PAC Learning-Based Verification and Model Synthesis

Yu-Fang Chen
Academia Sinica

Chiao Hsieh
Academia Sinica
National Taiwan University

Ondřej Lengál
Academia Sinica
Brno University of Technology

Tsung-Ju Lii
Academia Sinica
National Taiwan University

Ming-Hsien Tsai
Academia Sinica

Bow-Yaw Wang
Academia Sinica

Farn Wang
National Taiwan University

ABSTRACT

We introduce a novel technique for verification and model synthesis of sequential programs. Our technique is based on learning a regular model of the set of feasible paths in a program, and testing whether this model contains an incorrect behavior. Exact learning algorithms require checking equivalence between the model and the program, which is a difficult problem, in general undecidable. Our learning procedure is therefore based on the framework of *probably approximately correct* (PAC) learning, which uses sampling instead and provides correctness guarantees expressed using the terms *error probability* and *confidence*. Besides the verification result, our procedure also outputs the model with the said correctness guarantees. Obtained preliminary experiments show encouraging results, in some cases even outperforming mature software verifiers.

1. INTRODUCTION

Formal verification of software aims to prove software properties through rigorous mathematical reasoning. Consider, for example, the C statement `assert(x > 0)` specifying that the value of the variable `x` must be positive. If the assertion is formally verified, it *cannot* be violated in any possible execution during runtime. Formal verification techniques are, however, often computationally expensive. Although sophisticated heuristics have been developed to improve scalability of the techniques, formally verifying real-world software is still considered to be impractical.

A common technique to ensure quality in industry is software testing. Errors in software can be detected by exploring different software behaviors via injecting various testing vectors. Testing cannot, however, guarantee software is free from errors. Consider again the assertion `assert(x > 0)`. Unless all system behaviors are explored by testing vectors, it is unsound to conclude that the value of `x` is always positive. Various techniques have been proposed to improve its coverage, but it is an inherent feature of software testing that it cannot establish program properties conclusively.

In this paper, we propose a novel learning-based approach that aims to balance scalability and coverage of existing software engineering techniques. In order to be scalable, as for software testing, our new technique explores only a subset of all program behaviors. Moreover, we apply machine learning to generalize observed program behaviors for better semantic coverage. Our technique allows software engineers to combine scalable testing with high-coverage formal analyses and improve the quality assurance process. We hope that this work reduces the dichotomy between formal and practical software engineering techniques.

In our technical setting, we assume programs are annotated with program assertions. A program assertion is a Boolean expression intended to be true every time it is encountered during program execution. Given a program with assertions, our task is to check whether all assertions evaluate to true on all possible executions. In principle, the problem can be solved by examining all program executions. It is, however, prohibitive to inspect all executions exhaustively since there may be infinitely many of them. One way to simplify the analysis is to group the set of program executions to paths of a control flow graph.

A *control flow graph* (CFG) is derived from the syntactic structure of a program source code. Each execution of a program corresponds to a path in its control flow graph. One can therefore measure the completeness of software testing on CFGs. Line coverage, for instance, gives the ratio of explored edges in the CFG of the tested program, while branch coverage is the ratio of explored branches of this CFG. Note that such syntactic measures of code coverage approximate program executions only very roughly. Executions that differ in the number of iterations in a simple program loop have the same line and branch coverages, although their computation may be drastically different. A full syntactic code coverage does not necessarily mean all executions have been explored by software testing.

Observe that program executions traversing the same path in a CFG perform the same sequence of operations (although maybe with different values). Consider a path corresponding to a program execution in a CFG. Such a path can be characterized by the sequence of decisions that the execution took when traversing conditional statements in the CFG. We call a sequence of such decisions a *decision vector*. A decision vector is feasible if it represents one or more (possibly infinitely many) program executions, and infeasible if it represents a sequence of branching choices than can never occur in an execution of the program. To check whether all assertions evaluate to *true* on all executions, it suffices to examine all feasible decision vectors and check they do not represent any assertion-violating program execution. Although feasibility of a decision vector can be determined by using an off-the-shelf Satisfiability Modulo Theories (SMT) solver, the set of feasible decision vectors is in general difficult to compute exactly. Therefore, we apply algorithmic learning, in particular the framework of probably approximately correct learning, to construct a regular approximation of this set.

Within the framework of *probably approximately correct* (PAC) learning with queries, learning algorithms query about target concepts to construct hypotheses. The constructed hypotheses are then validated by sampling. If a hypothesis is invalidated by witnessing a counterexample, learning algorithms refine the invalidated hypothesis by the witness and more queries. If, on the other hand, a hypothesis conforms to all samples, PAC learning algorithms return the inferred hypothesis with statistical guarantees. In our approach, we adopt a PAC learning algorithm with queries to infer a regular language approximation to the set of feasible decision vectors of a program.

To grasp the statistical guarantees provided by PAC learning, consider the task of checking defects in a large shipment using uniform sampling. Because of the size of the shipment, it is impractical to check every item. We instead want to know, with a given confidence δ, if the defect probability is at most ϵ. This can be done by selecting r (to be determined later) randomly chosen items. If all chosen items are good, the method reports that the defect probability is at most ϵ. We argue the simple method can err with probability at most $1-\delta$. Suppose r randomly chosen items are tested without any defect, but the defect probability is, in fact, *more than* ϵ. Under this thesis, the probability that r random items are all good is lower than $(1-\epsilon)^r$. That is, the method is incorrect with probability lower than $(1-\epsilon)^r$. Take r such that $(1-\epsilon)^r < 1-\delta$. The simple method reports incorrect results with probability at most $1-\delta$; we equivalently say the result of the method is $PAC(\epsilon,\delta)$-correct.

Using a similar argument, it can be shown that our PAC learning algorithm returns a regular set approximating the set of feasible decision vectors of the program with the error probability ϵ and confidence δ of our choice. If the inferred set contains no decision vector representing an assertion-violating program execution, our technique concludes the verification with statistical guarantees about correctness.

Our learning-based approach finds a balance between formal analysis and testing. Rather than exploring program behaviors exhaustively, our technique infers an approximation of the set of feasible decision vectors by queries and sampling. Although the set of feasible decision vectors is in general not computable, PAC learning with queries may still return a regular set approximation of it with a quantified guarantee. Such an approximate model with statistical guarantees can be useful for program verification. With an approximate model that is $PAC(\epsilon,\delta)$-correct and proved to be free from assertion violation, one can conclude that the program is also $PAC(\epsilon,\delta)$-correct. The statistical guarantees are different from syntactic code coverages in software testing. Recall that our application of PAC learning works over decision vectors. Decision vectors in turn represent program executions. When our technique does not find any assertion violation, the statistical guarantees give software engineers a semantic coverage about program executions. Along with conventional syntactic coverages, such information may help software engineers estimate the quality of software.

We implement a prototype, named PAC-MAN (PAC learning-based Model synthesizer and ANalyzer), of our procedure based on program verifiers CPACHECKER, CBMC, and the concolic tester CREST. We evaluate the prototype on the benchmarks from the recursive category of SV-COMP 2015 [1]. The results are encouraging—we can find all errors that can be found by CREST. We also provide quantified guarantee accompanied by a faithful approximate model for several examples that are challenging for program verifiers and concolic testers. This approximate model can later be reused, e.g., for verifying the same program with a different set of program assertions.

Our contributions are summarized in the following:

- We show the PAC learning algorithm can be applied to synthesize a faithful approximate model of the set of feasible decision vectors of a program. Such a model can be useful in many different aspect of program verification (cf. Section 12 for details). We believe it is not hard to adopt our approach to handle different type of systems (e.g., black box systems) and to obtain approximate models on a different level of abstraction (e.g., on a function call graph).
- We develop a verification procedure based on the approximate model obtained from PAC learning. The procedure integrates the advantages of both testing and verification. It uses testing techniques to collect samples and catch bugs. The PAC learning algorithm generalizes the samples to obtain an approximate model that can then be analyzed by verification techniques for statistical guarantees.

2. PRELIMINARIES

Let $\mathcal{X}$ be the set of program variables and $\mathcal{F}$ the set of function and predicate symbols. We use $\mathcal{X}'$ for the set $\{x' \mid x \in \mathcal{X}\}$. The set $\mathcal{T}[\mathcal{X},\mathcal{F}]$ of *transition* formulae consists of well-formed first-order logic formulae over $\mathcal{X}, \mathcal{X}'$, and $\mathcal{F}$. For a transition formula $f \in \mathcal{T}[\mathcal{X},\mathcal{F}]$ and $n \in \mathbb{N}$, we use $f^{\langle n\rangle}$ to denote the formula obtained from f by replacing all free variables $x \in \mathcal{X}$ and $x' \in \mathcal{X}'$ with $x^{\langle n\rangle}$ and $x^{\langle n+1\rangle}$ respectively.

We represent a program with a single procedure using a control flow graph. (Section 9 extends the notion to programs with multiple procedures and procedure calls.) A *control flow graph (CFG)* is a graph $G = (V, E, v_i, v_r, V_e, \mathcal{X}_{FP})$ where $V = V_b \cup V_s$ is a finite set of *nodes* consisting of disjoint sets of *branching* nodes V_b and *sequential* nodes V_s, $v_i \in V$ is the *initial* node, $v_r \in V_s$ is the *return* node, $V_e \subseteq V$ is the set of *error* nodes, $\mathcal{X}_{FP} \subseteq \mathcal{X}$ is the set of formal parameters, and E is a finite set of *edges* such that $E \subseteq V \times \mathcal{T}[\mathcal{X},\mathcal{F}] \times V$ and the following conditions hold:

- for any branching node $v_b \in V_b$, there are exactly two nodes $v_0', v_1' \in V$ with $(v_b, f_0, v_0'), (v_b, f_1, v_1') \in E$, where $f_0, f_1 \in \mathcal{T}[\mathcal{X},\mathcal{F}]$ are transition formulae;
- for any non-return sequential node $v_s \in V_s \setminus \{v_r\}$, there is exactly one node $v' \in V$ with $(v_s, f, v') \in E$; and
- for the return node $v_r \in V_s$, there is no $v' \in V$ such that $(v_r, f, v') \in E$ for any $f \in \mathcal{T}[\mathcal{X},\mathcal{F}]$.

We say v' is a *successor* of v if $(v, f, v') \in E$. Assume, moreover, that the two successors v_0' and v_1' of the branching node v are ordered. Intuitively, the '1' corresponds to the *if* branch and the '0' corresponds to the *else* branch. We call v_0' and v_1' the 0-*successor* and 1-*successor* of v respectively. Similarly, f_0 and f_1 are called the 0-*transition* and 1-*transition formulae* of v. Note that the definition of a CFG allows us to describe nondeterministic choice, which is commonly used to model the environment. To be more specific, a nondeterministic choice from a branching node v can be represented by defining both the 0-transition and 1-transition formulae of v as $\bigwedge_{x\in\mathcal{X}} x = x'$.

A *path* in the CFG G is a sequence $\pi = \langle v_0, f_1, v_1, f_2, v_2, \ldots, f_m, v_m\rangle$ such that $v_0 = v_i$ and $(v_j, f_{j+1}, v_{j+1}) \in E$ for every $0 \le j < m$. The path π is *feasible* if the path formula $\bigwedge_{k=1}^{m} f_k^{\langle k\rangle}$ is satisfiable. It is an *error* path if $v_j \in V_e$ for some $0 \le j \le m$. The task of our analysis is to check whether G contains a feasible error path.

A sequence $w = a_1 a_2 \cdots a_n$ with $a_j \in \{0,1\}$ for $1 \le j \le n$ is called a *word* over $\{0,1\}$. The *length* of w is $|w| = n$. The word of length 0 is the *empty* word λ. We also use $w[j]$ to denote the j-th symbol a_j. If u, w are words over $\{0,1\}$, $u \cdot w$ denotes the *concatenation* of u and w. A *language* L over $\{0,1\}$ is a set of words over $\{0,1\}$.

We introduce the function $\mathsf{decision}$ that maps a path π of G to a sequence of decisions made in the branching nodes traversed by π. Formally, $\mathsf{decision}$ is a function from paths to words over $\{0,1\}$ defined recursively as follows: $\mathsf{decision}(\langle v_0, f_1, v_1, f_2, v_2, \ldots, f_m, v_m\rangle) = \mathsf{decision}(\langle v_0, f_1\rangle) \cdot \mathsf{decision}(\langle v_1, f_2, v_2, \ldots, f_m, v_m\rangle)$ such that

$$\mathsf{decision}(\langle v\rangle) = \lambda$$

$$\mathsf{decision}(\langle v, f\rangle) = \begin{cases} \lambda & \text{if } v \in V_s, \\ 0 & \text{if } v \in V_b \text{ and} \\ & f \text{ is the 0-transition formula of } v, \\ 1 & \text{if } v \in V_b \text{ and} \\ & f \text{ is the 1-transition formula of } v. \end{cases}$$

For a path π, $\mathsf{decision}(\pi)$ is the *decision vector* of π. We lift $\mathsf{decision}$ to a set of paths Π and define *decision vectors* of Π as $\mathsf{decision}(\Pi) = \{\mathsf{decision}(\pi) \mid \pi \in \Pi\}$.

A *finite automaton* (with λ-moves) A is a tuple $A = (\Sigma, Q, q_i, \Delta, F)$ consisting of a finite *alphabet* Σ, a finite set of *states* Q, an *initial* state $q_i \in Q$, a *transition relation* $\Delta \subseteq Q \times (\Sigma \cup \{\lambda\}) \times Q$, and a set of *accepting* states $F \subseteq Q$. A transition $(q, \lambda, q') \in \Delta$ is called a *λ-transition*. A word w over Σ is *accepted* by A if there are states $q_0, \ldots, q_m \in Q$ and symbols (or λ's) $a_1, \ldots, a_m \in (\Sigma \cup \{\lambda\})$, such that $w = a_1 \cdots a_m$, for every $0 \le j < m$ there is a transition $(q_j, a_{j+1}, q_{j+1}) \in \Delta$, and further $q_0 = q_i$ and $q_m \in F$. The *language* of A is defined as $L(A) = \{w \mid w \text{ is accepted by } A\}$. A language R is *regular* if $R = L(A)$ for some finite automaton A. The finite automaton A is *deterministic* if its transition relation is a function from $Q \times \Sigma$ to Q. For any finite automaton A, there exists a deterministic finite automaton (DFA) B such that $L(A) = L(B)$.

A *pushdown automaton* (PDA) is a tuple $P = (\Sigma, Q, \Gamma, q_i, \Delta, F)$ where Σ is a finite *input alphabet*, Q is a finite set of *states*, Γ is a finite *stack alphabet*, $q_i \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and $\Delta \subseteq Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\}) \times Q$ is a *transition relation*. We use $(q, [a; b/c], q')$ to denote the transition (q, a, b, c, q'), and we sometimes simplify $(q, [a; \lambda/\lambda], q')$ to (q, a, q'). We define a *configuration* of P as a pair $(q, \gamma) \in Q \times \Gamma^*$. A word w over Σ is *accepted* by P if there exists a sequence of configurations $(q_0, \gamma_0), \ldots, (q_m, \gamma_m) \in Q \times \Gamma^*$ and a sequence of symbols (or λ's) $a_1, \ldots, a_m \in (\Sigma \cup \{\lambda\})$, such that $w = a_1 \cdots a_m$, $q_0 = q_i$, $\gamma_0 = \epsilon$, $q_m \in F$, and for every $0 \le j < m$ it holds that there are some $b_j, b_{j+1} \in (\Gamma \cup \{\lambda\})$ and $\gamma'_j, \gamma'_{j+1} \in \Gamma^*$ such that $\gamma_j = b_j \gamma'_j$, $\gamma_{j+1} = b_{j+1}\gamma'_{j+1}$, and there is $(q_j, [a_{j+1}; b_j/b_{j+1}], q_{j+1}) \in \Delta$. The language of P is defined as $L(P) = \{w \mid w \text{ is accepted by } P\}$.

3. OVERVIEW

In this section, we give an overview of our verification procedure. Let G be a CFG of a program. Our goal is to check whether there is a feasible error path in G. More concretely, consider the set Π of feasible paths in G and the set $\mathcal{B}$ of error paths in G. We call the languages $\mathsf{decision}(\Pi)$ and $\mathsf{decision}(\mathcal{B})$ over the alphabet $\{0,1\}$ as *feasible decision vectors* and *error decision vectors* respectively. The program is correct if the intersection $\mathsf{decision}(\Pi) \cap \mathsf{decision}(\mathcal{B})$ is empty, i.e., if G contains no feasible error path.

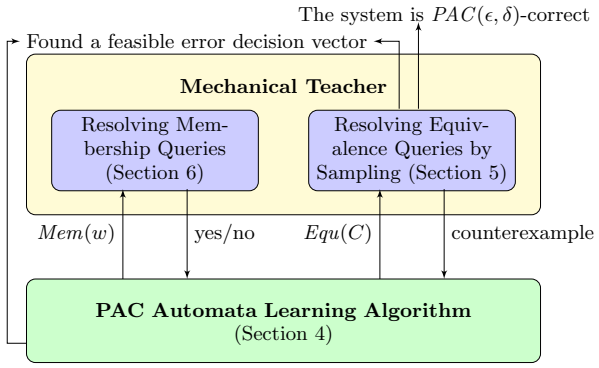

Figure 1: Components of our verification procedure

Representation of the language $\mathsf{decision}(\Pi)$ of all feasible decision vectors in G is not so easy. In general, this language may not be regular or even computable. In our procedure, we construct a candidate finite automaton C that approximates $\mathsf{decision}(\Pi)$, the set of feasible decision vectors of G. We infer C using a *probably approximately correct* (PAC) online automata learning algorithm [2]. The use of PAC learning provides us with statistical guarantees about the correctness of C—we can claim that C is $PAC(\epsilon, \delta)$-correct, i.e., with *confidence* δ, the deviation of $L(C)$ from $\mathsf{decision}(\Pi)$ is less than ϵ (we give a proper explanation of the terms in Section 4).

On the other hand, it is straightforward to convert G to a finite automaton B accepting the set of all error decision vectors $\mathsf{decision}(\mathcal{B})$. Intuitively, states of B correspond to nodes of G, the initial state of B corresponds to the initial node of G, and accepting states of B correspond to G's error nodes. An edge from a sequential node is translated to a λ-transition. For a branching node, the edges to its 0- and 1-successors are translated to transitions over symbols 0 and 1 respectively (cf. Section 7).

A high-level overview or our learning procedure is given in Figure 1 (the procedure is similar in structure to the one of [2]). It consists of two main components: The learning algorithm asks the teacher two kinds of questions: *membership* ("Is a given decision vector feasible?") and *equivalence* ("Is a given candidate finite automaton $PAC(\epsilon, \delta)$-correct?") queries. The teacher resolves the queries, at the same time observing whether some of the tested decision vectors corresponds to a feasible error path. By posing these queries, either the learning algorithm iteratively constructs a $PAC(\epsilon, \delta)$-correct approximation of $\mathsf{decision}(\Pi)$ or our procedure finds a feasible error decision vector.

As with other online learning-based techniques, we need to devise a mechanical teacher that answers queries from the learning algorithm. Checking membership queries (i.e., membership in the set $\mathsf{decision}(\Pi)$ of feasible decision vectors) is relatively easy—for example, given a decision vector d, we obtain its corresponding path π by unfolding the CFG G according to d, and use an off-the-shelf solver to decide whether π is feasible or not (cf. Section 6).

When the automata learning algorithm infers a candidate finite automaton C, we need to check whether $L(C)$ approximates $\mathsf{decision}(\Pi)$, i.e., whether C is $PAC(\epsilon, \delta)$-correct. Since we cannot compare $\mathsf{decision}(\Pi)$ with $L(C)$ directly, we employ the sampling-based approximate equivalence technique of PAC learning. While generally unsound, the technique still provides statistical guarantee about the correctness of the inferred model (details are given in Section 5).

4. PAC AUTOMATA LEARNING

Here we explain the PAC automata learning algorithm that we use to find an approximation to $\mathsf{decision}(\Pi)$. Classical PAC automata learning algorithm cannot be used directly for the purpose of program verification. It has to be modified to handle the case when the program contains an error. The classical PAC automata learning algorithm was obtained from modifying the requirement of the exact automata learning algorithm [3]. Our modification follows the same route. In this section, we first describe the classical "exact" automata learning algorithm of regular languages and then describe how to modify it for verification. Then we explain how to relax the requirement of an exact automata learning algorithm to infer an approximation to $\mathsf{decision}(\Pi)$.

4.1 Exact Learning of Regular Languages

Suppose R is a *target* regular language such that its description is not directly accessible. *Automaton learning* algorithms [2, 18, 14, 5] infer automatically a finite automaton A_R recognizing R. The setting of an online learning algorithm assumes a *teacher* who has access to R and can answer the following two types of queries:

- Membership query $Mem(w)$: is the word w a member of R, i.e., $w \in R$?
- Equivalence query $Equ(C)$: is the language of the finite automaton C equal to R, i.e., $L(C) = R$? If not, what is a counterexample to this equality (a word in the symmetric difference of $L(C)$ and R)?

The learning algorithm will then construct a finite automaton A_R such that $L(A_R) = R$ by interacting with the teacher. Such an algorithm works iteratively: In each iteration, it performs membership queries to get information about R from the teacher. Using the results of the queries, it proceeds by constructing a candidate automaton C and, finally, makes an equivalence query $Equ(C)$. If $L(C) = R$, the algorithm terminates with C as the resulting finite automaton A_R. Otherwise, the teacher returns a word w distinguishing $L(C)$ from the target language R. The learning algorithm uses w to modify the conjecture for the next iteration. The mentioned learning algorithms are guaranteed to find a finite automaton A_R recognizing R using a number of queries polynomial to the number of states of the minimal DFA recognizing R. In the rest of the paper, we denote "online automata learning" simply as "automata learning".

4.2 Learning for Program Verification

Under the context of program verification, it may be the case that $\mathsf{decision}(\Pi) \cap L(B) \neq \emptyset$; in such a case, our procedure should return a feasible error path in the program. This is very similar to the setting of *learning-based verification* [11, 9], where the learning algorithm is modified to return a counterexample in case the system contains an error. We modified the used learning algorithm in a similar way. To be more specific, when the classical learning algorithm poses an equivalence query $Equ(C)$, we first check whether there exists a decision vector c such that $c \in L(C) \cap L(B)$ and then test if $c \in \mathsf{decision}(\Pi)$.

1. In case that the two tests identified a decision vector c such that $c \in L(C) \cap L(B)$ and $c \notin \mathsf{decision}(\Pi)$, then c is in the difference of $L(C)$ and $\mathsf{decision}(\Pi)$ and hence a valid counterexample for the classical learning algorithm to refine the next conjecture automaton C.
2. In case that the two tests identified a decision vector c such that $c \in L(C) \cap L(B)$ and $c \in \mathsf{decision}(\Pi)$, then c is a feasible error decision vector and we report c to the user.
3. In case that $L(C) \cap L(B) = \emptyset$, the modified learning algorithm poses an equivalence query $Equ(C)$ to the teacher.

Given a teacher answers membership and equivalence queries about $\mathsf{decision}(\Pi)$, the modified automata learning algorithm has the following properties.

LEMMA 1. *Assume* $\mathsf{decision}(\Pi)$ *is a regular set. The modified automata learning algorithm eventually finds a counterexample* $c \in L(B) \cap \mathsf{decision}(\Pi)$ *when* $L(B) \cap \mathsf{decision}(\Pi) \neq \emptyset$. *It eventually finds a finite automaton recognizing* $\mathsf{decision}(\Pi)$ *when* $L(B) \cap \mathsf{decision}(\Pi) = \emptyset$.

Observe that when the program does not contain any error, the behavior of the modified learning algorithm is identical to the classical one and hence is still an exact automata learning algorithm. Next we explain how to relax the requirements of the exact automata learning algorithm to obtain a PAC automata learning algorithm that is suitable for program verification.

4.3 Probably Approximately Correct Learning

The techniques for learning automata we just discussed in the previous section assume a teacher who has the ability to answer equivalence queries. Such an assumption is, however, invalid in our procedure. Checking $\mathsf{decision}(\Pi) = L(C)$ can be undecidable. Angluin showed in [3] that if we substitute equivalence queries with sampling, we can still make statistical claims about the difference of the inferred set and the target set.

Assume that we are given a probability distribution D over the elements of a universe $\mathcal{U}$, and a hypothesis in the form

$$Prob_{w \in \mathcal{U}|_D}[\neg\varphi(w)] \leq \epsilon.$$

In the hypothesis, the term $Prob_{w \in \mathcal{U}|_D}[\neg\varphi(w)]$ denotes the probability that the formula $\varphi(w)$ is invalid for w chosen randomly from $\mathcal{U}$ according to the distribution D. We call ϵ the *error* parameter and use the term *confidence* to denote the least probability that the hypothesis is correct. We say that $\varphi(w)$ is $PAC(\epsilon, \delta)$-*valid* if $Prob_{w \in \mathcal{U}|_D}[\neg\varphi(w)] \leq \epsilon$ with confidence δ.

In the setting of automata learning, the considered universe is Σ^* and the target regular language is $R \subseteq \Sigma^*$. The task of an equivalence query $Equ(C)$ is changed from checking *exact* equivalence, which we can express as checking that $\forall w \in \Sigma^* : w \notin R \ominus L(C)$ (we use $\ominus$ to denote the symmetric difference operator), to checking *approximate equivalence*, i.e., checking whether the formula

$\varphi(w) = w \notin R \ominus L(C)$ is $PAC(\epsilon, \delta)$-valid. In other words, we check whether $Prob_{w \in \Sigma^*|_D}[w \in R \ominus L(C)] \leq \epsilon$ with confidence δ. For a fixed R and a candidate C, we say that C is $PAC(\epsilon, \delta)$-*correct* if $w \notin R \ominus L(C)$ is $PAC(\epsilon, \delta)$-valid.

The teacher checks the $PAC(\epsilon, \delta)$-correctness of C by picking r samples according to D and testing if all of them are not in $R \ominus L(C)$. For the i-th equivalence query of the learning algorithm, the number of samples q_i needed to establish that C is $PAC(\epsilon, \delta)$-correct is given by Angluin in [2] as

$$q_i = \left\lceil \frac{1}{\epsilon} \left(\ln \frac{1}{1-\delta} + i \ln 2 \right) \right\rceil. \quad (1)$$

Since the inferred set C is guaranteed to be $PAC(\epsilon, \delta)$-correct, this approach is termed *probably approximately correct* (PAC) learning [21].

5. RESOLVING EQUIVALENCE QUERIES BY SAMPLING

The current section discusses how to design a mechanism that the teacher can use for equivalence queries to provide the $PAC(\epsilon, \delta)$-correctness guarantee, as defined in Section 4. Given a probability distribution D over the set of feasible decision vectors $\mathsf{decision}(\Pi)$, we can use D to give a formal definition of the quality of a candidate automaton C. In particular, we use as a measure the probability with which a decision vector chosen randomly from $\mathsf{decision}(\Pi)$ (according to the distribution D) is contained in C.

A sampling mechanism offering such a distribution must satisfy the following conditions:

1. Only decision vectors in $\mathsf{decision}(\Pi)$ are sampled.
2. The samples are *independent and identically distributed* (IID), i.e., the distribution is fixed and the probability of sampling a particular element does not depend on the previously picked samples.

In this paragraph, we introduce the *random input sampling* mechanism. We treat all nondeterministic choices and formal parameters of the program as input variables and assume that all input variables are over finite domains. Each set of initial values of input variables yields a path in the CFG of the program. Based on this observation, random input sampling works by (1) picking uniformly at random a set of initial values for input variables of the program and then (2) obtaining the corresponding decision vector by traversing the CFG of the program using the picked values. The sampling mechanism forms a distribution such that the probability of a decision vector d being chosen is proportional to the number of program paths corresponding to d.

The issue of random input sampling is that it suffers from the well-known fact that coverage of input values is not a good approximation of program path coverage. Depending on the sizes of input domains of program variables, some paths might have only a negligible probability of being selected—for instance, given two 64-bit integers `x` and `y`, the probability of taking the *true* branch in the test `x == 0 && y == 0` is equal to 2^{-128}. The situation gets even worse for input variables over unbounded domains. Even with an extremely high coverage rate of input variables' values, many paths may still not be explored, while other are explored repeatedly. In order to get a sampling mechanism with a better distribution over program paths, we developed a technique that randomly explores program's paths using a concolic tester [12, 19, 7], which is an efficient means for exploring decision vectors corresponding to rare paths.

We describe the technique and prove its properties in the rest of this section.

5.1 Concolic Testing

Concolic testing is a testing approach that explores paths in the CFG of a program while searching for bugs. The algorithm begins with a decision vector generated by randomly picked input values. Then, it finds the next decision vector by flipping some decision made in the chosen path and obtains new input values that lead the program execution according to the new path. This mechanism gives rare paths a much greater chance to be explored. The selection of which decision should be flipped depends on the used *search strategy* of the tester.

In our procedure, we use the concept of a batched sample. A *batched sample* is defined as a set of decision vectors of the size k (where k is a given parameter) obtained from a concolic tester by exploring k paths using its search strategy. We denote D_k the distribution over elements of $(\Sigma^*)^k$ obtained in this way. Our procedure restarts the concolic tester after taking every batched sample. The previous point gives us the guarantee that the probability of taking each batched sample remains the same during the execution our procedure (we assume that the concolic tester does not keep state information between its restarts), and that the distribution is IID and, therefore, meets condition 2 defined above. The principal functioning of concolic testers guarantees that condition 1 is also met.

5.2 Generalized Stochastic Equivalence

In this section, we show that our sampling mechanism using batched samples has the property required for the $PAC(\epsilon, \delta)$-correctness guarantee of the learning algorithm given in Section 4.3.

Recall that for the set of feasible decision vectors of a program $\mathsf{decision}(\Pi)$ and a candidate automaton C inferred by the learning algorithm using some distribution D over Σ^*, if the teacher gives the answer *yes* for the equivalence query $Equ(C)$, it guarantees with confidence δ that

$$Prob_{w \in \Sigma^*|_D}[w \in \mathsf{decision}(\Pi) \ominus L(C)] \leq \epsilon. \quad (2)$$

Since our sampling technique uses batched samples from the universe $\mathcal{U}_k = (\Sigma^*)^k$ w.r.t. the distribution D_k instead of elements of Σ^* and distribution D, we need to change the provided guarantee in our modification of the learning algorithm. If our algorithm answers *yes*, it guarantees that

$$Prob_{S \in \mathcal{U}_k|_{D_k}}[\exists w \in S : w \in \mathsf{decision}(\Pi) \ominus L(C)] \leq \epsilon \quad (3)$$

with confidence δ (we hereafter use the term $PAC(\epsilon, \delta)$-correct to denote this form of guarantee).

When a teacher receives an equivalence query $Equ(C)$, it uses a concolic tester to obtain q_i (given in (1)) batch samples. For each batch sample S, the teacher checks if there exists a decision vector $w \in S$ such that $w \notin L(C)$ (by definition $w \in \mathsf{decision}(\Pi)$). The teacher answers *yes* if there is no such w. Otherwise, the teacher checks if w is an error decision vector and either reports w as a feasible error decision vector or returns w to the learning algorithm to refine the next conjecture.

The following lemma shows that if we use the number q_i batched samples for testing the equivalence, we obtain the

modified $PAC(\epsilon, \delta)$-correctness guarantee from (3).

LEMMA 2. *Let ϵ and δ be the error and confidence parameters, and R be the target language. If no decision vector $w \notin L(C)$ is found in the q_i batched samples, then it holds that C is $PAC(\epsilon, \delta)$-correct.*

Based on the fact that $L(C) \cap L(B) = \emptyset$ (the property of the modified learning algorithm in Section 4.2) and the lemma above, we obtain the following corollary.

COROLLARY 1. *Let ϵ and δ be the error and confidence parameters, and R be the target language. If no decision vector $w \notin L(C)$ is found in q_i batched samples, then it holds that the program is $PAC(\epsilon, \delta)$-correct.*

6. RESOLVING MEMBERSHIP QUERIES

In this section, we describe how a membership query $Mem(d)$ in the algorithm in Figure 1 is discharged by the teacher. Let Π be the set of feasible paths of a CFG G. When the learning algorithm asks a membership query $Mem(d)$, the teacher needs to check whether the decision vector d is in the set of feasible decision vectors $\mathsf{decision}(\Pi)$. To answer the query, the teacher first constructs a path $\pi = \langle v_0, f_1, v_1, f_2, v_2, \ldots, v_{m-1}, f_m, v_m \rangle$ in G such that

- there are exactly $|d|$ occurrences of branching nodes in the prefix $\langle v_0, f_1, v_1, f_2, v_2, \ldots, v_{m-1} \rangle$ of π,
- if v_k is the j-th branching node in π, it holds that $\mathsf{decision}(v_k, f_{k+1}) = d[j]$, and
- v_{m-1} is a branching node.

Recall that π is a feasible path if and only if $\varphi = \bigwedge_{j=1}^{m} f_j{}^{\langle j \rangle}$ is satisfiable. Therefore, the teacher can simply construct the formula φ from the path π and check its satisfiability using an off-the-shelf constraint solver.

Alternatively, the teacher can check feasibility by translating the path into a sequence of program statements (with conditions removed and substituted by assertions on the values of the conditions) and asking a symbolic executor or software model checker whether the final line of the constructed program is reachable. The alternative option is easier to implement but usually suffers from some performance penalty.

7. ERROR DECISION VECTORS

Let $\mathcal{B}$ be the set of error paths in a CFG. In this section, we show how we construct a finite automaton accepting the set of all error decision vectors $\mathsf{decision}(\mathcal{B})$ of the given CFG. This automaton will later be intersected with the automaton representing the set of feasible paths to determine whether the CFG contains a feasible error path.

DEFINITION 1. *Let $G = (V_b \cup V_s, E, v_i, v_r, V_e, \mathcal{X}_{FP})$ be a CFG. We define the* error trace automaton *for G as the finite automaton $B = (\{0,1\}, V_b \cup V_s, v_i, \Delta_E, V_e)$ where Δ_E is defined as follows:*

- *$(v, 0, v'_0) \in \Delta_E$ if $v \in V_b \setminus V_e$, $(v, f_0, v'_0) \in E$, and v'_0 is the 0-successor of v;*
- *$(v, 1, v'_1) \in \Delta_E$ if $v \in V_b \setminus V_e$, $(v, f_1, v'_1) \in E$, and v'_1 is the 1-successor of v;*
- *$(v, \lambda, v') \in \Delta_E$ if $v \in V_s \setminus V_e$ and $(v, f, v') \in E$; and*
- *$(v, 0, v), (v, 1, v) \in \Delta_E$ if $v \in V_e$.*

Informally, B contains a state for every node and a transition for every edge of G. It reads a symbol in each state corresponding to a branching node and performs λ-transitions for states corresponding to sequential nodes. For every error node, B reads all remaining symbols and accepts the input word. It is straightforward to see that B accepts exactly the set of decision vectors corresponding to error paths in G.

LEMMA 3. *Let $G = (V, E, v_i, v_r, V_e, \mathcal{X}_{FP})$ be a CFG and $\mathcal{B}$ the set of error paths in G. Let B be the error trace automaton for G. It holds that $L(B) = \mathsf{decision}(\mathcal{B})$.*

In Section 9, we describe an extension of our procedure to programs with procedure calls. Because representing the set of error decision vectors using a finite automaton is in this setting imprecise, the section also discusses an extension that represents the set of error paths in a program with procedure calls using pushdown automata.

8. THE MAIN PROCEDURE

We summarize our procedure in this section. Let G be the CFG of the verified program, k be the size of a batched sample, ϵ be the error parameter, and δ be the confidence parameter. The goal of our procedure is to either find a feasible error decision vector of G or show that G is $PAC(\epsilon, \delta)$-correct. In the latter case, we also accompany our answer with a $PAC(\epsilon, \delta)$-correct regular representation of the set of feasible decision vectors of G. Let Π be the set of feasible paths of G, D_k be the distribution defined by our sampling mechanism (cf. Section 5), and $L(B)$ be the set of error decision vectors of G (cf. Section 7).

A detailed flow chart of our procedure can be found in Figure 2. First, the bottom part of the figure describes our learning algorithm. We extend the online automata learning algorithm with two additional tests for verification, as described in Section 4.2. In particular, when the automata learning algorithm outputs a candidate C, before sending teacher the equivalence query $Equ(C)$, we first test whether $L(C)$ contains a *feasible* error decision vector c. In case it does, we report c as an error. Otherwise, in the case c is both in $L(C)$ and $L(B)$ but is not feasible, we return c to the learning algorithm to further refine the conjecture.

The top part of the figure describes our design of a mechanical teacher. The task of the teacher is to answer queries from the learning algorithm. Membership queries of the form $Mem(w)$ can be answered by constructing the path corresponding to the decision vector w and the associated path formula, which is then solved using a constraint solver (cf. Section 6). Equivalence queries, on the other hand, are discharged using a concolic tester by checking whether there is a decision vector s in the set of batched samples S such that it does not belong to the language of C (cf. Section 5). If no such a decision vector exists, we conclude that the program is $PAC(\epsilon, \delta)$-correct. Otherwise, we test whether $s \in L(B)$; if this holds, we report that we have found a feasible error decision vector. In the case $s \notin L(B)$, it holds that s is a feasible decision vector in $\mathsf{decision}(\Pi)$ but not in the language of the current conjecture $L(C)$. If this happens, we return s to the automata learning algorithm to refine the conjecture and continue with the next iteration of the learning loop.

In general, our procedure is not guaranteed to terminate. When the procedure terminates and reports an error (either by the teacher or the learning algorithm), a feasible error

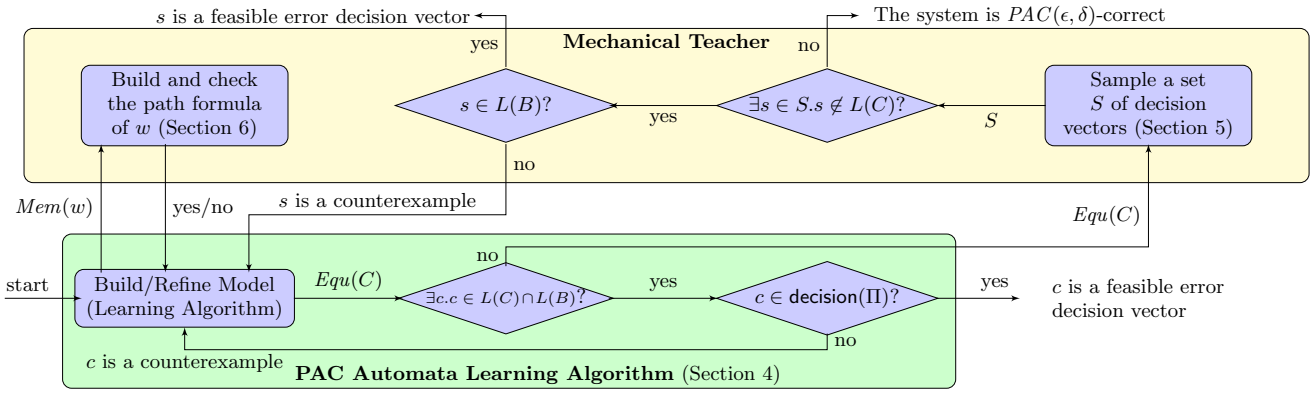

Figure 2: A detailed flow chart of our verification procedure

decision vector is found and the program is reported to be incorrect. If the teacher approves an approximate finite automaton C, our procedure reports that C is an approximate model of decision(Π) w.r.t. the $PAC(\epsilon, \delta)$-correctness guarantee, which, in turn, implies that the program is $PAC(\epsilon, \delta)$-correct. From Lemma 2, we have the following theorem

THEOREM 1. *Let ϵ and δ be the error and confidence parameters respectively and Π be the set of feasible paths of a given CFG G. If our procedure terminates with an approximate finite automaton C, the program is $PAC(\epsilon, \delta)$-correct.*

Moreover, we obtain the following corollary.

COROLLARY 2. *Suppose our procedure reports a program P is $PAC(\epsilon, \delta)$-correct. If we run the concolic tester with the same search strategy and batch size used in our procedure on P, with confidence δ, the concolic tester will find an error with a probability less than ϵ.*

Thanks to the properties of the modified automata learning algorithm, when decision(Π) is a regular set, our algorithm is guaranteed to terminate and either (1) return a counterexample $c \in L(B) \cap$ decision(Π) or (2) find an approximate model of decision(Π) that is disjoint with $L(B)$.

9. HANDLING PROCEDURE CALLS

In this section, we extend our formalism of CFGs to handle programs with multiple procedures. We use a PDA to represent error decision vectors in this setting. The issue of using finite automata to represent error decision vectors in the said setting is that when returning from a procedure call, a finite automaton cannot remember an unbounded number of return points (in the case of recursive procedures). Therefore, an overapproximation, such as a nondeterministic jump to any possible return point, needs to be used. The said overapproximation is, however, too imprecise and yields numerous spurious errors. In contrast, PDAs can represent the set of error decision vectors precisely.

On the other hand, we still use a finite automaton to represent the approximation of the set of feasible decision vectors decision(Π). As a consequence, except that we need to use PDA operations instead of FA operations and handle procedure calls in the membership queries, all other components remain unchanged for the setting of multiple procedures.

9.1 Extending CFGs with Procedure Calls

Assume the set of procedure names $\mathcal{P}$. A *CFG with calls* (CFGC) is defined as a graph $G = (V, E, v_i, v_r, V_e, \mathcal{X}_{FP})$ where V, v_i, v_r, V_e, and $\mathcal{X}_{FP}$ are defined in the same way as for a CFG, and $E \subseteq (V \times \mathcal{T}[\mathcal{X}, \mathcal{F}] \times V) \cup (V \times (\mathcal{P} \times \mathcal{T}[\mathcal{X}, \mathcal{F}] \times \mathcal{T}[\mathcal{X}, \mathcal{F}]) \times V)$ is an extended set of edges that apart from local CFG edges (v, f, v') for $f \in \mathcal{T}[\mathcal{X}, \mathcal{F}]$ also contains *procedure call edges* $e = (v, (p, g_{in}, g_{out}), v')$ for $(p, g_{in}, g_{out}) \in \mathcal{P} \times \mathcal{T}[\mathcal{X}, \mathcal{F}] \times \mathcal{T}[\mathcal{X}, \mathcal{F}]$ and sequential nodes v. The g_{in} and g_{out} components of e correspond to formulae for passing actual values to formal parameters of p (formula g_{in}) and passing the return value of p back to the caller procedure (formula g_{out}).

In this extension, we define a *program* as a set of CFGCs $prog = \{G_1, \ldots, G_n\}$ together with a (bijective) mapping $cfgc_{prog} : \mathcal{P} \to prog$ that assigns procedure names to CFGCs. We abuse notation and use $prog$ to denote $cfgc_{prog}$, i.e., $prog(p)$ denotes the CFGC of a procedure p in a program $prog$. We assume that all CFGCs in $prog$ have pairwise disjoint sets of nodes, and further assume an entry point $\texttt{main} \in \mathcal{P}$.

In this paragraph, we give an informal description of how we extend the definition of a path from a program consisting of single CFG to a program consisting of a set of CFGCs and a dedicated entry point. Given a procedure call edge e in a CFGC G, we call the *inlining* of e in G the CFGC G' obtained from G by substituting e with the CFGC of the called procedure. We use $[\![prog]\!]$ to denote the set of CFGs obtained from $prog(\texttt{main})$ by performing all possible (even recursively called) sequences of inlinings, and removing any left procedure call edges from the output CFGCs. A *path* in $prog$ is then a sequence $\pi = \langle v_0, f_1, v_1, f_2, v_2, \ldots, f_m, v_m \rangle$ such that there exists a CFG $G' \in [\![prog]\!]$ for which it holds that $\pi \in G'$.

9.2 Encoding Error Decision Vectors with Pushdown Automata

In this section, we describe how we construct the PDA encoding the set of error traces in the considered extension. The general idea is the same as the one for the use of finite automata (described in Section 7). The main difference is that we add jumps between CFGCs (corresponding to procedure call edges), which use the stack to remember which state the PDA should return to after the procedure call terminates.

In the following, given a CFGC $G = (V, E, v_i, v_r, V_e, \mathcal{X}_{FP})$, we use $V(G), E(G), \ldots, \mathcal{X}_{FP}(G)$ to denote the corresponding components of G, and, moreover, we use $V_s(G)$ and $V_b(G)$ to denote the set of sequential and branching nodes of G respectively. Consider a program $prog = \{G_1, \ldots, G_n\}$.

We construct the *error path automaton* as the PDA $B_P = (\{0,1\}, Q, Q, q_i, \Delta, F)$ in the following way:

- $Q = V(G_1) \cup \cdots \cup V(G_n)$,
- $q_i = v_i(G_k)$ such that $prog(\texttt{main}) = G_k$,
- $F = V_e(G_1) \cup \cdots \cup V_e(G_n)$,
- $\Delta = \Delta_1 \cup \cdots \cup \Delta_n$ where every Δ_j is defined as follows:
  - $(v, 0, v'_0) \in \Delta_j$ if $v \in V_b(G_j) \setminus V_e(G_j)$, $(v, f_0, v'_0) \in E(G_j)$, and v'_0 is the 0-successor of v;
  - $(v, 1, v'_1) \in \Delta_j$ if $v \in V_b(G_j) \setminus V_e(G_j)$, $(v, f_1, v'_1) \in E(G_j)$, and v'_1 is the 1-successor of v;
  - $(v, \lambda, v') \in \Delta_j$ if $v \in V_s(G_j) \setminus V_e(G_j)$ and $(v, f, v') \in E(G_j)$;
  - $(v, 0, v), (v, 1, v) \in \Delta_j$ if $v \in V_e(G_j)$; and
  - $(v, [\lambda; \lambda/v'], v_i(G_k)), (v_r(G_k), [\lambda; v'/\lambda], v') \in \Delta_j$ if $v \in V_s(G_j) \setminus V_e(G_j)$, $(v, (p, g_{in}, g_{out}), v') \in E(G_j)$, and $G_k = prog(p)$.

LEMMA 4. *Let prog be a program, $\mathcal{B}$ the set of error paths of prog, and B_P be the error path PDA for prog. Then it holds that $L(B_P) = \mathsf{decision}(\mathcal{B})$.*

10. IMPLEMENTATION

We created a prototype tool PAC-MAN that implements the verification procedure described in this paper. The tool uses several third-party libraries and tools. First, it uses CIL (C Intermediate Language) [17] to convert the verified C program to a set of CFGCs, from which we construct the error trace pushdown automaton B_P. Further, we use the LIBAMORE++ library [16, 6] to perform operations of automata, such as testing their membership and emptiness, or computing their intersection.

For learning automata, we use the implementation of various learning algorithms within the LIBALF library [6]. Membership queries are discharged using a concolic tester, mentioned as an alternative option in Section 6. Given a decision vector, our tool uses the CFG of the program to generate a path corresponding to the decision vector. The path is passed in the form of a sequence of program statements to the software model checker CPACHECKER [4], which checks its feasibility. It is possible to switch the model checker with other checkers, such as CBMC [10].

To deal with equivalence queries, we modified the concolic tester CREST [7] to generate a batch of k decision vectors, as described in Section 5. As CREST may fail to generate the decision vector of a program execution when the execution terminates abnormally, we modified CREST to take a finite prefix of the execution in this case. One issue of CREST that we encountered is that when it processes a condition composed using Boolean connectives, it expands the condition into a cascade of `if` statements corresponding to the Boolean expression, making the program longer and harder to learn. We addressed this by modifying CREST so that it can process conditions with Boolean connectives inside without expanding them, and in this way we increased the performance and precision of the analysis.

We also implemented the following three optimizations to improve the performance of the prototype.

Intersection with Bad Automaton.

Recall that our modified learning algorithm (described in Section 4.2) first checks whether the intersection of the language of the conjecture $L(C)$ and the bad language $L(B_P)$ is empty. Checking emptiness of a PDA is, however, more difficult than that of a finite automaton. To speed up the procedure, we build a finite automaton B_O that over-approximates the error language and always first checks whether $L(C) \cap L(B_O) = \emptyset$, which is an emptiness test for finite automata. We check $L(C) \cap L(B_P) = \emptyset$ only for the cases that the previous test fails.

Counterexample from the Learning Algorithm.

When an equivalence query returns a counterexample c, automata learning algorithms usually do not guarantee that c is not a valid counterexample in the next conjecture automaton. In our preliminary experiments, we found out that it happens very often that the mechanical teacher returns the same counterexample in several consecutive iterations. Therefore, we decided to check whether c is still a valid counterexample (by a membership query) for the learning algorithm before proceeding to the emptiness test. In the case c is valid, it will be immediately returned to the learning algorithm to refine the conjecture.

Handling Membership Queries.

The main bottleneck of our approach is the time spent for membership queries. In our implementation, the software model checker CPACHECKER is used to check whether a path is feasible. For each membership query, if we invoke CPACHECKER with a system call, a Java virtual machine will be created and the components of CPACHECKER need to be loaded, which is very time consuming. To make membership queries more efficient, we modified CPACHECKER to run in a server mode so that it can check more than a single path without being re-invoked.

11. EXPERIMENTS

This section presents our experimental results to justify the claims made in this paper. We evaluated the performance of our prototype using the recursive category of SV-COMP 2015 [1] as the benchmark. The recursive category consists of 24 non-trivial examples such as Ackermann, McCarthy 91, and Euclidean algorithms. Among eight participating tools, only two can solve 20 or more examples correctly. Among the 24 examples, 8 of them contain an error. We performed our experiments with the error parameter $\epsilon = 0.1$, confidence $\delta = 0.9$, and size of batched samples $k = 10$. We ran our prototype on each example three times in all experiments. The provided statistical data were calculated based on the average of the three runs unless explicitly stated otherwise. We set the timeout to 900 s to match the rules of SV-COMP 2015.

11.1 Comparison of Learning Algorithms

We evaluated our approach with different automata learning algorithms implemented within the LIBALF library. There are five active online automata learning algorithms implemented in LIBALF: Angluin's L^* [2], L^*-columns, Kearns/Vazirani (KV) [14], Rivest/Schapire (RS) [18], and NL^* [5]. Among the search strategies provided by CREST, we chose the *random branch* strategy. The experimental results are in Table 1.

Table 1: Comparison of learning algorithms

	Algorithms				
	KV	L^*	L^*-col.	RS	NL^*
Verified	15	9.67	10	11.33	8.33
Bug found	6	6.33	6	6.33	6
by Bad	4	4.67	4	3	4.33
by CREST	2	1.67	2	3.33	1.67
False positives	0	0	1	0.33	1
False negatives	2	1.67	1.33	1.67	1
Timeouts	1	6.33	5.67	4.33	7.67
# of *Mem* queries	2896	8898	10071	15377	14463
# of *Equ* queries	548	78	77	367	67
Total time [s]	2406	6668	6565	5786	7972
Mem queries time	30 %	59 %	58 %	63 %	70 %

Table 2: Comparison of search strategies of CREST

	Search Strategy	
	RBS	*CDS*
Verified	15	15
Bug found	6	6
False positives	0	0
False negatives	2	2
Timeouts	1	1
# of *Mem* queries	2896	2362
# of *Equ* queries	548	463
Total time [s]	2406	2013
Time for one sample [s]	0.75	0.82

The results show that KV is the algorithm with the best performance—it solved 21 out of the 24 examples. Our technique solves more than any participant in the recursive category of SV-COMP 2015 but the winner. The main reason for the performance difference is that KV uses a tree-based data structure to store query results. Compared to other learning algorithms that use table-based structures, KV requires much less number of membership queries to maintain the consistency of the tree-based structure. For all learning algorithms except RS, the number of error paths found by the emptiness test of the intersection of the conjecture and the bad automaton is more than that found by CREST. In our experiments, the time spent for membership queries is usually the performance bottleneck. Table 1 shows that membership queries took 30 % of the total execution time for KV and at least 58 % for other algorithms.

11.2 Comparison of Search Strategies

We also evaluated the performance of our algorithm against different CREST search strategies. According to [7], the most efficient ones are *random branch* strategy (RBS for short) and *control-flow directed* strategy (CDS for short). Therefore, we tested the performance of our prototype using these two strategies. We selected KV as the learning algorithm in this experiment. The results are shown in Table 2.

Table 2 shows that although the average time for taking one sample with CDS is more than with RBS, the total time is less. The main reason is that CDS explores untouched branching points more aggressively than RBS but requires more overhead. Our experiments conform the results in [7].

11.3 Evaluation of CREST with Restarts

To justify our modification to the $PAC(\epsilon, \delta)$-correctness guarantee given in Section 5.2, we show in the experiment below that running CREST in batches does not decrease its bug-hunting capabilities. We compared the performance of CREST with two different scenarios: (1) restart after each 10 decision vectors and (2) never restart. We performed the experiment on the 8 buggy examples in the recursive category and calculated in how many examples CREST found a bug for within the timeout period. In Table 3, we chose RBS as the search strategy. We also tried the experiments with the CFG strategy and got a similar result. We list the worst result for scenario (1) and the best result for scenario (2) that we received in our three runs. We found out that the worst runs in scenario (1) can still find with a little overhead all bugs found by the best runs in scenario (2).

Table 3: Evaluation of CREST with and without restart. For each example, the number of batches and the number of iterations used to find bugs are obtained respectively from the worst run in scenario (1) and from the best run in scenario (2).

	Settings	
Examples	*Batch Size 10*	*Never Restart*
Ackermann02	batch 3	iteration 14
Addition02	batch 1	iteration 2
Addition03	Timeout	Timeout
BallRajamani-SPIN2000	batch 1	iteration 1
EvenOdd03	batch 1	iteration 2
Fibonacci04	batch 4	iteration 2
Fibonacci05	Timeout	Timeout
McCarthy91	batch 1	iteration 2

Table 4: Comparison of CREST search strategies in terms of the quality of the learned automata. The total number of tested batches is 1500.

		Equ query strategy	
		RBS	*CDS*
Evaluation (RBS)	Accepted	1487	1473
	Ratio	99.13 %	98.2 %
Evaluation (CDS)	Accepted	1489	1500
	Ratio	99.27 %	100 %

11.4 Evaluating Quality of Learned Automata

Besides the performance in terms of the running time, we also compared the quality of the learned automata produced by our prototype using the two strategies for the 15 successfully verified bug-free examples. To evaluate the quality of the learned automata, for each example, we ran CREST with the given search strategy to get 100 batched samples and tested how many of the them are all accepted by the learned automaton. The average values of the runs are shown in Table 4 where evaluation strategies are strategies used to generate the testing batched samples. The table shows that the quality of the automata learned with the two strategies is almost the same. Also, observe that the guarantee of our procedure is that the sample coverage is higher than 90 %. Our experimental results show that the quality of the automata produced by our procedure matches the theoretical expectations.

Finally, we tested how many words generated by CREST are not covered in the automata learned with the KV algorithm and RBS. Again, we ran CREST with RBS in two scenarios: (1) restart after each 10 decision vectors and (2) never restart. For each learned automaton (there were 15) and each scenario, we generated 1000 decision vectors

and checked how many of them are accepted by the automaton. In total, for scenario (1), we observed 1487 accepted batches of size 10 (for the total of 15 000 tested vectors), yielding the correctness 99.13 %. For scenario (2), we observed 14 977 accepted vectors, for the correctness 99.86 %. We notice that no matter which strategy we use, the learned automaton accepts over 99 % of the decision vectors produced by CREST.

12. DISCUSSION

There are several advantages of having a program model with statistical guarantees. For instance, the model can be reused for verifying a different set of properties of the program. Assume that the new property to be verified is described as an error path automaton B' and C is the learned automaton. If $L(B') \cap L(C) = \emptyset$, we verified the program with the new property and the same $PAC(\epsilon, \delta)$-correctness guarantee. For the case that there exists a decision vector $w \in L(B') \cap L(C)$, we test whether w is feasible and either report that w is a feasible error decision vector w.r.t. B' or continue the learning algorithm with w as a counterexample for refining the next conjecture.

In this paper we focus on checking validity of program assertions. The verification step is handled by making an intersection of the conjecture automaton C and the error path automaton B and testing its emptiness. This procedure can be generalized to more sophisticated safety properties by replacing the tests $L(C) \cap L(B) = \emptyset$ and $s \in L(B)$ with other tests. For example, we can check the property "the program contains at most 10 consecutive 1-decisions on any path" with a statistical guarantee of the correctness of the received answer. By extending the alphabet $\{0, 1\}$ with program labels, one can also check temporal properties related to those labels, e.g., "label A should be reached within 10 decisions after label B is reached."

One possible extension of our work is to learn sequences of feasible function calls instead of decision vectors. This might lead to a more compact model in contrast to the current approach. However, in this case the alphabet of the model to be learned will be all function names in the program, which is usually much larger than 2, the size of the alphabet in our work. Moreover, in this setting, it is much harder to answer membership queries; a program path composed of function calls might perform a complex traversal through loops and branches in between the calls, making the problem of checking feasibility of a program path already undecidable.

One benefit of our approach is that, in principle, it can be extended to black box system verification and model synthesis. By observing the behavior of the environment, we may find some pattern (e.g., some statistical distribution) of the inputs and then, based on that, design a sampling mechanism. Under the assumption that the behavior of the environment remains unchanged, we can verify or synthesize the model of the system w.r.t. the given sample distribution.

13. RELATED WORKS

Exact automata learning algorithm was first proposed by Angluin [2] and later improved by many people [2, 18, 14, 5]. The concept of probably approximately correct (PAC) learning was first proposed by Valiant in his seminal work [21]. The idea of turning an exact learning algorithm to a PAC learning algorithm can be found in Section 1.2 of [3].

Applying PAC learning to testing has been considered before [22, 13]. The work in [13] considers a program that manipulates graphs and check if the output graph of the program has properties such as being bipartite, k-colorable, etc. Our work considers assertion checking, which is more general than the specialized properties. The work [22] considers more theoretical aspects of the problem. The author estimates the maximal number of queries required to infer a model of a black box machine. The context is quite different, e.g., the work does not discuss how to sample according to some distribution efficiently to produce the desired guarantee (bounded path coverage) as we do in this paper.

The L^* algorithm has been used to infer the model of error traces of a program. In [8], instead of decision vectors, the authors try to learn the sequences of function calls leading to an error. Their teacher is implemented using a bounded model checker and hence can only guarantee correctness up to a given bound. The authors do not make use of the PAC learning technique as we did in this work.

Both our approach and statistical model checking [20, 15, 23] provide statistical guarantees. As mentioned in the introduction, statistical model checking assumes a given model while our technique generates models of programs with statistical guarantee. Those models can be analyzed using various techniques and reused for verifying different properties.

APPENDIX

A. AN INLINING IN A CFGC

Consider a procedure call edge $e = (v, (p, g_{in}, g_{out}), v')$ of $G_j = (V_j, E_j, v_{ij}, v_{rj}, V_{ej}, \mathcal{X}_{FPj})$ for $1 \leq j \leq n$. Further assume that $prog(p) = G_k = (V_k, E_k, v_{ik}, v_{rk}, V_{ek}, \mathcal{X}_{FPk})$ for $1 \leq k \leq n$. The *inlining* of e in *prog* is the program $prog^{\#} = \{G_1, \ldots, G_{j-1}, G_j^{\#}, G_{j+1}, \ldots, G_n\}$ such that $G_j^{\#} = (V_j \cup V_k^{\#}, E^{\#}, v_{ij}, v_{rj}, V_{ej} \cup V_{ek}^{\#}, \mathcal{X}_{FPj})$ where $V_k^{\#}$ is a set of fresh nodes (i.e., $V_k^{\#}$ is disjoint from the set of nodes of any CFGC in *prog*) and $V_{ek}^{\#} \subseteq V_k^{\#}$. Moreover, there exist bijections $\sigma : V_k^{\#} \to V_k$ and $\sigma_e : V_{ek}^{\#} \to V_{ek}$ such that $\sigma_e \subseteq \sigma$. The set of edges $E^{\#}$ is defined as $E^{\#} = (E_j \setminus \{e\}) \cup E_k^{\#} \cup \{(v, g_{in}, v_{ik}^{\#}), (v_{rk}^{\#}, g_{out}, v')\}$ where $E_k^{\#} = \{(v_k^{\#}, f, v_k^{\#\prime}) \mid (\sigma(v_k^{\#}), f, \sigma(v_k^{\#\prime})) \in E_k\}$.

Let $inln(prog)$ be the smallest (potentially infinite) set that contains *prog* and is closed w.r.t. inlinings in `main`, i.e., if $inln(prog)$ contains a program $prog^{\#}$ with a procedure call edge e in the CFGC $prog^{\#}(\texttt{main})$, it also contains the inlining of e in $prog^{\#}$. We abuse notation and use σ uniformly to denote for all programs $prog^{\#} \in inln(prog)$ the mapping of the nodes of the CFGCs in $prog^{\#}$ to their original nodes in *prog* (for nodes V of *prog*, we assume that $\sigma|_V = \mathrm{id}$, i.e., that the restriction of σ to V is the identity relation). We denote as $[\![prog]\!]$ the set of CFGs (without procedure call edges) obtained by starting from the set $inln(prog)$, collecting the CFGCs of `main` functions of all inlinings of *prog* into the set $M = \{prog^{\#}(\texttt{main}) \mid prog^{\#} \in inln(prog)\}$, and, finally, transforming the CFGCs of M into CFGs of $[\![prog]\!]$ by substituting each procedure call edge $(v^{\#}, (p^{\#}, g_{in}^{\#}, g_{out}^{\#}), v^{\#\prime})$ with the edge $(v^{\#}, \mathsf{ff}, v^{\#\prime})$.

We extend the definition of a path as follows: A *path* in *prog* is a sequence $\pi = \langle v_0, f_1, v_1, f_2, v_2, \ldots, f_m, v_m \rangle$ such that there exists a CFG $G' \in [\![prog]\!]$ and a CFG path $\pi' = \langle v_0', f_1, v_1', f_2, v_2', \ldots, f_m, v_m' \rangle$ in G' such that $\forall 0 \leq j \leq m : v_j = \sigma(v_j')$.

14. REFERENCES

[1] Software verification competition 2015, `http://sv-comp.sosy-lab.org/2015/`.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[3] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[4] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. of CAV'11*, pages 184–190. Springer, 2011.

[5] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of NFA. In *Proc. of IJCAI'09*, pages 1004–1009. IJCAI, 2009.

[6] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In *Proc. of CAV'10*, volume 6174 of *LNCS*, pages 360–364. Springer, 2010.

[7] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proc. of ASE'08*, pages 443–446. IEEE Computer Society, 2008.

[8] Martin Chapman, Hana Chockler, Pascal Kesseli, Daniel Kroening, Ofer Strichman, and Michael Tautschnig. Learning the language of error. In *Proc. of ATVA'15*, LNCS. Springer, 2015. To appear.

[9] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning minimal separating DFA's for compositional verification. In *Proc. of TACAS'09*, pages 31–45, 2009.

[10] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[11] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Proc. of TACAS'03*, pages 331–346, 2003.

[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proc. of PLDI'05*, pages 213–223, 2005.

[13] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of ACM*, 45(4):653–750, July 1998.

[14] Michael J. Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. MIT Press, 1994.

[15] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *Proc. of RV'10*, volume 6418 of *LNCS*, pages 122–135. Springer, 2010.

[16] Oliver Matz, Axel Miller, Andreas Potthoff, Wolfgang Thomas, and Erich Valkema. Report on the program amore. 1995.

[17] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of CC'02*, pages 213–228, 2002.

[18] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[19] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proc. of FSE'05*, pages 263–272, 2005.

[20] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *Proc. of CAV'04*, volume 3114 of *LNCS*, pages 202–215. Springer, 2004.

[21] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[22] Neil Walkinshaw. Assessing test adequacy for black-box systems without specifications. In *Proc. of ICTSS'11*, volume 7019 of *LNCS*, pages 209–224. Springer, 2011.

[23] Paolo Zuliani, André Platzer, and Edmund M. Clarke. Bayesian statistical model checking with application to Stateflow/Simulink verification. *Formal Methods in System Design*, 43(2):338–367, 2013.