AMC: An Adaptive Model Checker

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

[1] Computer Science Department, Carnegie Mellon University
[2] Department of Elec. and Comp. Eng., University of Texas at Austin
[3] Avaya Laboratories

Abstract. The AMC (for adaptive model checking) system allows one to perform model checking directly on a system, even when its internal structure is unknown or invisible. It also allows one to perform model checking using an inaccurate model, incrementally improving the model each time that a false negative (i.e., not an actual) counterexample is found.

1 Introduction

Inconsistencies are often present between a system and a corresponding finite state model that is used for the verification of the system. Such inconsistencies can be the result of modeling (or implementation) errors or changes made to the system after the model was finalized. Our tool, AMC, attempts to perform automatic verification of full LTL properties despite such discrepancies. Previous work proposed the idea of *black box checking* (BBC) [6], in which the verification is performed directly on the system, without a model being presented in advance. Moreover, in BBC we may not have access to the internal structure of the system and may not be able to record the different states while performing the verification; we are restricted to performing interactions with the system, from a given predefined vocabulary (which must include a reliable *reset* action that returns the system to its initial state from any other state). We have also extended this framework to *adaptive model checking* (AMC) [5], where rather than beginning from scratch, we may use finite state learning algorithms to improve the accuracy of a possibly faulty model.

AMC takes as input a (possibly erroneous) model of a system and an LTL property or specification automaton. We currently handle, in addition to the format used by AMC itself, input of models or specification automata produced by the Concurrency Workbench [3]. AMC must also be equipped with an interface to the actual system. AMC alternates between model checking runs on the current model, and incremental learning for improving the model (see Figure 1). The usual output of model checking is either a counterexample for the given property or a statement that no error was found. In our system, we may also use the model and a false negative counterexample generated during verification to improve the model used for verification. The learning algorithm of Angluin [1] and the conformance testing of Vasilevskii and Chow [2, 7] (VC) are used in order to compare the given model to the actual system and improve it if a discrepancy occurs.

The AMC system fills the gap between verifying a model of the system (as is done usually in model checking), and the direct verification of finite state systems [4, 6]. The user can either start with an empty model, in which case the system will attempt to learn the model while performing the verification, or enter a model that may be an approximation of the actual system.

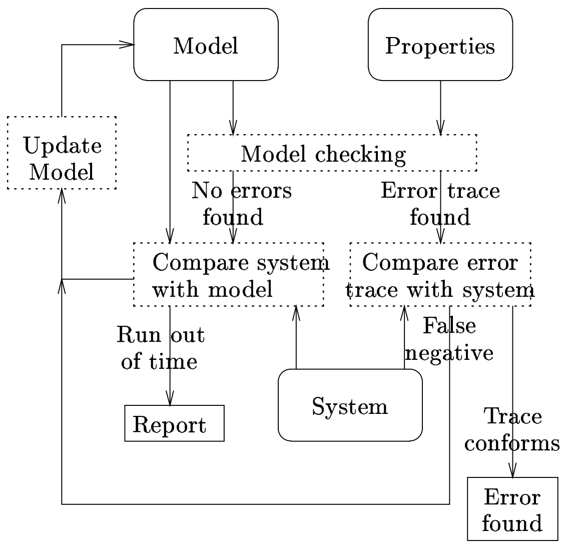

Fig. 1. The adaptive model checking strategy

2 The Principle of AMC

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 a finite state 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 different states, 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 $v_1 a \equiv v_2 a \ mod(W)$. Notice that if the table is not consistent, then there are $v_1, v_2 \in V$, $a \in \Sigma$ and $w \in W$, such that $v_1 \equiv v_2 \ mod(W)$, and exactly one of $v_1 aw$ and $v_2 aw$ is an execution of $\mathcal{S}$. This means that $f(v_1 a, w) \neq f(v_2 a, w)$. 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 **Reset** $v\,a\,w$.

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 **Reset** $v\,w$.

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.

The subroutine in the Angluin learning algorithm 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.

Black Box Testing. Comparing a model $M = (S, s_0, \Sigma, \delta)$ with a finite state system $\mathcal{S}$ can be performed using the Vasilevskii-Chow [7, 2] algorithm. As a preparatory step, we require the following:

- A spanning tree $G : S \to 2^{\Sigma^*}$ for M, and its corresponding runs T.

– A *separation function* ds. That is, for every pair of distinct states $s, s' \in S$, we have that $ds(s) \cap ds(s')$ contains at least one sequence that is enabled from exactly one of s or s'. Furthermore, 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 $\mathbf{Reset}\, \sigma\, \rho$, performed on $\mathcal{S}$, as follows:

– The sequence σ is taken from $T.\Sigma^{\leq n-m+1}$.
– Run σ from the initial state s_0 of M. If σ is enabled from s_0, let s be the state of M that is reached after running σ. Then ρ is taken from the set $ds(s)$.

Adaptive Model Checking. The BBC algorithm involves applying Angluin's algorithm until a candidate model is found. Then this model is used for model checking against the given LTL specification. If a counterexample is found, it is compared against the actual system $\mathcal{S}$. If this is found to be an actual execution of $\mathcal{S}$, an error is reported. Otherwise, Angluin's algorithm is executed again, starting with the counterexample as a first experiment (since the counterexample separates the behavior of the candidate model and the actual system). If no counterexample is found, we use the VC algorithm to compare the model and the system and upon finding a discrepancy use it as a new experiment in a new iteration of Angluin's algorithm. This iterative process terminates when a true counterexample is found, or when the VC algorithm finds no difference between the model and the actual system.

Since the complexity of the VC algorithm is prohibitive when dealing with large systems, we try to eliminate its use as much as possible. The AMC strategy starts with an estimated model and, provided that a false negative counterexample is found, attempts to refine it. The Angluin algorithm is started, but not from scratch; we generate a separation function for the given model M and use the union of its sequences as the initial value for the separating sequences in W. We also generate a spanning tree for M, and use its sequences as the initial value of the access sequences V. In this fashion, if the model is not changed (as shown in [5], we can obtain it back without calling the VC algorithm. We have performed experiments in which this strategy was capable of changing the model in such a way that actual errors could be found [5].

3 The AMC System

The AMC system is written in Standard ML of New Jersey (SML). It includes around 5000 lines of code. It allows three different modes of interfacing with the actual system:

1. Each process of the system is an independent Unix process, written in C, and the AMC system is another process that interacts with all the other

processes. Inspired by the Verisoft system [4], we observe the interprocess communication operations. We replace each communication operation (and any other operation that can affect the checked property, such as a nondeterministic choice or a timeout) with a macro expansion that interacts with the AMC verifier using a shared file.
2. The direct verification of a system. The system is a single Unix process, or an external device whose input and (binary) output can interface with our AMC system. We interact with a system using a Unix bidirectional pipeline, which in one direction emits the sequence of inputs (or a **Reset**), and in the other direction waits for a response of 0 or 1 (for *enabled*, or *disabled*, respectively).
3. The verified system is a single SML process, which is compiled together with our AMC system. The SML interface to foreign code could also be used in this case.

The above modes of operation of AMC lend themselves to different capabilities and difficulties. Mode 1 allows one to check C programs directly, without modeling them (or verifying a model while improving its accuracy against actual C code). The speed of the execution is highly affected by the frequent interprocess interaction. Each communication between processes is replaced with a sequence of commands in which the communicating process notifies the AMC process about its intention to communication, and preempts itself until the communication is scheduled.

Mode 2 allows us to interact with an external device directly and perform the verification or the update on an actual device. The speed of the interaction depends on the actual device involved and the speed of using (Unix) pipelines. Since a physical device is involved, the automatic scheduling of the AMC by the operating system is also an important factor.

Mode 3 involves only one process and hence reflects the actual speed of the adaptive model checking algorithm. As was shown in [5], this may be, in the worst case of verification without a model, exponential time in the (estimated minimized) size of the system, although the average case complexity of finding errors in a system (if they exist) is polynomial.

References

1. D. Angluin, Learning Regular Sets from Queries and Counterexamples, Information and Computation, 75, 87–106 (1987).
2. T. S. Chow, Testing software design modeled by finite-state machines, IEEE transactions on software engineering, SE-4, 3, 1978, 178–187.
3. R. Cleaveland, J. Parrow, B. Steffen, The Concurrency Workbench: a semantic-based tool for the verification of concurrent systems, TOPLAS 15(1993), 36–72.
4. P. Godefroid, Model checking for programming languages using VeriSoft, Proc. 24th ACM Symp. on Progr. Lang. and Sys., 174-186, 1996.
5. A. Groce, D. Peled, M. Yannakakis, Adaptive Model Checking, TACAS 2002, LNCS 2280, Springer, 357–370.
6. D. Peled, M. Y. Vardi, M. Yannakakis, Black Box Checking, FORTE/PSTV 1999, Beijing, China, 225–240.
7. M. P. Vasilevskii, Failure diagnosis of automata, Kibertetika, no 4, 1973, 98–108.