Journal of Artificial Intelligence Research () Submitted ; published

Learning the Language of Software Errors*

Hana Chockler HANA.CHOCKLER@KCL.AC.UK
Department of Informatics,
King's College London

Pascal Kesseli PASCAL.KESSELI@DIFFBLUE.COM
Daniel Kroening KROENING@CS.OX.AC.UK
Department of Computer Science,
University of Oxford, UK

Ofer Strichman OFERS@IE.TECHNION.AC.IL
Information Systems Engineering,
Technion, Haifa, Israel

Abstract

We propose to use algorithms for learning deterministic finite automata (DFA), such as Angluin's L^* algorithm, for learning a DFA that describes the possible scenarios under which a given program error occurs. The alphabet of this automaton is given by the user (for instance, a subset of the function call sites or branches), and hence the automaton describes a user-defined abstraction of those scenarios. More generally, the same technique can be used for visualising the behavior of a program or parts thereof. It can also be used for visually comparing different versions of a program (by presenting an automaton for the behavior in the symmetric difference between them), and for assisting in merging several development branches. We present experiments that demonstrate the power of an abstract visual representation of errors and of program segments, accessible via the project's web page. In addition, our experiments in this paper demonstrate that such automata can be learned efficiently over real-world programs. We also present *lazy learning*, which is a method for reducing the number of membership queries while using L^*, and demonstrate its effectiveness on standard benchmarks.

1. Introduction

Many automated verification tools produce a counterexample trace when an error is found. These traces are often unintelligible because they are too long (an error triggered after a single second can correspond to a path with millions of states), too low-level, or both. Moreover, a trace focuses on just one specific scenario. Thus, error traces are frequently not general enough to help focus the attention of the programmer on the root cause of the problem.

A variety of methods have been proposed for the *explanation of counterexamples*, such as finding similar paths that satisfy the property (Groce et al., 2006) and analysing causality (Beer et al., 2012), but these focus on a single counterexample. The analysis of *multiple*

*. This work was supported in part by the Google Faculty Research Award 2014. A preliminary version of the first part of the paper appeared in the Proceedings of the 13th International Symposium on Automated Technology for Verification and Analysis (ATVA 2015).

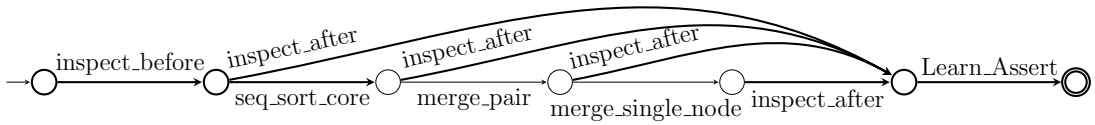


Figure 1: An abstract description of a merge-sort program, where the letters are the function calls.

counterexamples has been suggested in the hardware domain by Copty et al. (2003), who propose to compute all counterexamples and present those states that occur in all of them to the user. The analysis of multiple counterexamples has also been suggested in the context of a push-down automaton (PDA) (representing software) and a deterministic finite automaton (DFA) (representing a negated property) by Basu et al. (2003), who describe the generation of all loop-free counterexamples of a certain category, and the presentation of them to the user in a tree-like structure. In the software domain, another notable example is the model checker MS-SLAM, which reports multiple counterexamples if they are believed to relate to different causes (Ball et al., 2003). Each example is 'localized' by comparing it to a trace that does not violate the property.

We suggest a method (and a corresponding tool) that presents the user a DFA that summarizes all the ways (up to a given bound, as will be explained) in which an assertion can fail. Furthermore, the alphabet of this automaton is user-defined and corresponds to events in the program execution. For example, the user can specify that a call to a function `f` in a specific location is a letter and an entry to a specific branch is a different letter. With a small number of such user-defined events, the traces of the program that lead to an assertion failure can be visualized concisely. We argue that this combination of user-defined abstraction with a compact representation of multiple counterexamples addresses all three problems mentioned above. Moreover, the same idea can be applied to *describing* a program or, more realistically, parts of a program by adding an '`assert(false)`' at the end of the sub-program to be explained. Figure 1, for instance, gives an automaton that describes the operation of a merge-sort program in terms of its possible function calls.[1] We obtained it by inserting such a statement at the end of the `main` function.

Our method is based on a DFA-learning algorithm, such as L^* (Angluin, 1987b), RS (Rivest & Schapire, 1993), COMFORT (Chaki & Strichman, 2008), KV (Kearns & Vazirani, 1994), DT (Howar, 2012) or TTT (Isberner et al., 2014). These algorithms learn a minimal DFA that captures the (regular) language of a model $\mathcal{U}$ over a given alphabet Σ, the behavior of which is communicated to the learning algorithm via interface functions called the 'teacher' and the 'oracle'. In the discussion that follows we will simply call this algorithm the *learner*, as our methods are orthogonal to the learning algorithm.

The learner asks the teacher *membership* queries over Σ, namely whether $w \in \mathcal{U}$, where w is a word and $\mathcal{U}$ is the language (the model), and *conjecture* queries, namely whether for a given DFA $\mathcal{A}$, $L(\mathcal{A}) = \mathcal{U}$. The number of queries that the algorithm performs is polynomial in the size of the alphabet, in the number of states of the resulting minimal DFA, and in the length of the longest counterexample (feedback) to a conjecture query returned by the oracle (see Sec. 2 for a detailed description).

1. Source code for all examples we use is available online (*http://www.cprover.org/learning-errors/*, 2015).

The use of learners in the verification community, to the best of our knowledge, has been restricted so far to the verification process itself: to model components in an assume-guarantee framework, e.g., by Giannakopoulou et al. (2012), or to model the input-output relation in specific types of programs, in which that relation is sufficient for verifying certain properties (Botinčan & Babić, 2013; Argyros et al., 2016). Another popular direction of research that uses learners and has some similarities to our work is *specification mining* (Alur et al., 2005; Shoham et al., 2008). In both verification and specification mining the focus is on inferring the specification of an API, i.e., deducing the correct usage of it, and in both cases the answer is given in the form of a finite automaton where the alphabet is the API functions' identifiers. As the setting in specification mining is different from ours, the methodology and the results differ as well; for example, the specification-mining algorithms typically infer an overapproximating abstraction of the code under analysis.

Trivially, the language that describes a part of a program or the behaviors that fail an assertion, is not regular in the general case. We therefore bound the length of the traces we consider by a constant, and thereby obtain a finite set of bounded words. The automaton that we learn may accept unbounded words, but our guarantee to the user is limited: any word in $L(\mathcal{A})$, up to the given bound, corresponds to a real trace in the program. We will formalize this concept in Sect. 3. The fact that $\mathcal{A}$ may have loops has both advantages and disadvantages. Consider, for example, the program in Figure 2 (left). Suppose that Σ is the set of functions that are called. With a small bound on the word length we may get the automaton in Figure 2 (right), which among others, accepts the word $g^{120} \cdot f$. The bound is not long enough to exclude this word, rendering the language of the automaton a superset of the language of error. On the other hand, if g had no effect on the reachability of f (i.e., in this case, g did not contain the `exit(0)` statement), then the automaton would capture the language of error precisely, despite the fact that we are only examining bounded traces.

```
void g(int x) { if (x > 100) exit(0); }

void f() { assert(0); }

int main(int argc, char* argv[]) {
  for (int i=0; i < argc; i++) g(i);
  f();
}
```

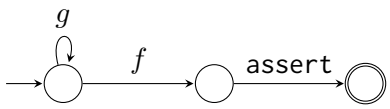


Figure 2: A program and an automaton that we learn from it when using a low bound (< 100) on the word length.

We note that the automaton we generate is conceptually different from a *control-flow graph* (CFG) of a program mapped on a set of interesting events. This is because a CFG is based on the structure of a program, whereas the automaton generated by the learner is based on the actual executions, and in general, cannot be deduced from the CFG. It is also very distinct from just showing the *call graph* of the program, even if we choose function calls as our alphabet: first, it is based on a semantic analysis of the program and hence excludes non existing transitions; second, a call graph has one node per function, and hence cannot make a distinction between paths that go through it. For example, suppose that the

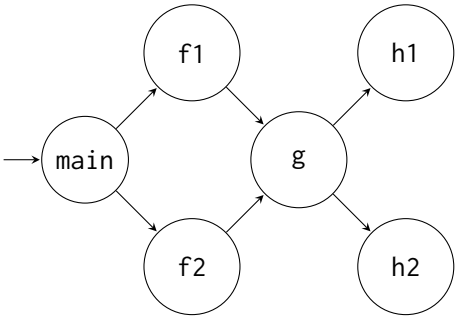


Figure 3: Example call graph.

program can progress (only) in the following two paths:

$$\texttt{main} \rightarrow \texttt{f1} \rightarrow \texttt{g} \rightarrow \texttt{h1}$$

and

$$\texttt{main} \rightarrow \texttt{f2} \rightarrow \texttt{g} \rightarrow \texttt{h2}.$$

A call graph, depicted in Figure 3, illustrates this point: based on the control graph, we cannot distinguish our program from one that also allows the path

$$\texttt{main} \rightarrow \texttt{f1} \rightarrow \texttt{g} \rightarrow \texttt{h2}.$$

A DFA representation, on the other hand, does not make this abstraction.

In the next section we will briefly describe a DFA learning algorithm (specifically, we describe the L^* algorithm, since it is the most basic from the various known DFA-learning algorithms that later improved it). In Sec. 3 we will define the language we learn precisely, and describe our method for answering the queries in Sec. 4. In Sec. 5 we pause the technical description of the method, and describe usage scenarios. We then continue in Sec. 6 by describing a performance optimization, which constitutes the main contribution of this article comparing to the early proceedings version (Chapman et al., 2015). Specifically, we describe a technique we call *lazy learning*, which is targeted at reducing expensive membership queries. In Sec. 8 we describe various aspects of our system and our empirical evaluation of it. More examples can be found on the project's website (*http://www.cprover.org/learning-errors/*, 2015). We conclude with some ideas for future research in Sec. 9.

2. Preliminaries

The ideas and algorithms presented in this paper are based on the combination of learning and software (bounded) model-checking. We now outline the relevant concepts from these two areas. In both cases we focus on the minimum that is necessary for understanding the contribution, as we mostly use them (especially the latter) as 'black-box'.

2.1 Deterministic Finite Automata

We start by revisiting the (well-known) definition of deterministic finite automata (Hopcroft et al., 2000).

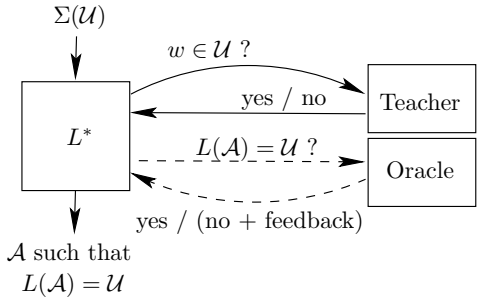


Figure 4: The input and output of L^*, and its interaction with the teacher and oracle.

Definition 1 (Deterministic Finite Automaton)**.** *A deterministic finite automaton (DFA) $\mathcal{A}$ is a 5 tuple $\langle S, init, \Sigma, \delta, F\rangle$, where S is a finite set of states, $init \in S$ is the initial state, Σ is the alphabet, $\delta : S \times \Sigma \to S$ is the transition function, and $F \subseteq S$ is the set of accepting states. A finite string w over Σ is said to be* accepted *by $\mathcal{A}$ if and only if starting at init and following the path in $\mathcal{A}$ induced by w ends in an S state. The set of strings accepted by $\mathcal{A}$ is called $\mathcal{A}$'s* language *and is denoted by $L(\mathcal{A})$.*

The *complement* operation on DFA $\mathcal{A}$ is naturally defined as $\bar{\mathcal{A}} = \langle S, init, \Sigma, \delta, S \backslash F\rangle$, that is, an automaton in which the accepting and non-accepting states are switched. A complement automaton accepts a complement language: $L(\bar{\mathcal{A}}) = \Sigma^* \setminus L(\mathcal{A})$.

Given two DFAs $\mathcal{A}_1, \mathcal{A}_2$ over the same alphabet, there are well-known polynomial techniques (see, e.g., Sipser, 1997) for constructing the *intersection* automaton $\mathcal{A}_\cap$, i.e., $L(\mathcal{A}_\cap) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$, the *difference* automaton $\mathcal{A}_\setminus$, i.e. $L(\mathcal{A}_\setminus) = L(\mathcal{A}_1) \setminus L(A_2)$ and the *union* automaton $\mathcal{A}_\cup$, i.e., $L(\mathcal{A}_\cup) = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$. The *symmetric difference* between $\mathcal{A}_1$ and $\mathcal{A}_2$ is computed as the union of $L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$ and $L(\mathcal{A}_2) \setminus L(\mathcal{A}_1)$.

In the context of this paper, a *language* L over a given alphabet Σ^* is a subset of the set of all finite sequences (words) in Σ^*, denoted as $L \subseteq 2^{\Sigma^*}$. A *language* $L(\mathcal{A})$ of a given automaton $\mathcal{A}$ is the set of all finite words accepted by $\mathcal{A}$ (we also say that $\mathcal{A}$ *recognizes* L). A language L is *regular* if it is a language of some DFA $\mathcal{A}$.

A DFA $\mathcal{A}$ is *minimal* if it has the smallest number of states out of all DFAs over the same alphabet Σ that recognize $L(\mathcal{A})$ (since a deterministic finite automaton has exactly one transition from each state on each letter of Σ, the number of transitions is linear in the number of states).

2.2 The L^* Algorithm

The L^* algorithm, developed by Angluin (1987b), introduces a framework for iterative learning of DFA. Essentially, L^* learns an unknown regular language $\mathcal{U}$ by iteratively constructing a minimal DFA $\mathcal{A}$ such that $L(\mathcal{A}) = \mathcal{U}$. The algorithm includes two types of queries, as can be seen in Figure 4:

- *membership queries* (top arrow), namely whether for a given word $w \in \Sigma^*$, $w \in \mathcal{U}$, and
- *conjecture queries* (third arrow from top), namely whether a given conjectured automaton $\mathcal{A}$ has the property $L(\mathcal{A}) = \mathcal{U}$.

The components that answer the two queries are called 'teacher' and 'oracle', respectively.

L^* begins with the trivial conjecture query corresponding to $\mathcal{A}_0$ such that $L(\mathcal{A}_0) = \emptyset$. If the answer is yes (i.e., $\mathcal{U}$ is the empty language), L^* terminates with $\mathcal{A}_0$ as the answer. Otherwise it expects the teacher to provide a counterexample string σ such that $\sigma \in \mathcal{U} \setminus L(\mathcal{A}_0)$ or $\sigma \in L(\mathcal{A}_0) \setminus \mathcal{U}$. In the first case, we call σ a *positive* feedback, because it should be added to $L(\mathcal{A}_0)$. In the second case, we call σ a *negative* feedback since it should be removed from $L(\mathcal{A}_0)$. Based on the counterexample, L^* initiates a series of membership queries, until it is able to build a new conjecture automaton $\mathcal{A}_1$, that has the property that it is 1) a legal DFA and 2) consistent with the answers to the membership queries answered so far. It then makes a new conjecture query with $\mathcal{A}_1$. This process repeats, building conjectures $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots$ until it converges on a DFA $\mathcal{A}_i$ for some i such that $L(\mathcal{A}_i) = \mathcal{U}$.

Additional details about the inner-workings of L^* are immaterial for the current work. The interested reader is referred to Angluin's original paper (Angluin, 1987b), or one of its improvements (Rivest & Schapire, 1993; Kearns & Vazirani, 1994; Howar, 2012; Isberner et al., 2014).

Complexity: The number of conjecture queries is bounded by n, where n is the number of states of the resulting (minimal) DFA. The number of membership queries is bounded by $O(km^2n^3)$, where k is the size of the alphabet, and m is the length of the longest feedback (counterexample) returned by the oracle during learning.

2.3 Control Flow Graphs

A control flow graph (CFG) is a data structure representing an overapproximation of all paths which may be traversed in a program execution. For every statement in a program the control flow graph provides all potential successor statements which may be executed after the original statement. At runtime for every given program state only a single successor statement is actually executed, depending on loop guards and branching conditions.

2.4 Bounded Model Checking of Software

A key component in our solution is the software analyzer CBMC (Clarke et al., 2004), which we use in the implementation of the teacher and oracle in the learner, as will later be explained. CBMC implements the bounded model checking algorithm as described below.

Let P be a program instrumented with an *assertion* φ (an assertion is a statement in the program parameterized by a propositional formula over the variables of P that are currently in scope; it is only used for debugging and has no impact on the execution of the program). The goal of model checking is to determine whether φ holds in P. The general model checking problem is undecidable (this can be proved by a simple reduction from the Halting problem), hence all model-checking tools work by solving a decidable fragment of it. SAT-based bounded model checking (BMC) is an approach that solves the model-checking problem up to a bound k, for a given program P with an assertion φ (Biere et al., 2003). BMC works by first 'unwinding' the symbolic transition relation up to k steps, hence obtaining a program without loops or recursive calls. CBMC also supports setting a different bound for each loop and recursive call. It then generates a propositional formula that represents symbolically all the possible traces in this program (this is possible since all traces are of length at most k), and adds to the formula the negation of the assertion.

It passes the resulting formula to a SAT solver for determining its satisfiability. If the BMC formula is *unsatisfiable*, the program satisfies φ within the bound k, and is hence safe with respect to this bound (note that errors might still occur on traces longer than k steps). Otherwise, the program contains errors, and each satisfying assignment of the formula corresponds to a counterexample, i.e., an execution that falsifies φ.

BMC is the standard approach to model-checking of software and is widely used both in academia and in the industry, the latter mostly for verifying safety-critical code. While CBMC (as well as other bounded model-checkers) was originally built in the verification community for detecting errors in programs, here we use it for answering queries about programs as part of our learning algorithm.

3. The Language We Learn

Our learning scheme is based on user-defined *events*, which are whatever a user chooses as their atoms for describing the behaviors that lead to an assertion violation. At source level, events are identified by instrumenting the code with a `Learn(id)` instruction at the desired position, where `id` is an identifier of the event. Typical locations for such instrumentation are at the entry to functions and branches, both of which can be done automatically by our tool. Each location obtains its own unique id.

The set of event identifiers constitutes the alphabet Σ of the automaton $\mathcal{A}$ that we construct. A sequence of events is a Σ-*word* that may or may not be in $L(\mathcal{A})$, the language of $\mathcal{A}$. For an instrumented program P, a trace π of P induces a Σ-word by a projection to Σ, which we denote by $\alpha(\pi)$. The language of such a program, denoted $L(P)$, is defined naturally by

$$L(P) \doteq \{\alpha(\pi) \mid \pi \in P\} \ . \tag{1}$$

Recall that our goal is to obtain a representation over the set of events Σ of P's traces that violate a given assertion. Let φ be that assertion, and denote by $\pi \not\models \varphi$ the fact that a given trace π violates φ. We now define

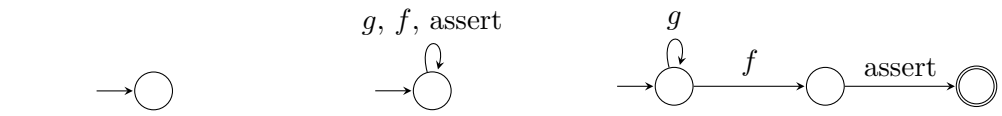
$$\mathit{Fail}(P) \doteq \{\alpha(\pi) \mid \pi \in P \wedge \pi \not\models \varphi\} \ . \tag{2}$$

In general, this set is irregular and incomputable and, even in cases in which it is computable, it is likely to contain too much information to be useful. However, if we bound the loops and recursion in P, this set becomes finite, and hence regular and computable. Let b be such a bound, and let

$$\mathit{Fail}(P, b) \doteq \{\alpha(\pi) \mid \pi \in P \wedge \mathit{Depth}(\pi) \leq b \wedge \pi \not\models \varphi\} \ , \tag{3}$$

where $\mathit{Depth}(\pi)$ denotes the maximal number of loop iterations or recursive calls made along π. Restricting the set of paths this way implicitly restricts the length of the abstract traces that we consider, i.e., $|\alpha(\pi)| \leq b'$, where b' can be computed from P and b. We also allow users to bound the word length $|\alpha(\pi)|$ directly with another value b^{wl}. In Sec. 8 we will describe strategies for obtaining such bounds automatically. Based on these bounds we define

$$\mathit{Fail}(P, b, b^{wl}) \doteq \{\alpha(\pi) \mid \pi \in P \wedge \mathit{Depth}(\pi) \leq b \wedge |\alpha(\pi)| \leq b^{wl} \wedge \pi \not\models \varphi\} \ . \tag{4}$$



(a) Empty conjecture (b) Intermediate conjecture (c) Accepted automaton

Figure 5: A sequence of automata that are conjectured by the learner.

```
void g(int x) { if (x > 100) exit(0); }

void f() { assert(0); }

int main(int argc, char* argv[]) {
  for (int i=0; i < argc; i++) g(i);
  f();
}
```

Figure 6: The program analyzed by the conjectures in Figure 5.

The DFA $\mathcal{A}$, that we learn and present to the user, has the following property for all $\pi \in P$:

$$Depth(\pi) \leq b \wedge |\alpha(\pi)| \leq b^{wl} \wedge \alpha(\pi) \in L(\mathcal{A}) \iff \alpha(\pi) \in Fail(P, b, b^{wl}) \,. \tag{5}$$

Note that this means that given a trace π which has more loop iterations or recursive calls than b, or its projection to Σ is longer than b^{wl}, we give no guarantee regarding whether it is accepted or rejected by $\mathcal{A}$.

4. Answering the Queries

Consider the automaton in Figure 5c, which is learned by our system from the program in Figure 6, when $b = b^{wl} = 4$. This automaton is the second conjecture of the learner. Let us briefly review the steps the learner follows that lead to this conjecture. Initially it has a single state with no transitions, as given in Figure 5a. Then it asks the teacher three single-letter membership queries: whether `f`, `g` and `assert` are in $\mathcal{U}$. The answer is 'no' to all three since, e.g., we cannot reach an assertion failure on a path hitting `f` alone (in fact the first two are trivially false because they do not end with `assert`).

After answering these queries, the learner has a legal DFA that is consistent with the answers to the membership queries, which appears in Figure 5b: an automaton with one non-accepting state. It poses this automaton as a conjecture to the oracle, which answers 'no' and returns $\sigma = \texttt{f} \cdot \texttt{assert}$ as positive feedback, i.e., this word should be added to $L(\mathcal{A})$. Now the learner poses 12 further membership queries and conjectures the automaton in Figure 5c. The oracle answers 'yes', which terminates the algorithm. □

We continue by describing the teacher and the oracle in our case, namely how we answer those queries. The source code of P is instrumented with two functions: Learn(id) at a

```
1:  int word[|w|] = {3, 3, 6, 2, 0};                          ▷ The checked word w
2:  int idx = 0;
3:  function void Learn(int x)                                              ▷ Event
4:    if idx ≥ |w| ∨ word[idx] ≠ x then
5:      assume(false);                          ▷ Block paths incompatible with query
6:    idx = idx + 1;
7:  function void Learn_Assert(bool assertCondition)
8:    if ¬assertCondition then                                          ▷ Assertion fail
9:      if idx = |w| − 1 then assert(false);      ▷ w ∈ L(A). Answer 'true' to query
10:     assume(false);                  ▷ Arrived here at the wrong time: block path.
```

Figure 7: Pseudocode generated for a particular membership query.

location of each Σ-event (recall that ID is the identifier of the event), and Learn_Assert at the location of the assertion that is being investigated. The implementation of these functions depends on whether we are checking a membership or a conjecture query, as we will now show.

4.1 Membership Queries

A membership query is as follows: "given a word w, is there a $\pi \in P$ such that $\alpha(\pi) = w$ and $\pi \not\models \varphi$?" Figure 7 gives the pseudocode that we generate for a membership query — in this case for the word $(3 \cdot 3 \cdot 6 \cdot 2 \cdot 0)$. The letter '0' always symbolizes an assertion failure event, and indeed queries that do not end with '0' are trivially rejected. This code, which is an implementation of the instrumented functions mentioned above, is added to P, and the combined code is then checked with the Bounded Model Checker for software CBMC (Clarke et al., 2004). CBMC supports 'assume(*pred*)' statements, which block any path that does not satisfy the predicate *pred*. In lines 4–5 we use this feature to block paths that are not compatible with w.

Learn_Assert is called when the path arrives at the checked assertion, and declares the membership to be true (i.e., $w \in L(\mathcal{A})$) if the assertion fails exactly at the end of the word. Figure 8 provides an example illustrating how a user program would be instrumented to invoke these API functions.

Optimisations

We bypass a CBMC call and answer 'no' to a membership query if one of the following holds:

- the query does not end with a call to assert;
- the query contains more than one call to assert;
- w is incompatible with the control-flow graph.

The last optimisation is based on the CFG properties described in Sec. 2.3. If there is no path π in the CFG, restricted by the configured loop unwinding, such that $w = \alpha(\pi)$, we skip exploration using membership queries in CBMC, since the CFG models an overapproximation

```
1:  function void main(int input)                          ▷ Entry point
2:    if input ≥ 10 then
3:      F();                                               ▷ Learn(3)
4:    if input ≥ 20 then
5:      F();                                               ▷ Learn(3)
6:      G();                                               ▷ Learn(6)
7:    H();                                                 ▷ Learn(2)
8:    Learn_Assert(input < 20);                            ▷ Learn(0)
9:
10: function F
11:   Learn(3);
12:   // ...
13: function G
14:   Learn(6);
15:   // ...
16: function H
17:   Learn(2);
18:   // ...
```

Figure 8: Sample instrumentation of an input program.

of all possible paths in the program. We call those preliminary p-time checks *pre-checks*. In our experiments they answer 96% of the membership queries. The remaining 4%, which involve a call to CBMC, dominate the overall run-time. We will describe a heuristic that can significantly reduce the number of CBMC calls in Sec. 6.

4.2 Conjecture Queries

A conjecture query is: "given a DFA $\mathcal{A}$, is there a $\pi \in P$ such that

- $\alpha(\pi) \in L(\mathcal{A}) \wedge \pi \models \varphi$, or
- $\alpha(\pi) \notin L(\mathcal{A}) \wedge \pi \not\models \varphi$?"

The two cases correspond to negative and positive feedback to the learner, respectively.

Figure 9 presents the code that we add to P when checking a conjecture query. The candidate $\mathcal{A}$ is given in a two-dimensional array, A (representing the edges in δ), and the accepting states of $\mathcal{A}$ are given in an array *accepting* (both are not shown here). We denote by *path* an array that captures the abstract path, as can be seen in the implementation of learn. Learn_Assert simulates the path accumulated so far (lines 7–8) on $\mathcal{A}$ in order to find the current state. It then aborts if one of the two conditions above holds. In both cases the path *path* serves as the feedback to the learner.

Eliminating Spurious Words

The conjecture-query mechanism described above only applies to those paths that end with Learn_Assert. Other paths should be rejected, and for this we add a 'trap' at the exit

```
1:  function LEARN(int x)                                          ▷ Event
2:      assume(idx < b^wl);
3:      path[++idx] = x;
4:  function LEARN_ASSERT(bool assertCondition)
5:      if ¬assertCondition then LEARN(0);                  ▷ 0 = the 'assert' letter
6:      char state = 0;                                              ▷ Initial state
7:      for (int i = 0; i < idx; ++i) do
8:          state = A[state][path[i]];                          ▷ Finding current state.
9:      if assertCondition ∧ accepting[state] then assert(FALSE);        ▷ neg. feedback
10:     if ¬assertCondition ∧ ¬accepting[state] then assert(FALSE);      ▷ pos. feedback
```

Figure 9: Code added to P for checking conjecture queries.

points of the program. The implementation of this function appears in Figure 11. It ends with negative feedback if the current path is a prefix of a path that a) reaches an accepting state in $\mathcal{A}$ (line 6), and b) was not marked earlier as belonging to $L(\mathcal{A})$ (line 7). The reason for this filtering is that the same abstract path can belong to both a real path $\pi \in P$ and to a path $\pi' \notin P$ that we chose nondeterministically in this function (see line 9). For example, consider the program in Figure 10, which is the same as the one in Figure 2, with the sole difference that it has a branch before f(), allowing it to exit in case a particular input has been received. The final automaton is unchanged. At the exit point a line before the end (the call to `exit`), our instrumentation adds a call to the trap function. Now the path $g \cdot f \cdot 0$ can occur in two different ways: both the path that calls f and the path through the trap function that nondeterministically chooses $f \cdot 0$. Since this word is in the language, we do not want to report it as a negative feedback, and this is the reason for the second condition.

```
void g(int x) { if (x > 100) exit(0); }

void f() { assert(0); }

int main(int argc, char* argv[]) {
  for (int i=0; i < argc; i++) g(i);
  if (argv[0] == '1') exit(0);
  f();
}
```

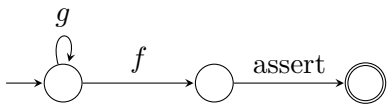


Figure 10: The same program as Figure 2, but with a branch before calling f().

We now show that the above implementation indeed guarantees the properties described in Eqn. (5):

Theorem 4.1. *The implementations of Figure 9 and 11 ensure that for all $\pi \in P$:*

$$|\pi| \leq b \wedge |\alpha(\pi)| \leq b^{wl} \wedge \alpha(\pi) \in L(\mathcal{A}) \iff \alpha(\pi) \in \mathit{Fail}(P, b, b^{wl}) \ .$$

Proof. $\Rightarrow$ We need to show that for all $\pi \in P$

$$|\pi| \leq b \wedge |\alpha(\pi)| \leq b^{wl} \wedge \alpha(\pi) \in L(\mathcal{A}) \Rightarrow |\pi| \leq b \wedge |\alpha(\pi)| \leq b^{wl} \wedge \pi \not\models \varphi$$

```
1: function LEARN_TRAP
2:   char state = 0;
3:   for (; idx < b^wl; ++idx) do
4:     for (int i = 0; i ≤ idx; ++i) do                ▷ Compute current state in A
5:       state = A[state][path[i]];
6:     if accepting[state] then                        ▷ state is an accepting state
7:       if path ∈ L(A) is known then assume(FALSE);   ▷ Block path
8:       assert(FALSE);                                ▷ Negative feedback
9:     path[idx] = non-deterministic element from Σ;
```

Figure 11: LEARN_TRAP is called at P's exit points. It gives negative feedback to conjecture queries in which $\exists w \in L(\mathcal{A})$ such that w does not correspond to any path in P.

The first two conjuncts are trivially true. We prove the third by contradiction. Thus assume that $\pi \models \varphi$. We separate the discussion to two cases:

- $\alpha(\pi)$ ends with a LEARN_ASSERT statement. In the (last) conjecture query π calls that function, which appears in Figure 9. Since $\pi \models \varphi$ the guard in line 5 is false. In line 8 *state*, in the last iteration of the for loop, is accepting, because we know from the premise that $\alpha(\pi) \in L(\mathcal{A})$. This fails the assertion in line 9, and the conjecture is rejected. Contradiction.
- Otherwise, in the (last) conjecture query, π calls the trap function of Figure 11. In line 5 *state*, in the end of the for loop, is accepting, again because we know from the premise that $\alpha(\pi) \in L(\mathcal{A})$. The condition in line 7 is false, and the assert(0) in the following line is reached. The conjecture is rejected. Contradiction.

$\Leftarrow$ We need to show that for all $\pi \in P$

$$|\pi| \leq b \wedge |\alpha(\pi)| \leq b^{wl} \wedge \alpha(\pi) \in L(\mathcal{A}) \quad \Leftarrow \quad |\pi| \leq b \wedge |\alpha(\pi)| \leq b^{wl} \wedge \pi \not\models \varphi$$

Again, the first two conjuncts are trivially true, and we prove the third by contradiction: assume that $\alpha(\pi) \notin L(\mathcal{A})$. Since $\pi \not\models \varphi$, π must end with a call to LEARN_ASSERT. Hence in the (last) conjecture query π calls that function, which appears in Figure 9. By our premise, the state is not accepting. Hence the condition in line 10 is met, and $\alpha(\pi)$ is returned as a positive feedback to L^*, which adds it to $\mathcal{A}$. Contradiction.

□

The trap function has an additional benefit: it brings us close to the following desired property for every word $w \in \Sigma^*$:

$$w \in L(\mathcal{A}) \wedge |w| \leq b^{wl} \implies \exists \pi \in P.\ \alpha(\pi) = w\ . \tag{6}$$

That is, ideally we should exclude from $L(\mathcal{A})$ any word w, $|w| \leq b^{wl}$ that does *not* correspond to a path in P. The reason that this trap function does not guarantee (6) is that it only catches a word $w \in L(\mathcal{A})$ if there is a path $\pi \in P$ to an exit point, such that $\alpha(\pi)$ is a prefix of w. In other cases, the user can check the legality of $w \in L(\mathcal{A})$ either manually or with a membership query.

5. Usage Scenarios for the Learning Framework

We foresee three distinct use cases for our framework. The first is to help understand software errors. Given such an error, we build a DFA that presents an abstraction of the language of error to user-defined events, as defined in Theorem 4.1. The DFA represents the set of error traces in a concise way and is amenable to standard analyses on DFAs, such as the computation of *dominators* and *doomed states and transitions*. These analyses may aid in understanding the *root cause* of errors.

The second use case is *program explanation*, i.e., providing a DFA that describes abstractly the behavior of a program fragment. The motivation for this should be clear to any programmer, because programs are frequently difficult to understand. This is either because of the sheer complexity of the implementation, because of a change in ownership of the code, or because the program was, at least in part, generated automatically. By adding a failing assertion to the exit point of the program, our framework produces a DFA that represents a bounded regular abstraction of the program behavior with respect to important events (as defined by the user or defined automatically).

Finally, our framework can be used for assisting in merging software development branches. In this common scenario, several developers make changes to different branches of the same (version controlled) source code. Often, when the developers attempt to merge their changes back into the parent branch, *automatic merging* is performed by the version control system. This can introduce unexpected behavior. For example, consider the source code in Figure 12, representing an original program, its two branches developed independently, and the result of the automatic merge (this example is based on MergeCase, 2012). Note that the merge operation creates an unexpected behavior, where `funcZ()` calls `funcC()`, despite this not being the intent of either developer.

```
main {
  ...
  funcA();
  funcB();
  ...
}
```

(a) Source

```
main {
  ...
  funcA();
  ...
  funcZ();
}

funcZ() {
  funcB();
}
```

(b) Branch A

```
main {
funcA();
  if (guard)
    funcB();
  else
    funcC();
  ...
}
```

(c) Branch B

```
main {
  ...
  funcA();
  ...
  funcZ();
}

funcZ() {
  if (guard)
    funcB();
  else
    funcC();
  ...
}
```

(d) Merged

Figure 12: The effects of automatic merge in a version control system.

Using our framework, both versions and the merged program are represented as DFAs, and the difference between the merged program and each branch is computed as a difference

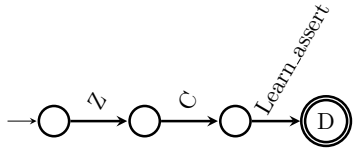


Figure 13: Behavior not in Branch A

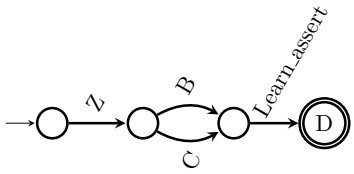


Figure 14: Behavior not in Branch B

between automata (see Sec. 2). Figure 13 and Figure 14 draw attention to the new behavior introduced by the automatic merge.

The changes are easy to see in small examples; in arbitrarily large programs, however, the issues introduced by an automatic merge could be hard to identify. Moreover, this behavior in larger programs might be difficult to understand due to a lack of single ownership over the code. This mechanism can also be applied to merge *conflicts* (i.e., when different versions of code cannot be merged automatically), in order to visually display differences between branches, rather than annotating the repository code directly with conflict markers.

6. Performance Optimization: Lazy Learning

In the previous sections, we saw that the learning algorithm requires to answer membership queries which may amount in our case to a computationally expensive CBMC call. More generally, while it is known that with an $O(1)$-time teacher and oracle, DFA-learners are polynomial in the size of the resulting automaton $\mathcal{A}$, in *all* applications we are aware of in verification the teacher and oracle are exponential, as they involve some type of a formal semantic reasoning about the program. Examples include assume-guarantee reasoning (Chaki & Strichman, 2008), model learning (Fiterau-Brostean et al., 2016) and synthesis (Drachsler-Cohen et al., 2017).

The method we suggest here, *lazy learning*, is relevant to any application of DFA-learners in which answering membership queries is computationally expensive or impossible. We emphasize that we do not intervene in the learning algorithm itself (i.e., the generation of the queries), and this is why this method is orthogonal to the variant of the learning algorithm that is used.

The idea is to *guess* or *approximate* the answer to membership queries whenever they are difficult to answer; then we *detect* wrong guesses during conjecture queries, *cache* the correct answer so as not to repeat this wrong guess, and finally *restart* the process with the updated cache.

What should that guess be? In the case of error explanation, we guess 'no', since most paths do not fail the assertion and hence do not belong to the language that we wish to learn. On the other hand in case of program explanation our default guess is 'yes' because all paths fail the assertion `assert(false)` and hence belong to the language (the only abstract paths that do not belong to the language are those that do not have a corresponding concrete path in the program). In the following we will only focus on the case of error explanation, and hence by default the guess is 'no'.

Figure 15 visualizes our lazy learning algorithm. We recommend the reader to read the caption.

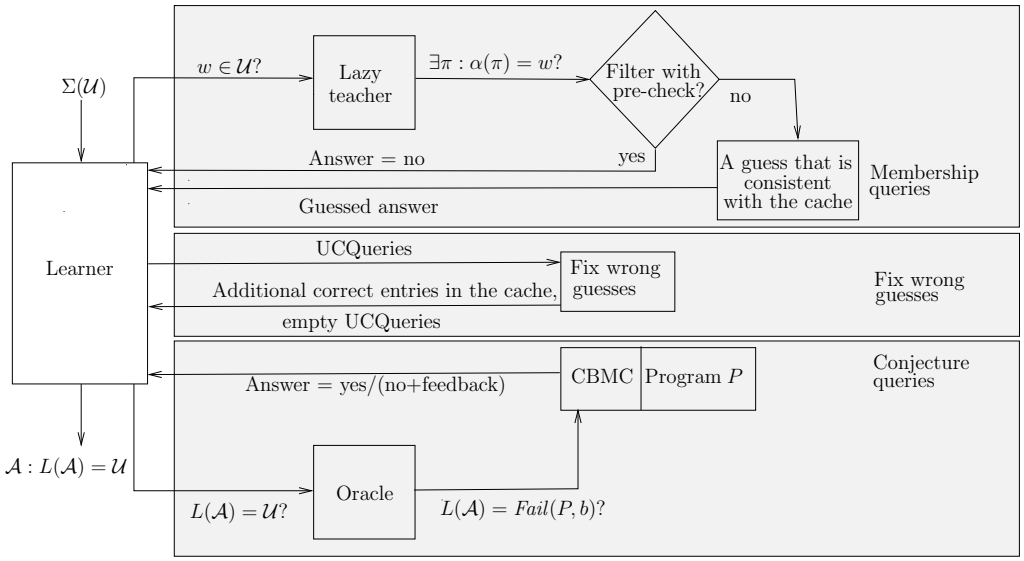


Figure 15: The three main elements of lazy learning: (top) membership queries with polynomial-time pre-checks and guessing. In our case the pre-checks can only answer 'no' to a membership query. The guesses are consistent with the cache of correct answers; (middle) a fixing stage (see Figure 16) of the wrong guesses that ends up with an enlarged cache of correct answers; and (bottom) a conjecture query. UCQueries holds the set of UnConfirmed queries. After a fixing step, the state after the last conjecture is reloaded from memory and the membership queries are answered against an updated cache. Note that the learner is *adaptive*, which means that the queries may be different as a result of the different answers to the membership queries. Conjecture queries (bottom) is invoked only when we know that the conjecture was built with fully correct answers to the membership queries.

A summary of the part of the algorithm that answers the conjecture query, while fixing previous wrong guesses to the membership queries (the middle part in Figure 15), appears in Figure 16. The queries in this algorithm (lines 2 and 7) are not normal membership queries, rather the program is restricted to paths that match one of the membership queries for which the algorithm received possibly incorrect answers. We call such queries *batch membership checks*. Technically this is done by adding automatically an `assume` statement to the C file with which we check the conjecture query, with a disjunction between paths. For example, if we had two membership queries, $(1, 2)$ and $(2, 3)$ where 1,2,3 are letters in Σ, for which we guessed the answer, our tool will now add an assume statement

```
assume((path[0] == 1 && path[1] == 2) ||
       (path[0] == 2 && path[1] == 3))
```

where `path` is an array recording the events in the code corresponding to Σ. CBMC checks the membership property under this assumption (lines 2 and 7), and hence any feedback that it finds, corresponds to one of those words. In our case since we always guess 'no', the counterexample must be a positive feedback.

We update our cache with the correct answer in line 5, remove that word from the `assume` statement in line 6 and re-run CBMC in line 7. We repeat this process as long as

```
1: function FIX( )
2:   feedback = BatchMembership(UCQueries);
3:   if (feedback.empty()) then return ;
4:   do
5:     fix the cache according to the feedback;
6:     UCQueries.remove(feedback);
7:     feedback = BatchMembership(UCQueries);
8:   while (!feedback.empty())
9:   UCQueries.clear();
10:  ls.table=PrevConjectureTable;                ▷ ls is the learner object
```

Figure 16: Lazy learning with local restarts. In the case of our tool (called PV), where the guessed answers are always 'false', entries in the cache are always fixed from 'no' to 'yes'.

```
1: function LearnUptoBound(Program P)
2:   for b ∈ [1 ... b_max] do
3:     for b^wl ∈ [b^wl_min ... b^wl_max] do
4:       A = learn P with b and b^wl;
5:       if A = A_prev and A does not have back edges then return A;
6:       A_prev = A;
```

Figure 17: The autonomous discovery of the appropriate bounds

CBMC finds new positive feedbacks among the unconfirmed queries. Note that the number of CBMC calls is equal to the number of *wrong guesses* plus one. Hence the success of this technique depends on the success rate of the guess.

Once all the wrong answers are fixed, we backtrack the learner to the last conjecture. For this purpose we always save the state of the learner at the last conjecture, in a variable `PreviousConjectureTable`. Since the cache has been updated in the process, we are guaranteed to advance to the next conjecture with all guesses either corrected or confirmed.

7. Usability Optimizations

In this section we describe further features that we implemented in PV in order to improve its usability. We first describe how we determine the bounds automatically, and then how we remove unnecessary edges from the resulting automaton.

7.1 Determining the Bounds

The automatic estimation of suitable values for both the loop bound b and the word length b^{wl} contributes significantly to the usability of our framework. Our strategy for this is illustrated in Figure 17. We let b range between 1 and $b_{\max}$, where $b_{\max}$ is relatively small (4 in our default configuration). This reflects the fact that higher values of b may have a negative impact on performance, and that in practice, low values of b are usually sufficient for triggering the error. As an initial value for b^{wl} ($b^{wl}_{\min}$), we take a conservative estimation of the shortest word possible, according to the length of a shortest loop-free path to the

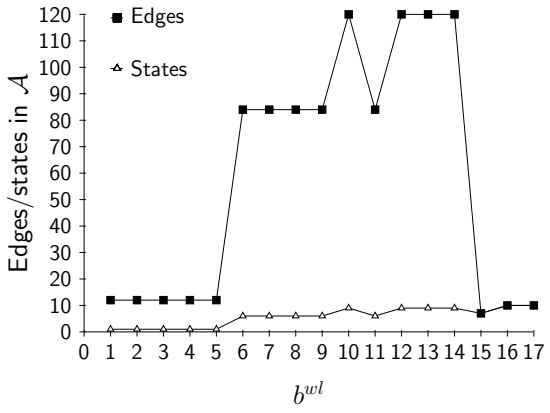


Figure 18: Size of $\mathcal{A}$ (bubble-sort example).

assertion violation on the control-flow graph of P. We increase the value of b^{wl} up to a maximum of $b^{wl}_{\max}$, which is user-defined. The value of $b^{wl}_{\max}$ reflects a user's estimation of how long these words can be before the explanation becomes too long for her to understand.

Recall that the value of b implies a bound on the word length (we denoted it b' in Sec. 3), and hence for a given b, increasing the explicit bound on the word length b^{wl} beyond a certain value is meaningless. In other words, for a given b, the process of increasing b^{wl} converges. Until convergence, the number of states of $\mathcal{A}$ can both increase and decrease as a result of increasing b^{wl} (it can decrease because paths not belonging to the language are caught in the conjecture query, which may lead to a smaller automaton).

Figure 18 demonstrates this fact for one of the benchmarks (bubble sort with $b = 2$). We are not aware of a way to detect convergence in P-time, so in practice we terminate when two conditions hold (see line 5): a) $\mathcal{A}$ has not changed from the previous iteration, and b) $\mathcal{A}$ does not contain edges leaving an accepting state ('back edges'). Recall that a failing assertion aborts execution, and hence no path can continue beyond it. Therefore, the existence of such edges in $\mathcal{A}$ indicates that increasing b, b^{wl} or both should eventually remove them. These conditions can also help the user to decide whether to continue increasing $b^{wl}_{\max}$.

The incremental nature of LearnUptoBound is exploited by our system for improving performance. We maintain a cache of words that have already been proven to be in $\mathcal{U}$, and consult it as the first step of answering membership queries. Negative results from membership queries can only be cached and reused if this result does not depend on the bound. For example, the optimisation mentioned in Sec. 4, by which we reject words that are not compatible with the control-flow graph, does not depend on the bound and hence can be cached and reused. In our experiments caching reduces the number of membership queries sent to CBMC by an average of 32%.

7.2 Post-processing

Our system performs the following post-processing on $\mathcal{A}$ in order to assist the user:

- Marking *dominating edges*: edges that represent events that must occur in order to reach the accepting state. In order to detect these edges, we remove each event in

turn (recall that the same event can label more than one edge), and check whether the accepting state is still reachable from the root.

- Marking *doomed states*: states such that the accepting state is inevitable (Hoenicke et al., 2010).

- Removing the (non-accepting) sink state and its incoming edges: Such a state always exists, because the outgoing edges of the accepting state must transition to it (because, recall, an assertion failure corresponds to aborting the execution). Missing transitions, then, are interpreted as rejection.

7.3 Using PV for Debugging

Our tool PV has an option of storing traces and using them for debugging later. If it is executed with the `--debug` option, the executions that correspond to membership queries are stored and can be used for guiding the debugging process. This option is useful when a particular execution demonstrating the assertion violation is only possible for a specific combination of parameters, such as values of input variables, which are not captured by the learned automaton. In this case, there are many executions corresponding to a particular sequence of important events represented as a trace in the automaton, and only some of them may actually exhibit the assertion violation. Having a debug option allows the user to replay the particular execution that demonstrates the error and to see the values of other relevant parameters in this execution.

8. Experimental Evaluation

Multiple programs with errors and the corresponding DFA that explain those errors can be viewed in our online supplement (*http://www.cprover.org/learning-errors/*, 2015). In that web site the user can check the effect of changing the bounds b and b^{wl}. We encourage the reader to check that page in order to appreciate the value of visual explanations.

In this section we describe our experimental setup and results of implementing and executing the optimisation techniques described in Sec. 6 on a set of benchmarks. Thus the focus here is on performance.

8.1 Setup

We implemented lazy learning in PV, the error/program visualization tool (Chapman et al., 2015). The implementation of the algorithm uses the automata library libalf (Bollig et al., 2010) as the learner (it supports L^*, its improvement RS (Rivest & Schapire, 1993) and several other variants; we experimented with the first two) and the bounded software model checker CBMC 5.4 (Clarke et al., 2004) for answering queries. For each benchmark we measured both the run-time and the number of CBMC calls. We evaluated two configurations:

- baseline – CBMC only, as in (Chapman et al., 2015).

- lazy – the method described in Sect. 6.

p	97.9 %
Runtime Decrease	30.06 %
Saved CBMC Queries	75.31 %

Table 1: Saved CBMC queries and runtime decrease for lazy learning, comparing to a run with standard learning.

8.2 Results

We applied PV for visualizing errors in a set of seven software verification benchmarks that are relatively easy as verification targets for CBMC. They were drawn from three sources: the Competition on Software Verification (Beyer, 2015), the Software-artifact Infrastructure Repository[2], and a 'DockingApproach' program: a program describing the behavior of a space shuttle as it docks with the International Space Station (an open-source version of the NASA system *Docking_Approach*). Each of these programs contain a single instrumented assertion. These are the same benchmarks used in (Chapman et al., 2015), other than the addition of the JQ benchmark, which is a lightweight command-line JSON parser. We use JQ as a motivating industrial example which, as we explain below, benefits greatly from our lazy algorithms. Each of these benchmarks is executed at different loop unwinding and word length settings, resulting in 83 data points. An Excel sheet with detailed results can be downloaded from the project's website (*http://www.cprover.org/learning-errors/*, 2015). Tab. 1 summarizes the average runtime decrease as well as the percentage of saved CBMC queries when using lazy learning. We denote by p the ratio of correct guesses.

As we mentioned in Sec. 6, in our application domain, the strategy of guessing "no" performs best for learning the language of error, and "yes" for program explanation, if there is no information on the query. As we can see from Tab. 1, the probability of an answer being correct with this strategy is more than 0.9.

In our experiments, an average of 75 % of all CBMC queries can be avoided using lazy learning. This results in an average runtime decrease of 30 %. This suggests that, while lazy learning invokes CBMC fewer times, the batch CBMC queries create larger formulas on average and thus take more time to solve. Hence the runtime decrease does not exactly match the reduction in CBMC invocations.

Tab. 2 shows the number of saved CBMC queries and the runtime improvement for each of the examined benchmarks. Note that lazy learning leads to a runtime *increase* in the benchmark *sll_to_dll_rev_false*, owing to the fact that the CBMC queries only amount to a few seconds for this benchmark, and that the overhead of constructing more complicated batch queries negates any potential benefits. While this represents an outlier in our benchmark set, it illustrates that lazy learning is not always beneficial.

In addition to those experiments, Appendix A describes in detail an industrial use case that we experimented with, based on the JQ JSON parser. It describes not only performance, but also the process of using PV that helped us detect a previously unknown bug in this widely used application.

2. http://sir.unl.edu

	p (%)	Runtime Decrease (%)	Saved CBMC Queries (%)
DockingApproachExample_Ext	100.0	81.54	85.45
ifstofunctions_defroster	98.5	11.58	44.50
merge_sort_false	95.5	23.61	75.83
sll_to_dll_rev_false	99.0	−11.92	89.58
tcas_auto_instrumented	100.0	5.32	92.10
JQ	96.5	24.88	63.77
schedule	95.7	75.44	75.97

Table 2: Experimental results per benchmark

9. Conclusions and Future Work

Our definition of $\mathit{Fail}(P)$ in (2) captures the 'language of error', but this language is, in the general case, not computable. We have presented a method for automatically learning a DFA, $\mathcal{A}$, that captures a well-defined subset of this language (see Theorem 4.1), for the purpose of assisting the user in understanding the cause of the error. More generally, the same technique can be used for visualising the behavior of a program or parts thereof, and for assisting in merging code.

There are multiple venues for future research. On the application side, perhaps a similar method can be used for visually comparing different versions of a program (by presenting an automaton that captures the behavior in the symmetric difference between them).

On the more theoretical side, an interesting future work is to adapt the framework to learn ω-regular languages, represented by Büchi automata (see Farzan, Chen, Clarke, Tsay, & Wang, 2008; Angluin & Fisman, 2016 for extensions of DFA-learners to ω-regular languages). This extension would enable the learning of behaviors that violate the liveness properties of non-terminating programs. Another future direction is learning non-regular languages, as it will enable the learning of richer abstract representations of the language of error for a given program. Context-free grammars are of particular interest because of the natural connection between context-free grammars and the syntax of programming languages; some subclasses of context-free grammars have been shown to be learnable, such as k-bounded context free grammars (Angluin, 1987a), providing us with the possibility of harnessing these algorithms in our framework. The class of context-free grammars without restrictions is not believed to be learnable (Angluin & Kharitonov, 1995).

Finally, since one of the main goals of our framework is to present the language of error (or interesting behavior) in a compact, easy to analyze and understandable way, then clearly small automata are preferable. Even for a given alphabet Σ, we believe it should be possible to reduce the size of the learned DFA $\mathcal{A}$, based on the observation that we *do not care* whether a word w such that $\forall \pi \in P. |\pi| \leq b \wedge |\alpha(\pi)| \leq b^{wl} \Rightarrow \alpha(\pi) \neq w$ is accepted or rejected by the automaton. Adding a 'don't care' value to the learning scheme requires a learning mechanism that can recognize three-valued answers (see Leucker & Neider, 2012 for a learning algorithm with inconclusive answers).

```
> jq -L mod -n 'import␣"module0"␣as␣check;␣check::check'
> true

> jq -L ./mod -n 'import␣"module0"␣as␣check;␣check::check'
> jq: error: module not found: module0
>
> jq: 1 compile error
```

Figure 19: Command line example triggering the JQ bug.

Appendix A. A Detailed Case Study: JQ

The JQ[3] project is a lightweight command-line JSON processor. JQ is one of the most popular C projects on GitHub with over 5000 stars, and is widely used in industrial and productive settings. It is comprised of over 75k lines of C code and includes a multitude of complex heap data structures. The latter makes JQ a very challenging target for static analysis, since even simple operations such as string concatenation are implemented using sizeable heap data structures. The goal of our case study is to illustrate that for a large industrial program such as JQ, the improvements to the error learning algorithm can reduce the analysis time significantly.

In order to demonstrate the increased learning speed, we picked a known and—at the time of our first experiments—open bug from the JQ bug tracker page[4]. We give a brief bug description in the following paragraph and then describe how we used the lazy error learning framework to create a visual description of the bug which allowed us to understand and fix it.[5]

The bug occurs during the module-loading phase, where JQ is linking external modules into the current JSON context. JQ users can provide library search paths in which JQ will look for these additional modules. Users can specify absolute as well as relative paths for this purpose. However, depending on the input syntax, module lookup using relative paths fails. In particular, using, e.g., `./mod` instead of `mod` will trigger this bug. Figure 19 shows both a successful and a failing invocation of JQ.

In order to isolate and fix this bug, we first used a traditional GDB debugging session to identify the function `build_lib_search_chain(...)`, which is responsible for creating the fully expanded list of library search paths from the user input. "Expanded" in this context means absolute file system paths with all JQ environment variables replaced. Our debugging session further showed that this process works for some invocations of `build_lib_search_chain(...)`, even with the `./` path syntax, but fails for others. Our assumption was that this discrepancy must be linked to the values of the three arguments passed into the function. We created user-defined learning events monitoring the status of these arguments and analysed the program using PV. All generated automata are published on the project's website (*http://www.cprover.org/learning-errors/*, 2015).

3. https://github.com/stedolan/jq
4. https://github.com/stedolan/jq/issues/817
5. By the time we fixed the bug a formal patch was already committed.

The resulting automaton highlighted which events and input parameter manipulations were necessary to reach the bug in `build_lib_search_chain(...)`. Using this information we successfully implemented a patch fixing this issue. For this particular benchmark, the lazy learning algorithm described in Sec. 6 reduced the runtime of PV from over 59 minutes down to 28 minutes at user unwind limit 4 and maximum word length 12. On average, a runtime decrease by 25% was achieved across all unwind limits and word lengths. More detailed results can be found in (*http://www.cprover.org/learning-errors/*, 2015).

References

Alur, R., Cerný, P., Madhusudan, P., & Nam, W. (2005). Synthesis of interface specifications for Java classes. In *Principles of Programming Languages (POPL)* (pp. 98–109). ACM.

Angluin, D. (1987a). *Learning k-bounded context-free grammars* (Tech. Rep.). Dept. of Computer Science, Yale University.

Angluin, D. (1987b). Learning regular sets from queries and counterexamples. *Information and Computation*, *75*(2), 87-106.

Angluin, D., & Fisman, D. (2016). Learning regular omega languages. *Theor. Comput. Sci.*, *650*, 57–72.

Angluin, D., & Kharitonov, M. (1995). When won't membership queries help? *J. Comput. Syst. Sci.*, *50*(2), 336–355.

Argyros, G., Stais, I., Kiayias, A., & Keromytis, A. D. (2016). Back in black: Towards formal, black box analysis of sanitizers and filters. In *IEEE Symposium on Security and Privacy (S&P)* (pp. 91–109). IEEE Computer Society.

Ball, T., Naik, M., & Rajamani, S. K. (2003). From symptom to cause: localizing errors in counterexample traces. In *Principles of Programming Languages (POPL)* (pp. 97–105).

Basu, S., Saha, D., Lin, Y., & Smolka, S. A. (2003). Generation of all counter-examples for push-down systems. In *Formal Techniques for Networked and Distributed Systems (FORTE)* (pp. 79–94). Springer.

Beer, I., Ben-David, S., Chockler, H., Orni, A., & Trefler, R. J. (2012). Explaining counterexamples using causality. *Formal Methods in System Design*, *40*(1), 20–40.

Beyer, D. (2015). Software verification and verifiable witnesses – report on SV-COMP 2015. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (pp. 401–416). Springer.

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., & Zhu, Y. (2003). Bounded model checking. *Advances in Computers*, *58*, 117–148.

Bollig, B., Katoen, J., Kern, C., Leucker, M., Neider, D., & Piegdon, D. R. (2010). `libalf`: The automata learning framework. In *Computer Aided Verification (CAV)* (pp. 360–364). Springer.

Botinčan, M., & Babić, D. (2013). Sigma*: Symbolic learning of input-output specifications. In *Principles of Programming Languages (POPL)* (pp. 443–456). ACM.

Chaki, S., & Strichman, O. (2008). Three optimizations for assume–guarantee reasoning with L*. *Formal Methods in System Design*, *32*(3), 267–284.

Chapman, M., Chockler, H., Kesseli, P., Kroening, D., Strichman, O., & Tautschnig, M. (2015). Learning the language of error. In *Automated technology for verification and*

analysis (ATVA) (Vol. 9364, pp. 114–130). Springer.
Clarke, E., Kroening, D., & Lerda, F. (2004). A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Vol. 2988, pp. 168–176). Springer.
Copty, F., Irron, A., Weissberg, O., Kropp, N. P., & Kamhi, G. (2003). Efficient debugging in a formal verification environment. *STTT*, *4*(3), 335–348.
Drachsler-Cohen, D., Shoham, S., & Yahav, E. (2017). Synthesis with abstract examples. In *Computer Aided Verification (CAV), part I* (Vol. 10426, pp. 254–278). Springer.
Farzan, A., Chen, Y., Clarke, E. M., Tsay, Y., & Wang, B. (2008). Extending automated compositional verification to the full class of omega-regular languages. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (pp. 2–17). Springer.
Fiterau-Brostean, P., Janssen, R., & Vaandrager, F. (2016). Combining model learning and model checking to analyze TCP implementations. In *Computer Aided Verification (CAV)* (pp. 454–471). Springer.
Giannakopoulou, D., Rakamaric, Z., & Raman, V. (2012). Symbolic learning of component interfaces. In *Static Analysis – 19th International Symposium (SAS)* (Vol. 7460, pp. 248–264). Springer.
Groce, A., Chaki, S., Kroening, D., & Strichman, O. (2006). Error explanation with distance metrics. *STTT*, *8*(3), 229-247.
Hoenicke, J., Leino, K., Podelski, A., Schäf, M., & Wies, T. (2010). Doomed program points. *Formal Methods in System Design*, *37*(2–3), 171–199.
Hopcroft, J., Motwani, R., & Ullman, J. (2000). *Introduction to automata theory, languages, and computation (2nd edition)*. Addison-Wesley.
Howar, F. (2012). *Active learning of interface programs* (Unpublished doctoral dissertation). TU Dortmund University.
(2015). (http://www.cprover.org/learning-errors/)
Isberner, M., Howar, F., & Steffen, B. (2014). The TTT algorithm: A redundancy-free approach to active automata learning. In B. Bonakdarpour & S. A. Smolka (Eds.), *Proceedings of RV* (Vol. 8734, pp. 307–322). Springer.
Kearns, M., & Vazirani, U. (1994). *An introduction to computational learning theory*. MIT Press.
Lee, T. (2012). *The problem of automatic code merging*. (http://www.personal.psu.edu/txl20/blogs/tks_tech_notes/2012/03/the-problem-of-automatic-code-merging.html)
Leucker, M., & Neider, D. (2012). Learning minimal deterministic automata from inexperienced teachers. In *Proc. of 5th ISoLA* (pp. 524–538). Springer.
Rivest, R. L., & Schapire, R. E. (1993). Inference of finite automata using homing sequences. In S. J. Hanson, W. Remmele, & R. L. Rivest (Eds.), *Machine learning: From theory to applications – cooperative research at Siemens and MIT* (Vol. 661, pp. 51–73). Springer.
Shoham, S., Yahav, E., Fink, S. J., & Pistoia, M. (2008). Static specification mining using automata-based abstractions. *IEEE Trans. Software Eng.*, *34*(5), 651–666.
Sipser, M. (1997). *Introduction to the theory of computation*. PWS Publishing Company.