Dynamic Model Learning Using Genetic Algorithm under Adaptive Model Checking Framework *

Zhifeng Lai, S.C. Cheung †
Department of Computer Science,
Hong Kong University of Science and Technology, Hong Kong
{zflai, scc}@cs.ust.hk

Yunfei Jiang
Software Institute,
Zhongshan (Sun Yat-sen) University,
Guangzhou, P. R. China
lncsri05@zsu.edu.cn

Abstract

Model-based techniques for reactive systems generally assume the availability of a state machine that describes the behavior of the system under study. However, the assumption may not always hold in reality. Even the assumption holds, the state machine could be invalidated when the system evolves. This triggers the study of adaptive model checking, which necessitates an iterative construction of a state machine for a system. In this paper, we propose a dynamic learning approach based on genetic algorithm to iteratively generate a finite-state automaton from a given system. In view of the fact that modern systems are apt to change, our algorithm postpones expensive equivalence checking until the associated accuracy is required for the verification of some properties. We explain in details the core learning process of our algorithm, including encoding the model and its synthesis from a given training set. Experimental results show that our algorithm is scalable in memory consumption. Dynamic model learning technique helps model checking of evolving reactive system.

1 Introduction

Model-based techniques, such as model checking, model-based testing, have become active research topics in the last decades for specification and verification of systems. For example, Finite State Machines (FSM) are widely used models for reactive systems. One typical assumption of these techniques is that a formal model, which correctly reflects the behaviors of the original system under study, is available. However, in many situations, these formal abstractions do *not* exist or are *invalidated* when the systems have evolved. Constructing the model by hand and subsequent updates are tedious. This procedure is error-prone and largely depends on the skills and experience of the engineers who perform the modelling. Even if such manually built model is present, it is unclear whether the model corresponds to the current system.

Now, consider the following problem. If we confront with a system whose internal structure is totally unknown, how can we check some properties of this black box? Since some interesting reactive systems can be modelled by Deterministic Finite State Automata (DFA), various methods [1, 2, 11, 16] have been proposed to learn DFAs theoretically. Most noticeably, Angluin's algorithm [1] has been referred to in many literatures, which combines it with other model-based techniques to enable model learning in practice. Two examples are Black Box Checking [15] and Adaptive Model Checking [5, 6].

Under the Adaptive Model Checking (AMC) framework, we propose an approach to learn DFA dynamically using genetic algorithm. It can either build a DFA from scratch, or iteratively improve the accuracy of the model. Static learning approaches tend to learn a "totally" correct model at one trial, which is usually a long time process. The problem is, modern systems are apt to change frequently. When it happens, the previous expensive computing efforts are very likely to become in vain. On the contrary, we iteratively refine the model in the spirit of "Driven On-Demand". Initially, we generate a training set to learn a rough model in order to enable its rapid usage. And each time, when model checking requests are submitted, we either refine the model or report a counterexample reflecting the absence of certain properties of the system, distributing the whole refinement process in later phase as the same manner as AMC.

*This research is partially supported by a grant from the Research Grants Council of Hong Kong (Project No. HKUST6167/04E) and the National Grand Fundamental Research 973 Program of China under Grant No.2006CB303000.

†All correspondence should be addressed to S.C. Cheung at the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. Tel: (+852) 23587016. Fax: (+852) 23581477. Email: scc@cs.ust.hk.

Our work distinguishes from AMC in that if no initial approximate model is given, we can quickly produce such an approximation, while it is not the case in AMC. We have realized the core algorithm of this approach, namely representing DFAs, choosing appropriate genetic operators, and optimizing the learning process. Our paper focuses on enhancing the practicality of model learning from real-life systems and hence automatically checks properties of finite state systems whose structure are unknown.

In the next section, we overview some related work of fundamental model learning algorithms and some composite learning approaches. In section 3, we recall some basic automata theory and introduce our learning approach. Experiments are described and discussed in section 4.

2 Related Work

From a top level view, typical model learning algorithm tries to construct a model $\mathcal{A}$ that is approximate to the underlying ideal model $\mathcal{M}$, which can actually describe the correct behavior of the real system. Generally, two auxiliary sources are required for generating such an approximation: (1) *Membership Oracle:* answering whether a word is accepted by $\mathcal{M}$ or not with **true** or **false**; (2) *Equivalence Oracle:* answering whether the learned model $\mathcal{A}$ is equivalent to $\mathcal{M}$ with **true** or counterexamples. In reality, the system performs the role of membership oracle by actually executing the input words. And a VC algorithm [4, 17] is regarded as the equivalence oracle. If it deems that $\mathcal{A}$ is equivalent to $\mathcal{M}$, the VC algorithm confirms it; otherwise it returns a counterexample that is accepted by $\mathcal{A}$ but rejected by $\mathcal{M}$ or rejected by $\mathcal{A}$ but accepted by $\mathcal{M}$.

Specifically, each algorithm exploits different method to store the gathered information. *Observation Packs* [2] abstracts away the data structure and stores the information in several observation packs sets in a unified way. Angluin's famous algorithm *Observation Table* [1], while still putting information into sets, uses a more concrete data structure: a table. The basic idea behind Angluin's algorithm is to systematically explore the system's behavior using the membership oracle and to build the transition table of a deterministic finite automaton with a minimal number of states. Closely related to Angluin's algorithm, *Reduced Observation Table* [16] contains a smaller version of the table. Obviously different from the previous approaches, *Discrimination Tree* [2, 11] uses a binary tree with labelled nodes.

From the viewpoint of machine learning, all the above are derivation algorithms. One demerit of derivation methods is that they must store the whole computation history in order to continue computing. In other words, all the intermediate results should be stored just as recording all the sub-problem solutions in dynamic programming algorithm. It therefore causes high space complexity and is incapable of dealing with complex system with large number of states, as concluded in [3], "Memory consumption is one major obstacle when we attempted to learn large examples." In practice, the expected bottleneck for derivation learner is the large number of membership queries, especially in discrete occasion, because a communication with a typically slow external device is required.

In addition to fundamental learning algorithms, various works focusing on integrating different techniques to make the learning process more practical have been put forward. Black Box Checking [15] first introduces the idea of implementing the equivalence oracle using VC algorithm. Moreover, it performs verification directly on the system, without a model being presented in advance. However, it depends largely on VC algorithm, and the high complexity of VC algorithm limits its use in dealing with large systems. A successive work, Adaptive Model Checking (AMC) [5, 6] requires an initial model and iteratively improves the accuracy of a possibly faulty model, rather than learning the model from scratch.

In experimental aspect, Hungar et al. [8] aim at optimizing the learning efficiency of Angluin's algorithm by considering domain-specific knowledge. They conduct the experiments on an industrial level test environment and experimental results show the drastic effect of their optimization. In another relevant work, Berg et al. [3] implement Angluin's algorithm in a straightforward way to gain further insights to the algorithm's practical applicability. The experiments focus on the number of states and alphabet size on the needed number of membership queries.

3 Dynamic Model Learning

3.1 Preliminaries

Definition 3.1 (*Deterministic Finite-state Automaton, DFA*) *A deterministic finite-state automaton is a quintuple $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$, where*

- *Q is a finite nonempty set of **states**;*
- *Σ is a finite nonempty set, called **alphabet**;*
- *$\delta : Q \times \Sigma \to Q$ is the **state transition function**;*
- *$q_0 \in Q$ is the **initial state**;*
- *$F \subseteq Q$ is a set of **accepting states**.*

The language accepted by $\mathcal{M}$, denoted by $\mathcal{L}(\mathcal{M})$, is defined as $\mathcal{L}(\mathcal{M}) = \{w \in \Sigma^* \mid \text{there is an accepted run of } \mathcal{M} \text{ on } w\}$. $\mathcal{M}_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \delta^{(\text{see Table 1})}, q_0, \{q_1\})$ (see Fig. 1) is used as an illustration to depict our algorithm.

Definition 3.2 *A language $\mathcal{L}$ is **prefix-closed** if for every word $w \in \mathcal{L}$, all prefixes of w are in $\mathcal{L}$.*

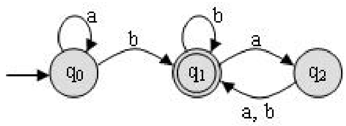


Figure 1. A deterministic finite automaton $\mathcal{M}_1$ that has three states.

Table 1. Transition function δ of DFA $\mathcal{M}_1$.

	a	b
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1

A DFA $\mathcal{M}$ is ***prefix-closed*** if its language $\mathcal{L}(\mathcal{M})$ is prefix-closed. It follows that a prefix-closed DFA has only one non-final state with transitions only to itself. A prefix-closed DFA is useful in modelling reactive systems (see Section 4) and it has a property that prefixes of accepted words are accepted.

Definition 3.3 (*Mealy Machine*) *A Mealy machine is a quintuple $\mathcal{M} = (Q, I, O, \delta, \lambda)$, where*

- *Q is a finite nonempty set of **states**;*
- *I and O are finite nonempty sets of **input symbols** and **output symbols**, respectively;*
- *$\delta : Q \times I \to Q$ is the **state transition function**;*
- *$\lambda : Q \times I \to O$ is the **output function**.*

A Mealy machine is a finite-state machine with inputs and outputs. Its ***characterizing set*** is used to separate every state while its **transition cover set** forces the machine to visit every transition and then stop. They are of significance to perform equivalence test between machines (see Section 4). (For the definition of these concepts, please refer to our technical report [12].)

Given a DFA $\mathcal{M}$, a ***training example*** x is a pair of the form $(w, +)$ or $(w, -)$ for a word w. The ***label*** $+$ signifies the word w is accepted by $\mathcal{M}$, i.e., $w \in \mathcal{L}(\mathcal{M})$, while $-$ means w is rejected by $\mathcal{M}$; A ***training set*** X is a set of all given training examples, i.e., $X = \{x \mid x \text{ is a training example}\}$.

Definition 3.4 *Given a training set X of DFA $\mathcal{M}$, DFA $\mathcal{A}$ is said to be **approximate to** $\mathcal{M}$, denoted by $\mathcal{A} =_{app} \mathcal{M}$, if and only if all words marked $+$ in X are accepted by $\mathcal{A}$, while all words marked $-$ are rejected by $\mathcal{A}$.*

3.2 AMC in a Nutshell

Generally speaking, Adaptive Model Checking (AMC) [5, 6] is concerned with incrementally updating a possibly inaccurate model. A candidate model is assumed to be provided, and this model is used for checking given properties (usually specified in LTL). If a counterexample is found, it will be compared with the real system. If this is found to be an actual execution of the system, true counterexample is reported. Otherwise, the learning algorithm is executed, taking account of the newly detected counterexample. If no counterexample is found, a conformance tester is used to check that whether the system and the model have the same behaviros. Upon finding a discrepancy, the learning algorithm is invoked and starts a new iteration. The entire iterative process terminates when a true counterexample is found, or when the conformance tester finds no difference between the model and the real system.

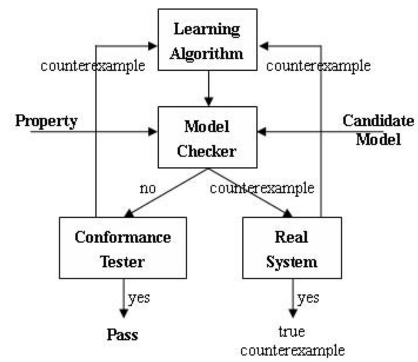


Figure 2. Overview of AMC.

This paper mainly focuses on devising such a learning algorithm as explained in the next section, and briefly discusses the whole learning process as well as its features in Section 3.5.

3.3 GA-based Model Learning Algorithm

We assume that the black box system to be learnt can be modelled as a DFA. The ideal model which actually depicts the real system is denoted by $\mathcal{A}_B$, while the learned model, also called conjecture or approximation of $\mathcal{A}_B$, is denoted by $\mathcal{A}_{app}$. Moreover, the actions (in Σ) that the system is able to perform and the number of states are assumed to be known. A collection of accepted and rejected words are also gathered and form the training set X. Given this training set, the learning task is to build a DFA $\mathcal{A}_{app}$ approximate to $\mathcal{A}_B$.

3.3.1 Algorithm Overview

Holland [7] proposes the original genetic algorithm (GA) by simulating the biological evolution procedure, and it is so called standard genetic algorithm. The core of our learning system (Deterministic Finite-state Automaton Learning System Based on Genetic Algorithm, GA-DFALS) is based on this framework (see Table 2). The genetic operators we used are fitness proportionate selection operator, uniform crossover operator, and point mutation operator. The details of the algorithm will be discussed in the following subsections.

Table 2. Algorithm Framework for GA-DFALS

Algorithm: Learning Deterministic Finite-state Automaton Based on Genetic Algorithm (A Basis)
Input:

$Fitness$	fitness function
n	the number of the population
p_c	crossover rate
p_m	mutation rate
X	training set

Output: hypothesis with the highest fitness value

- Represent hypothesis via binary encoding;
- Generate n hypotheses randomly and form a population P;
- For each hypothesis h, compute $Fitness(h)$;
- $max \leftarrow$ maximum fitness value of the current population P;
- **while** ($max < 1.0$) // If not all the examples are correctly recognized, generating a new population P_{new}.
  - Selection: Choose $(1-p_c) \cdot n$ members from P and add them into P_{new} using selection operator;
  - Crossover: Choose $p_c \cdot n/2$ pairs of hypotheses from P; For each pair of hypotheses, produce 2 descendants using crossover operator; add them into P_{new};
  - Mutation: Choose $p_m\%$ members from P_{new} with uniform probability; For each, randomly flip one bit or several bits;
  - Replace population P with P_{new};
  - For each hypothesis h, compute $Fitness(h)$;
  - $max \leftarrow$ maximum fitness value of the current population P;
- **end while**

Contrast to derivation methods, genetic algorithm is essentially a randomized algorithm, but its correctness is guaranteed by satisfying all the constraints specified in the training set. So, no intermediate results need recording. Only the encoding of the problem domain and the training set need memorizing. Therefore, it largely reduces the space complexity with respect to derivation algorithms.

3.3.2 Representing Hypothesis

Representing Finite Sets Let S be a finite set. The purpose of representing S is to encode each element of S as boolean values. We adopt the encoding scheme similar to symbolic model checking [13]. Each element $e \in S$ is assigned a unique vector of boolean values $(x_1, x_2, \ldots, x_n)$, such that $x_i \in \{0,1\}$. n should be chosen so that $2^{n-1} < |S| \leq 2^n$, where $|S|$ is the number of elements in S. If $|S|$ is not an exact power of 2, there will be some vectors which do not correspond to any element of S; they are ignored. To achieve this, the ***characteristic function*** $f_S : \{0,1\}^n \rightarrow \{0,1\}$ over S is given. f_S maps e, represented by $(x_1, x_2, \ldots, x_n)$, onto 1 if $e \in S$ and maps it onto 0 otherwise.

Representing States and Alphabet Since states and alphabet are finite sets, they can be encoded using the method described in Section 3.3.2. For example, Q of DFA $\mathcal{M}_1$ can be represented as

Table 3. Binary representation of states.

state in Q	binary representation
q_0	$(0,0)$
q_1	$(0,1)$
q_2	$(1,0)$

Representing Transition Function A transition $q \xrightarrow{a} q'$ can be represented by a 3-tuple boolean vectors $(\overrightarrow{x}, \overrightarrow{v}, \overrightarrow{x'})$, where $\overrightarrow{x}$, $\overrightarrow{x'}$ and $\overrightarrow{v}$ represents q, q' and 'a' respectively and $f_Q(\overrightarrow{x}) \wedge f_\Sigma(\overrightarrow{v}) \wedge f_Q(\overrightarrow{x'}) = 1$. While it is easy to interpret and readily to transform to boolean function [9], it has space complexity of $O(|\Sigma| \cdot |Q|^2)$. For the sake of saving memory, we devise scheme with space complexity $O(|\Sigma| \cdot |Q|)$. Note that the contents of transition function table are merely states (see Table 1). We can serialize them by concatenating boolean vectors of each cell state by state, from left to right, and row by row, from top to bottom. For example, the transition function δ of $\mathcal{M}$ can be represented as "101010011000". However, from implementation view, this scheme requires a auxiliary data structure to map a pair $< state, letter >$ onto $state$, which is not very time effiency. In reality, we realize this by the map template of C++ STL.

Representing an Initial State Since there exists one and only one initial state of DFA, we can simply allocate n bits to represent it and require that $f_Q((x_1, x_2, \ldots, x_n)) = 1$. For example, if the hypothesis conjectures "01" in this fragment, we can interpret that q_1 is the initial state, though it is not necessarily a correct guess.

Representing Final States Since there may be more than one state in the final state set, we can allocate $|S|$ bits to represent them, say $b_1 b_2 \ldots b_{|S|}$. At the same time, we require $b_1 \vee b_2 \vee \ldots \vee b_{|S|}$ is valid, which means several of them can be selected as final state, but none is not allowed.

3.3.3 Genetic Operators

The generation of successors in genetic algorithm is determined by a set of operators that recombine and mutate selected members of the current population [14]. These operators correspond to idealized version of the genetic operations found in biological evolution. The selection operator chooses individuals that are well adaptive to the environment via probability method; The crossover operator produces two new offsprings from parts of each parent; The mutation operator creates a single descendant from a single parent by changing the value of randomly chosen bits, which enable the population to reproduce new breeds with time.

Selection Operator The selection operator is used to preserve individuals that are competitive. Since the adaptability of population is evaluated by fitness function, one intuitive idea is that the higher the fitness value an individual obtains, the greater possibility it will be preserved. Assuming that there is altogether n hypotheses, the probability of selecting hypothesis h_i from population is given by

$$Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{n} Fitness(h_j)}$$

In other words, the probability that a hypothesis will be selected is given by the ratio of its fitness to the fitness sum of all members of the current population. This method is sometimes called *fitness proportionate selection*.

Crossover Operator The crossover operator produces two new offsprings from two parent strings, by copying selected bits from each parent. The bit at position i in each offspring is copied from the bit at position i in one of the two parents. The choice of which parent contributes the bit for position i is determined by an additional string called the *crossover mask*. There are many methods to implement crossover operator, such as single-point crossover, two-point crossover and uniform crossover. Since our encoding has certain syntax format, we adopt uniform crossover.

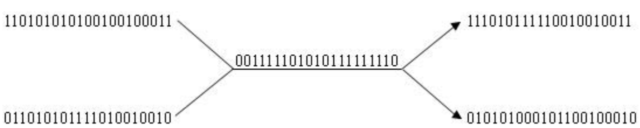


Figure 3. Uniform crossover operation.

Uniform crossover combines bits sampled uniformly from the two parents, as illustrated in Fig. 3. In this case the crossover mask is generated as a random bit string with each bit chosen at random and independent of the others. A "1" appears at position i in the mask indicates exchanging bit of parents at corresponding position, while a "0" prescribes doing nothing.

Mutation Operator The mutation operator produces small random changes to the bit string by choosing a single bit (some bits) at random, then changes its value (their values). Mutation operator is often performed after crossover operator has been applied.

Figure 4. Point mutation operation.

3.3.4 Fitness Function

We have two types of training examples, namely words labelled by '+' and '−'. Therefore, the learned DFA can be regarded as a classifier. The fitness of each hypothesis is thus based on its classification accuracy over the training set X. In particular, we design the function used to measure fitness as follow

$$Fitness(h) = \begin{cases} 0.0 & \exists q.f_Q(q) = 0;\ (q \text{ is interpreted from contents of transition function table or initial state.}) \\ Correct(h) & \text{otherwise.} \end{cases}$$

where $Correct(h)$ is the percent of all training examples correctly classified by hypothesis h, i.e. words labelled '+' are accepted by $\mathcal{A}_{app}$ and those labelled '−' are rejected by $\mathcal{A}_{app}$.

3.4 Algorithm Optimization

Since we use the standard genetic algorithm, the convergence is guaranteed by many theoretical results, and the algorithm itself is independent of specific domain. However, the problem of local minima still exists, and it is a well-known conundrum named premature convergence. Although the algorithm generally exploits global search, and can jump out of the local minima with mutation operator. Nevertheless, the strategy applied to prohibit premature problem really affects the speed of convergence. We design the following strategies to address this problem.

3.4.1 Adaptive Mutation Rate

In standard genetic algorithm, the number of population, crossover rate, and mutation rate are set to fixed values. Recently, many researchers realize that these parameters need to variate with the genetic evolution. An algorithm with self-organizing characteristic is apt to more robust, global optimized, and effective.

Whitney and Starkweather [18] propose a mutation strategy, where mutation rate p_m is inverse to the hamming distance of one randomly chose parent. Hamming distance $\delta : \{0,1\}^n \times \{0,1\}^n \rightarrow Int$ is a function that maps two boolean vectors into an integer representing the amount of the different bits in corresponding positions. For example, the hamming distance of "10001001" and "10110001" is 3. We make some adaptation of this strategy to merge it into our algorithm. In each generation, we store the best and worst hypothesis, h_{best} and h_{worst} respectively, and compute the mutation rate as follow.

$$p_m = p_m^{min} + (p_m^{max} - p_m^{min}) \times \frac{n - \delta(h_{best}, h_{worst})}{n}$$

Where p_m^{min} and p_m^{max} is the minimum and maximum fixed mutation rate suggested by classic genetic algorithm, n is the length of hypothesis, i.e. $n = |h_{best}| = |h_{worst}|$, and δ is the hamming distance mentioned above.

3.4.2 Inertia Punishment

We record the highest fitness value of each generation. If a continuous sequence of them with the same value, where the length of the sequence exceeds an allowed threshold (MAX_STABILITY), is detected, we temporarily enlarge the mutation rate to a fairly large value (≈ 1.0). This situation indicates appearance of local minima, and the whole population is inertia to evolve. The accidental change of mutation rate simulates environmental disaster, and punishes the population's inertia.

The application of "Adaptive Mutation Rate" and "Inertia Punishment" aim at keeping the diversity of the population. These strategies will help the algorithm jump out of local minima, and thus effectively prohibit the premature convergence as shown in Section 4.

3.4.3 Preserve Best Hypotheses

On one hand, the multiplicity of population allows the population to escape the local minima. On the other hand, there must be a leading force to urge the population to progress towards the optimum solution. We slightly modify the standard selection operator by preserving the top N best hypotheses into next generation, to achieve this goal. In reality, N is very small compared with the number of population. This idea is adapted from "Stochastic Tournament Selection".

3.5 Dynamic Learning Process

We put our work under the Adaptive Model Checking (AMC) [5, 6] framework (see the gray frame in Fig. 5), and dynamically learn the model while performing model checking. Vasilevskii-Chow (VC) algorithm [4, 17] plays the role of a conformance tester, checking equivalence between the model and the system.

First of all, we randomly generate an initial training set X of accepted and rejected words as input to GA-DFALS and obtain a rough model of the system. The initial learned model is not necessarily correct. But the point is, that the inaccurate model is still useful for verification and we thus enable the model checking process to carry out immediately. When model checking requests are submitted, the model checker uses the currently learned model to verify properties. At this point, there are two possible outcomes of this check.

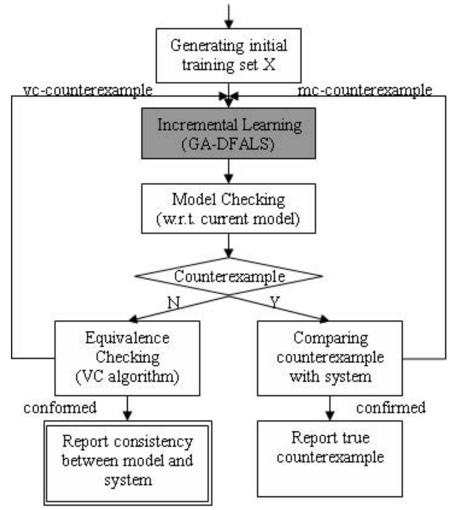


Figure 5. Dynamic model learning process.

Mc-counterexample found If the model checker finds a counterexample, called mc-counterexample, either the system indeed does not have the checked property or it is a false counterexample due to the fact that the model is incorrect. So, the system executes the counterexample to find out this.

If the system accepts this counterexample, it means that the system does not satisfy the property. We can merely report this counterexample and temporarily terminate the process. Otherwise, if the system rejects it, the mc-counterexample will be inserted into X and GA-DFALS will be invoked to refine the model.

No mc-counterexample In another branch, the model checker reports no mc-counterexample. In this case, VC algorithm will be used to make sure whether the model and the system is conformed, i.e. checking the equivalence between the learned model and the real system.

If the VC algorithm reports vc-counterexample, again, it will be included in training set X and the model is refined. Otherwise, if no vc-counterexample is found, the conformance between the model and the system has been established. Therefore, the learning process can eventually terminate, and reports the correct model which is consistent with the real system.

The incremental character of GA-DFALS is twofold. In the first place, new counterexamples treated as constraints of the model are combined with the original generated training set to refine the model. On the other hand, the previous learned model can be used as an initial hypothesis to shorten the learning process. However, our approach does not assume an initial model must be provided by the users. Pure model checking represents white box checking, where we have complete knowledge of the internal structure of the object system. Black box checking represents the other extreme, which enables us to check properties while knowing nothing about the system. Adaptive model checking is an intermediate case when we have an approximate but possibly inaccurate model $\mathcal{A}_{app}$. It starts with $\mathcal{A}_{app}$, and then iteratively refine it or discover an error. GA-DFALS learning process combines the benefits of both black box checking and adaptive model checking, which can either build a model from scratch or refine the it when the system evolves.

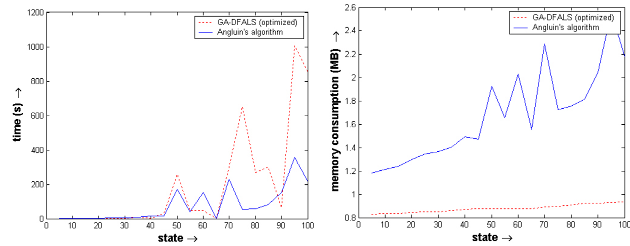
4 Experimental Result

We have implemented our learning algorithm closely following the high-level description in Section 3.3.1. At the same time, we implemented Augluin's algorithm described in [1] of our own to evaluate our proposed method. We randomly generated DFAs to simulate the membership oracles at the same computer as the learning algorithm. In practice, they will be realized as processes communicating with the real system. We use the W method [4] (a special case of VC algorithm), to generate training examples for our learning algorithm and to play the role of equivalence oracle for Augluin's algorithm. (Interesting readers may refer to our technical report [12] for a short description of W method.) If the learned automaton $\mathcal{A}_{app}$, which has been translated into a Mealy machine, produces the same outputs as $\mathcal{A}_B$ on all testing words generated by W method, then $\mathcal{A}_{app}$ is equivalent to $\mathcal{A}_B$. Therefore, when our learning algorithm converges, it produces equivalent automata as $\mathcal{A}_B$, so that the quality of the models learned by our algorithm and Angluin's algorithm are the same.

We refer to the experiment report [10] to configure our GA parameters: the number of population $n = 100$, crossover rate $p_c = 0.6$, minimum mutation rate $p_m^{min} = 0.01$, and maximum mutation rate $p_m^{max} = 0.05$. The maximum allowed threshold of the same highest fitness value sequence MAX_STABILITY is 9; the inertia punishment mutation rate is 1.0; the number of preserved best hypotheses is 2. Our developing environment is: CPU(Celeron 1.1G) + RAM(256M) + OS(Windows XP SP2) + Compiler(Microsoft Visual C++ .NET 2003).

4.1 Learning Randomly Generated DFAs

We randomly generated DFAs with states ranging from 5 to 100 and learned them, with fixed number of alphabet (=2). In Fig. 6(a), the dotted line indicates the learning time with optimization while the solid line indicates the learning time of Angluin's algorithm. We compare the learning time of Angluin's algorithm and GA-DFALS, using the optimizing version of our learning algorithm. Fig. 6(a) indicates

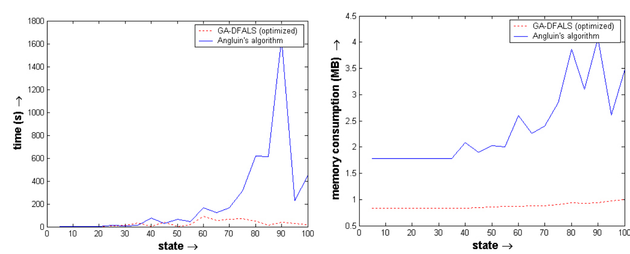


(a) Learning Time (b) Memory Consumption

Figure 6. Learning time and memory consumption of randomly generated DFAs.

that the learning time are quite the same when encountering small states, but GA-DFALS are a bit slower with larger states. We adopt the scheme of encoding transition function in Section 3.3.2 and therefore the space complexity of our algorithm is $O(|\Sigma| \cdot |Q|) + O(1) + O(|Q|) = O(|\Sigma| \cdot |Q|)$. Besides the basic system cost, it is linear with fixed $|\Sigma|$. We compare the memory consumption of GA-DFALS and Angluin's algorithm in Fig. 6(b) and it shows that GA-DFALS is more compact than Angluin's algorithm in space complexity.

4.2 Learning Randomly Generated Prefix-closed DFAs



(a) Learning Time (b) Memory Consumption

Figure 7. Learning time and memory consumption of learning randomly generated prefix-closed DFAs.

Though a suitable abstraction of reactive systems yields indeed a finite automaton, it is a very special one: Its set of finite traces forms a prefix-closed language [8], i.e. reactive systems can be modelled as prefix-closed DFAs. However, prefix-closed languages are relatively hard to learn compared to arbitrary regular languages [3]. Recall that a prefix-closed DFA has only one non-final state with transitions only to itself (Section 3.1). We use this property to construct prefix-closed DFAs, and making comparison with

Proceedings of the Sixth International Conference on Quality Software (QSIC'06)
0-7695-2718-3/06 $20.00 © 2006 IEEE

Angluin's algorithm.

We randomly generated prefix-closed DFAs with states ranging from 5 to 100 and learned them, with fixed number of alphabet (=2). In Fig. 7(a), GA-DFALS learns much faster, no matter what it encounters, either small states or large states. But, we cannot safely conclude that the learning time of GA-DFALS is promising than Angluin's algorithm, due to its randomness. However, GA-DFALS surely uses less memory than Angluin's algorithm, as can be seen in Fig. 6(b) and 7(b). (Two more comparison examples, which can be regarded as of large scale, are given in our technical report [12].) In fact, Berg et al. [3] reporte that "Memory consumption is one major obstacle when we attempted to learn large examples." The compact encoding scheme of GA-DFALS really enables us to deal with large complex systems in the aspect of memory consumption.

5 Conclusion and Future Work

In this paper, we developed a dynamic approach for automatically generating formal models from real systems, whose internal structure is totally unknown. Our learning process is under the adaptive model checking framework, and the core learning algorithm is based on genetic algorithm. Our method combines techniques from model checking, conformance testing and model learning theory. Unlike previous model learning methods, our approach can either build a model from scratch or iteratively refine it each time model checking requests are submitted, rather than learning a perfect model in one trial. Experimental results show that it has the potential to handle large systems as to the memory consumption aspect. We aim at bridging the gap between black box systems and available model-based techniques.

However, adaptive model learning is still a mainly unexplored area that future theory as well as practical insights are needed. This paper mainly focuses on the core learning algorithm. But in practice how to integrate the model checker and conformance tester need further investigate. One of our motivations is to enable rapid usages of formal models, but due to the inherent premature convergence of genetic algorithm, the efficiency of our current system is not very satisfiable. More optimization strategies should be devised in addition to our three ones. However, the inborn parallelism trait of genetic algorithm is a promising way to address efficiency problem.

How to generate the initial training set is another problem we encountered. If an initial model is given explicitly, this problem will not become so significant. However, if we want to learn the model from scratch, we must create criterion for generating initial training set.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[2] J. L. Balcázar, J. Díaz, and R. Gavaldà. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997.

[3] T. Berg, B. Jonsson, M. Leucker, and M. Saksena. Insights to angluin's learning. *Electr. Notes Theor. Comput. Sci.*, 118:3–18, 2005.

[4] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.

[5] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In J.-P. Katoen and P. Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370. Springer, 2002.

[6] A. Groce, D. Peled, and M. Yannakakis. Amc: An adaptive model checker. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 521–525. Springer, 2002.

[7] J. H. Holland. Concerning efficient adaptive systems. In *Yovits M C, eds. Self-Organizing Systems.*, pages 215–230, 1962.

[8] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In W. A. H. Jr. and F. Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.

[9] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning aobut Systems*, chapter 6, pages 382–390. Cambridge University Press, second edition, 2004.

[10] K. A. D. Jong, W. M. Spears, and D. F. Gordon. Using genetic algorithms for concept learning. *Machine Learning*, 13:161–188, 1993.

[11] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Cambridge, Massachusetts and London, England, 1994.

[12] Z. Lai, S.-C. CHEUNG, and Y. Jiang. Dynamic model learning using genetic algorithm under adaptive model checking framework. Technical Report HKUST-CS06-05, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, July 2006.

[13] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, Boston, 1992.

[14] T. Mitchell. *Machine Learning*, chapter 9, pages 249–273. McGraw Hill, first edition, Mar. 1997.

[15] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[16] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In S. J. Hanson, W. Remmele, and R. L. Rivest, editors, *Machine Learning: From Theory to Applications*, volume 661 of *Lecture Notes in Computer Science*, pages 51–73. Springer, 1993.

[17] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetic*, 9(4):653–665, 1973.

[18] D. Whitley and T. Starkweather. Genitor ii.: a distributed genetic algorithm. *J. Exp. Theor. Artif. Intell.*, 2(3):189–214, 1990.