'$.

A *run* of M is a nonempty sequence $a_1 a_2 \ldots a_n$, such that there exists a sequence $s_0 s_1 \ldots s_n$, where for $i \leq n$, $(s_i, a_i, s_{i+1}) \in \delta$. Similar variants of the definitions also apply to infinite runs. The issue of finite vs. infinite executions is orthogonal to this paper. The reader may refer to [9] for appropriate definitions of a model that allows *infinite* runs and for a way to perform *finite* testing on such models. Let $\mathcal{L}(M)$, the *language* of M, be the set of runs of M. We say that an input a is *enabled* from a state $s \in S$, if there exists $r \in S$, such that $(s, a, r) \in \delta$. Similarly, $a_1 a_2 \ldots a_n$ is enabled from s if there is a sequence of states $s_0, s_2, \ldots, s_n$ with $s_0 = s$ such that for $1 \leq i \leq n$, $(s_{i-1}, a_i, s_i) \in \delta$.

We view a system $\mathcal{S} = (\Sigma, T)$ as a (typically infinite) prefix closed set of strings $T \subseteq \Sigma^*$ over a finite alphabet of inputs Σ (if $v \in T$, then any prefix of v is in T). The strings in T reflect the allowed executions of $\mathcal{S}$.

We assume that we can perform the following *experiments* on $\mathcal{S}$:

– **Reset** the system to its initial state. The current experiment is reset to the empty string ε.
– Check whether an input a can be currently executed by the system. The letter a is added to the current experiment. We assume that the system provides us with information on whether a was executable. If the current *successful part* of the experiment so far was $v \in \Sigma^*$ (i.e., $v \in T$), then by attempting to execute a, we check whether $va \in T$. If so, the current successful part of the experiment becomes va, and otherwise, it remains v.

A model M *accurately models a system* $\mathcal{S}$ if for every $v \in \Sigma^*$, v is a successful experiment (after applying a **Reset**) exactly when v is a run of M. Note that our system generates binary output in accordance with the enabledness of a given input after executing some sequence from the initial state. We can easily generalize the construction and subsequent algorithms to deal with arbitrary output. We deal here with finite state systems, i.e., systems that are accurately modeled by *some* finite state automaton. The *size* of a system is defined to be the number of states of the minimal automaton that accurately models it.

Angluin's Learning Algorithm

Angluin's learning algorithm [1] plays an important role in our adaptive model checking approach. The learning algorithm performs experiments on the system $\mathcal{S}$ and produces a *minimized* finite automaton representing it.

The basic data structure of Angluin's algorithm consists of two finite sets of finite strings V and W over the alphabet Σ, and a table f. The set V is prefix closed (and contains thus in particular the empty string ε). The rows of the table f are the strings in $V \cup V.\Sigma$, while the columns are the strings in W. The set W must also contain the empty string. Let $f(v, w) = 1$ when the sequence of transitions vw is a successful execution of $\mathcal{S}$, and 0 otherwise. The entry $f(v, w)$ can be computed by performing the experiment vw after a **Reset**.

We call the sequences in V the *access* sequences, as they are used to access the different states of the automaton we are learning from its initial state. The sequences in W are called the *separating sequences*, as their goal is to separate between different states of the constructed automaton. Namely, if $v, v' \in V$ lead from the initial state into a different state, than we will find some $w \in W$ such that $\mathcal{S}$ allows either vw or $v'w$ as a successful experiment, but not both.

We define an equivalence relation $\equiv mod(W)$ over strings in Σ^* as follows: $v_1 \equiv v_2\ mod(W)$ when the two rows, of v_1 and v_2 in the table f are the same. Denote by $[v]$ the equivalence class that includes v. A table f is *closed* if for each $va \in V.\Sigma$ such that $f(v, \varepsilon) \neq 0$ there is some $v' \in V$ such that $va \equiv v'\ mod(W)$. A table is *consistent* if for each v_1, $v_2 \in V$ such that $v_1 \equiv v_2\ mod(W)$, either $f(v_1, \varepsilon) = f(v_2, \varepsilon) = 0$, or for each $a \in \Sigma$, we have that $v_1a \equiv v_2a\ mod(W)$. Notice that if the table is not consistent, then there are $v_1, v_2 \in V$, $a \in \Sigma$ and $w \in W$, such that $v_1 \equiv v_2\ mod(W)$, and exactly one of v_1aw and v_2aw is an execution of $\mathcal{S}$. This means that $f(v_1a, w) \neq f(v_2a, w)$. In this case we can add aw to W in order to separate v_1 from v_2.

Given a closed and consistent table f over the sets V and W, we construct a *proposed automaton* $M = \langle S, s_0, \Sigma, \delta\rangle$ as follows:

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

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

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

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

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

The Angluin algorithm is executed in phases. After each phase, a new proposed automaton M is generated. The proposed automaton M may not agree with the system $\mathcal{S}$. We need to compare M and $\mathcal{S}$ (we present later a short description of the VC black box testing algorithm for performing the comparison). If the comparison succeeds, the learning algorithm terminates. If it does not, we obtain a run σ on which M and $\mathcal{S}$ disagree, and add all its prefixes to the set of rows V. We then execute a new phase of the learning algorithm, where more experiments due to the prefixes of σ and the requirement to obtain a closed and consistent table are called for.

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

Fig. 2. An incremental learning step

The subroutine in Figure 2 is an incremental step of learning. Each call to this subroutine starts with either an empty table f, or with a table that was prepared in the previous step, and a sequence σ that distinguishes the behavior of the proposed automaton (as constructed from the table f) and the actual system. The subroutine ends when the table f is closed and consistent, hence a proposed automaton can be constructed from it.

Let m be the size of an automaton that faithfully represents the system $\mathcal{S}$. Assume that Angluin's algorithm is executed in such a way that each time an automaton that does not faithfully represents the system $\mathcal{S}$ is proposed, a shortest counterexample showing the discrepancy in behavior is presented, without accounting for the time it takes for calculating such a counterexample. This assumption is made in order to decouple the complexity of comparing $\mathcal{S}$ with M from the learning algorithm. Then, the time complexity is $\mathcal{O}(m^4)$.

We do not in practice perform the expensive breadth-first search required to copute the shortest counterexample, but we do apply iterative deepening to the nested depth-first search in order to avoid its preference for very long paths and cycles.

Spanning Trees

A *spanning tree* of an automaton $M = \langle S, \iota, \Sigma, \delta \rangle$ is a graph $G = \langle S, \iota, \Sigma, \Delta \rangle$ generated using the following depth first search algorithm.

explore(ι);
subroutine *explore*(s):
    set *old*(s);
    for each $a \in \Sigma$ do
        if $\exists s' \in S$ such that $(s, a, s') \in \delta$
                and $\neg old(s')$ /* s' was not found yet during the search */
            add (s, a, s') to Δ;
            explore(s');

A spanning tree thus is a subgraph G of M, with no cycles. Let T be the corresponding runs of G. Notice that in Angluin's algorithm, when a proposed automaton M is learned, the set V of access sequences includes the runs of a spanning tree of M.

Separating Sequences

Let $M = \langle S, \iota, \Sigma, \delta \rangle$ be an automaton with a set of states S. Let ds be a function $ds : S \to 2^{\Sigma^*}$. That is, ds returns, for each state S, a set of words over Σ. We require that if $s, s' \in S$, $s \neq s'$, then there are $w \in ds(s)$ and $w' \in ds(s')$, such that some $\sigma \in prefix(w) \cap prefix(w')$ is enabled from exactly one of s and s'. Thus, σ separates s from s'. We call ds the *separation function* of M (see, [8]).

A simple case of a separation function is a constant function, where for each s, s', $ds(s) = ds(s')$. In this case, we have *separation set*. Note that the set W generated by Angluin's algorithm is a separation set. We denote the (single) set of separating sequences (a *separation set*) for an automaton M by $DS(M)$.

The Hopcroft algorithm [7] provides an efficient $\mathcal{O}(n \log n)$ for providing a set of separating sequences, where n is the number of states.

Black Box Testing

Comparing a model M with a finite state system $\mathcal{S}$ can be performed using the Vasilevskii-Chow [10, 3] algorithm. As a preparatory step, we require the following:

- A spanning tree G for M, and its corresponding runs T.

– A separation function ds, such that for each $s \in S$, $|ds(s)| \leq n$, and for each $\sigma \in ds(s)$, $|\sigma| \leq n$.

Let $\Sigma^{\leq k}$ be all the strings over Σ with length smaller or equal to k. Further, let m be the number of states of the automaton M. We do the experiments with respect to a conjectured maximal size n of $\mathcal{S}$. That is, our comparison is correct as long as representing $\mathcal{S}$ faithfully (using a finite automaton) does not need to have more than n states. The black box testing algorithm prescribes experiments of the form **Reset** $\sigma\,\rho$, performed on $\mathcal{S}$, as follows:

– The sequence σ is taken from $T.\Sigma^{\leq n-m+1}$.
– Run σ from the initial state ι of M. If σ is enabled from ι, let s be the state of M that is reached after running σ. Then ρ is taken from the set $ds(s)$.

The complexity of the VC algorithm is $\mathcal{O}(n^2\, m\, |\Sigma|^{n-m+1})$.

3 Adaptive Verification

Our adaptive model checking methodology is a variant of black box checking. While the latter starts the automatic verification process without having a model, adaptive model checking assumes some initial model, which may be inaccurate. The observation is that the inaccurate model is still useful for the verification. First, it can be used for performing model checking. Caution must be taken as any counterexample found must still be compared against the actual system; in the case that no counterexample is found, no conclusion about the correctness of the system can be made. In addition, the assumption is that the given model shares some nontrivial common behavior with the actual system. Thus, the current model can be used for obtaining a better model.

The methodology consists of the following steps.

1. Perform model checking on the given model.
2. Provided that an error trace was found, compare the error trace with the actual system. If the trace involves a cycle, the cycle must be repeated a number of times equal to the upper bound given for the real size of the system. If this is an actual execution of the system, report it and stop.
3. Start the learning algorithm. Unlike the black box checking case, we do not begin with $V = W = \{\varepsilon\}$. Instead, we initiate V and W to values obtained from the given model M as described below. We experiment with several ways of doing so.
4. If no error trace was found, we can either decide to complete the verification attempt (assuming that the model is accurate enough), or perform some black box testing algorithm, e.g., VC, to compare the model with the actual system. A manual attempt to correct or update the model is also possible. Notice that black box testing is a rather expensive step that should be eliminated.

In the black box checking algorithm, we start the learning with an empty table f, and empty sets V and W. This immediately cause the initialization of $V = W = \{\varepsilon\}$ (see Figure 2). As a result, the black box checking algorithm alternates between the incremental learning algorithm and a black box testing (VC algorithm) of the proposed automaton with the actual system. Applying the VC algorithm may be very expensive. In the adaptive model checking case, we try to guide the learning algorithm using the already existing (albeit inaccurate) model. We assume that the modified system has a nontrivial similarity with the model. This is due to the fact that changes that may have been made to the system were based on the old version of it. We can use the following:

1. A false negative counterexample σ found (i.e., a sequence σ that was considered to be a counterexample, but has turned out not to be an actual execution of the system $\mathcal{S}$). We perform learning experiments with *prefix*(σ), i.e., the set of all prefixes of σ.
2. The runs T of a spanning tree G of the model M as the initial set of access sequences V. We precede the learning algorithm by performing `for each` $v \in T$ `do` *add_rows*(v)`.`
3. A set of separating sequences $DS(M)$ calculated for the states of M as the initial value of the set W. Thus, we precede the learning algorithm by setting f to be empty, and $W = DS(M)$.

Thus, we attempt to speed up the learning, using the existing model information, but with the learning experiments now done on the actual current system $\mathcal{S}$. We experiment later with the choices $1+2$ (in this case we set $W = \{\varepsilon\}$), $1+3$ (in this case we set $V = \{\varepsilon\}$) and $1+2+3$.

In order to justify the above choices of the sets V and W for the adaptive model checking case, we will show the following: If the model M accurately models a system $\mathcal{S}$, starting with the aforementioned choices of V and W the above choices allow Angluin's algorithm to learn M accurately, without the assistance of the (time expensive) black box testing (the VC algorithm).

Theorem 1. *Assume that a finite automaton M accurately models a system $\mathcal{S}$. Let G be a spanning tree of M, and T the corresponding runs. If we start Angluin's algorithm with $V = T$ and $W = \{\epsilon\}$, then it terminates learning a minimized finite automaton A with $\mathcal{L}(A) = \mathcal{L}(M)$. Moreover, the learning algorithm will not require the use of the black box testing.*

Sketch of proof. By induction on the length of experiment that is required to distinguish pairs of states of M. As the induction basis, by consistency, we will separate states in V according to whether va can be accessed from the initial state or not, for $v \in V$, $a \in \Sigma$. Then, suppose that the states reached by va and $v'a$ were separated. The table cannot become consistent before we separate va and $v'a$. □

Theorem 2. *Assume that a finite automaton M accurately models a system $\mathcal{S}$. Let $DS(M)$ be a set of separating sequences for M. If we start Angluin's algorithm with $V = \{\varepsilon\}$ and $W = DS(M)$, then it terminates learning a minimized* *finite automaton A with $\mathcal{L}(A) = \mathcal{L}(M)$. Moreover, the learning algorithm will not require the use of the black box testing.*

Sketch of Proof. Because of the selection of the separation set, each time a new state of M is accessed through an experiment with a string $v \in V$, it will be immediately distinguished from all existing accessed states. Consequently, by the requirement that the table will be closed, the learning algorithm will generate for it a set of immediate successors. Thus, the table f will not be closed before all the states of M are accessed via experiments with strings of V. $\square$

The above theorems show that the given initial settings do not prevent us from learning correctly any correct finite representation of $\mathcal{S}$ (note also that adding arbitrary access and separating sequences does not affect the correctness of the learning algorithm). Of course, when AMC is applied, the assumption is that the system $\mathcal{S}$ deviates from the model M. However, if the changes to the system are modest, the proposed initial conditions are designed to speed up the adaptive learning process.

4 An Implementation

Our implementation of AMC is described in this section. We provide some experimental results.

Experimental Results

Our implementation prototype is written is SML (Standard ML of New Jersey) and includes about 5000 lines of code. We have performed several experiments with our AMC prototype. We compared adaptive learning (AMC) and black box checking (BBC). In addition, we compare the behavior of different initial values with which we started the AMC. In particular, we experimented with starting AMC with a spanning tree T of the current model M, a set of distinguishing sequences $DS(M)$, or with both. In each case of AMC, the prefixes of the counterexample that was found during the verification of the given property against the provided, inaccurate, model was also used as part of the initial set of access sequences V.

The examples used in our experimental results are taken from a CCS model of a COMA (Cache Only Memory Architecture) cache coherence protocol [2]. We use the more recent CCS model obtained from the site

ftp.sics.se/pub/fdt/fm/Coma

rather than that in the paper; we also modify the syntax to that of the Concurrency Workbench [5]. We used the Concurrency workbench in order to convert the model into a representation that we can use for our experiment.

The model is of a system with three components: two clients and a single directory. The system $S2$ with 529 states is a set of processes to generate local read and write requests from the client. The system $S3$ with 136 states, allows observation only of which processes have valid copies of the data and which (if any) have write access. (We preserved the names $S2$ and $S3$ from the paper [2]).

Property φ_1 asserts that first the component called 'directory ' has a valid copy, then clients 1 and 2 alternate periodically without necessarily invalidating the data that any of the others hold. (The directory is the interface between the two memory units in the cache protocol. COMA models basically have only a cache to handle memory.) Property φ_2 describes a similar property but the traces now concern a cache having exclusivity on an entry (a cache can have a valid copy without exclusivity, which is more involved to obtain). For AMC we have selected properties that do not hold, and tampered with the verified model in order to experiment with finding (false negative) counterexamples and using them for the adaptive learning.

The next table summarizes the experiments done on $S2$. The columns marked BBC correspond to the black box checking, i.e., learning from scratch, while the rightmost column correspond to the three different ways in which the learning algorithm was initialized for the adaptive learning case. The notation $\varphi_1 \gg \varphi_2$ means that the experiment included checking φ_1 first, and then checking φ_2. In the black box checking this means that after a counterexample for φ_1 is found (which is intended to be the case in our experiments), we continue the incremental learning algorithm from the place it has stopped, but now with φ_2 as the property. This possibly causes continuing the incremental learning process for the proposed model automata, and performing the VC algorithm several times. In the adaptive case, it means that we initialize AMC with the information about the previously given model, according to the three choices. The memory and time measurements for these cases are the total memory and time needed for completing the overall checking of φ_1 *and* φ_2.

In the tables, time is measured in seconds, and memory in megabytes. The experiments were done on a Sun Sparc Ultra Enterprise 3000 with 250Mhz processors and 1.5 gigabytes of RAM.

Property	BBC		$V \neq \{\varepsilon\}$		$W \neq \{\varepsilon\}$		$V, W \neq \{\varepsilon\}$	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
φ_1	1234	31	423	41	682	32	195	37
φ_2	934	31	424	45	674	42	198	42
$\varphi_1 \gg \varphi_2$	1263	31	454	45	860	44	227	47
$\varphi_2 \gg \varphi_1$	1099	31	453	45	880	40	227	44

The following table includes the *number of states* learned in the various experiments, and the *length of the counterexample*.

Property	BBC		$V \neq \{\varepsilon\}$		$W \neq \{\varepsilon\}$		$V, W \neq \{\varepsilon\}$	
	States	Len	States	Len	States	Len	States	Len
φ_1	258	90	489	211	486	211	489	211
φ_2	174	113	489	539	486	539	489	539
$\varphi_1 \gg \varphi_2$	274	112	489	539	486	539	489	539
$\varphi_2 \gg \varphi_1$	259	160	489	211	486	211	489	211

The next table includes similar time and memory measurement experiments performed with the system $S3$:

Property	BBC		$V \neq \{\varepsilon\}$		$W \neq \{\varepsilon\}$		$V, W \neq \{\varepsilon\}$	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
φ_1	913	24	14	25	13	24	7	25
φ_2	13917	26	14	25	14	25	7	25
$\varphi_1 \gg \varphi_2$	1187	27	17	25	19	26	10	25
$\varphi_2 \gg \varphi_1$	13873	27	17	26	19	25	10	25

Similarly, the following table includes the number of states and length of counterexample for the experiments with $S3$.

Property	BBC		$V \neq \{\varepsilon\}$		$W \neq \{\varepsilon\}$		$V, W \neq \{\varepsilon\}$	
	States	Len	States	Len	States	Len	States	Len
φ_1	79	25	134	114	135	114	134	114
φ_2	108	118	134	142	135	142	134	142
$\varphi_1 \gg \varphi_2$	81	94	134	142	135	142	134	142
$\varphi_2 \gg \varphi_1$	114	113	134	114	135	114	134	114

In addition, we performed sanity checks. We applied AMC with the three different initializations on $S2$ and $S3$, and checked that we indeed obtained automata with 136 and 529 states, respectively. It should be commented that the deliberate change that was made to the original systems of $S2$ and $S3$ has resulted in no change in the number of states (in the minimal representation) of these systems.

Observing the tables, we see that performing BBC, i.e., learning a model from scratch, was 2.2 to 100 times slower than AMC. In addition, BBC has in some cases learned a model that is less than half of the actual states of the minimal automaton that faithfully represents the system (after the modification), while AMC was able to generate a representation that is less than 50 states short. It turned out that for the smaller system, $S3$, BBC has done a better job in learning a model than for $S2$. This means that it got a model with number of states closer to the actual minimal representation of the system.

We also see that the counterexample for BBC is shorter than that of AMC. This is not surprising, as BBC is 'zooming into' an error by considering incrementally growing automata for representing the system, while AMC is attempting to obtain a close enough representation first.

We comment that the implementation was done using SML, which takes about 20 megabytes for keeping its internal runtime structures. SML performs garbage collection phases during the execution, which slightly affects the running time and memory usage.

Improving the Prototype

Note that there is no guarantee that the adaptive model checking will zoom into a correct model by performing the learning algorithm. After the learning algorithm terminates, it is still possible that discrepancies exist, and to detect them we need to apply the black box testing part and then resume the learning. Of course, it is beneficial to avoid the testing part, in particular for relatively large models, as much as possible. For that, we may enhance the learning part with various heuristics. For example, we start AMC in Section 3 assuming that the actual structure of $\mathcal{S}$ would resemble a model M immediately after resetting $\mathcal{S}$. This does not need to be the case. Thus, we may look for behaviors that match or resemble the set of runs T of a spanning tree of the model M from other points in the execution of $\mathcal{S}$. For example, we may augment the learning algorithm by looking forward two or three inputs from every given state, and try to pattern match that behavior with that of set of runs T.

The Vasilevskii-Chow algorithm, used to compare the system with a model, is a bottleneck in our approach. In the limit, when there is no error, the algorithm is exponential in the difference between the conjectured size of the system and its actual size.

We apply the following heuristic improvement. The most wasteful part of the algorithm is manifested when arbitrary sequences of inputs over the input alphabet Σ (of length $n - m + 1$) are required by the algorithm. We generate a smaller set of sequences as follows. Given some information about the inputs, we calculate a partial state and carry the updating of the current state with the generation of the sequence. For example, if we know that some of the inputs correspond to message passing, we may include with each partial state a counter for each message queue. Such a counter will be set to zero in the initial state and will increment or decrement according to the corresponding send and receive events, respectively. Thus, from a current state where a counter that corresponds to some queue is zero, we do not allow an input that corresponds to a receive event.

5 Discussion

Our adaptive model checking approach is applicable for models that are inaccurate (but not completely irrelevant). When a principle change is made, the approach will still work, but the time to update the model may be substantial. In some pathological cases, simple changes can also lead to a substantial update effort. In particular, the following change to a system provides a 'worst case' example: The system functionality is not being changed, except for adding some security code that needs to be input before operating it.

The main problem we have dealt with is the ability to update the model according to the actual system, while performing full LTL model checking. While the changes learned may not fully reflect corresponding changes in the actual system $\mathcal{S}$, the obtained model may still be useful for verification.

We have compared two approaches: one of abandoning the existing model in favor of learning a finite state representation of the system $\mathcal{S}$ from scratch (BBC). The other one is using the current model to guide the learning of the potentially modified system (AMC). We argue that there are merits to both approaches. The BBC approach can be useful when there is a short error trace that identifies why the checked property does not work. In this case, it is possible that the BBC approach will discover the error after learning only a short proposed model. The AMC approach is useful when the modification of the system is simple or when it may have a very limited affect on the correctness of the property checked.

References

1. D. Angluin, Learning Regular Sets from Queries and Counterexamples, Information and Computation, 75, 87–106 (1987).
2. G. Birtwistle, F. Moller, Ch. Tofts, The verification of a COMA cache coherence protocol, IEEE Workshop on Formal Methods in Software Practice (FMSP'96).
3. T. S. Chow, Testing software design modeled by finite-state machines, IEEE transactions on software engineering, SE-4, 3, 1978, 178–187.
4. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, Counterexample-guided abstraction refinement, CAV 2000, 154–169.
5. R. Cleaveland, J. Parrow, B. Steffen, The Concurrency Workbench: a semantic-based tool for the verification of concurrent systems, TOPLAS 15(1993), 36–72.
6. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M., Memory efficient algorithms for the verification of temporal properties, Formal Methods in System Design 1(1992), pp. 275–288.
7. J.E. Hopcroft, An n log n algorithm for minimizing the states in finite automata, The theory of Machines and Computation, Academic Press, New York, 189–196, 1971.
8. D. Lee, M. Yannakakis, Principles and methods of testing finite state machines - a survey, Proceedings of the IEEE, 84(8), 1090–1126, 1996.
9. D. Peled, M. Y. Vardi, M. Yannakakis, Black Box Checking, Black Box Checking, FORTE/PSTV 1999, Beijing, China.
10. M. P. Vasilevskii, Failure diagnosis of automata, Kibertetika, no 4, 1973, 98–108.