Learning the Language of Error

Martin Chapman[1], Hana Chockler[1](✉), Pascal Kesseli[2], Daniel Kroening[2], Ofer Strichman[3], and Michael Tautschnig[4]

[1] Department of Informatics, King's College London, London, UK
{martin.chapman,hana.chockler}@kcl.ac.uk
[2] Department of Computer Science, University of Oxford, Oxford, UK
{pascal.kesseli,kroening}@cs.ox.ac.uk
[3] Information Systems Engineering, Technion, Haifa, Israel
ofers@ie.technion.ac.il
[4] EECS, Queen Mary University of London, London, UK
mt@eecs.qmul.ac.uk

Abstract. We propose to harness Angluin's L^* algorithm for learning a deterministic finite automaton 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. This can be used, for example, for visually comparing different versions of a program, by presenting an automaton for the behavior in the symmetric difference between them, or for assisting in merging several development branches. We present initial experiments that demonstrate the power of an abstract visual representation of errors and of program segments.

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 [16] and analysing causality [8], but these focus on a single counterexample. The analysis of *multiple* counterexamples has been suggested in the hardware domain by Copty et al. [13], who propose to compute all counterexamples and present those states that occur in all of them to the user. Multiple counterexample analysis has also been suggested in the context of a push-down automaton (PDA) (representing software) and a deterministic finite automaton (DFA) (representing a

This work was supported in part by the Google Faculty Research Award 2014.

B. Finkbeiner et al. (Eds.): ATVA 2015, LNCS 9364, pp. 114–130, 2015.
DOI: 10.1007/978-3-319-24953-7_9

negated property) by Basu et al. [7], who describe the generation of all loop-free counterexamples of a certain class, and the presentation of them to the user in a tree-like structure. In software, another notable example is the model checker MS-SLAM, which reports multiple counterexamples if they are believed to relate to different causes [6], and each example is 'localized' by comparing it to a trace that does not violate the property.

We believe that developers can benefit from seeing the multiple ways in which a given assertion can fail, and that raw counterexamples quickly become unhelpful. In this article we suggest that a user should be presented with 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, e.g., the user can give some subset of the function calls in a program. 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 subprogram to be explained. Fig. 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.

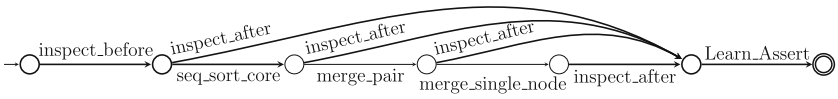


Fig. 1. An abstract description of a merge-sort program, where the letters are the function calls.

Our method is based on Angluin's L^*-learning algorithm [3]. L^* is a framework for learning 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 L^* via an interface called the 'teacher'. L^* 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 Sect. 2 for a more in-detail description).

The use of L^* 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., [15], or to model the input-output relation in specific types of programs, in which that relation is sufficient for verifying certain properties [11].

[1] Source code for all the programs mentioned in this article is available online from [1].

Trivially, the language that describes a part of a program, or the behaviors that fail an assertion, is neither finite nor 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 finite 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 Fig. 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 Fig. 2 (right), which among others, accepts the word $g^{120} \cdot f$. The bound is not long enough to exclude this word. On the other hand, if g had no effect on the reachability of f, 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(argc);
  f();
}
```

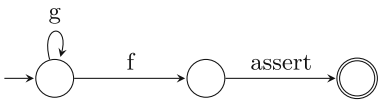


Fig. 2. A program and an automaton that we learn from it when using a low bound (< 100) on the word length.

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 L^* is based on the actual executions, and in general, cannot be deduced from the CFG.

In the next sections, we briefly describe the L^* algorithm and define the language we learn precisely. We follow with a detailed description of our method, which is based on the L^* algorithm, using it mostly as a black box. We describe various aspects of our system and our empirical evaluation of it in Sect. 6, and conclude with some ideas for future research in Sect. 7. More examples can be found on the project's website [1].

2 Preliminaries – the L^* Algorithm

We start by revisiting the well-known definition of deterministic finite automata (cf. [18]).

Definition 1 (Determinisic 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, $\delta : S \times \Sigma \rightarrow S$ is the transition function, and $F \subseteq S$ is the set of accepting states. We denote by $L(\mathcal{A})$ the language accepted by the automaton $\mathcal{A}$.*

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

The *intersection* operation $\mathcal{A}_1 \cap \mathcal{A}_2$, where $\mathcal{A}_1 = \langle S_1, init_1, \Sigma, \delta_1, F_1\rangle$ and $\mathcal{A}_2 = \langle S_2, init_2, \Sigma, \delta_2, F_2\rangle$, assuming the same alphabet, results in the automaton $\mathcal{A} = \langle S, init, \Sigma, \delta, F\rangle$, with the set of states S being a cross-product $S_1 \times S_2$ of the sets of states of $\mathcal{A}_1$ and $\mathcal{A}_2$, and the transition relation following the same letter on both parts of the state-pair. The initial state *init* is defined as $init_1 \times init_2$, and the set of accepting states F is $F_1 \times F_2$ (that is, both $\mathcal{A}_1$ and $\mathcal{A}_2$ need to accept in order for $\mathcal{A}$ to accept). The language $L(\mathcal{A})$ is the intersection of languages $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

The *difference* operation between two languages of DFA, $L(\mathcal{A}_1) \setminus L(\mathcal{A}_2)$, is computed as the language of $\mathcal{A}_1 \cap \bar{\mathcal{A}}_2$, that is, an intersection of $\mathcal{A}_1$ with the complement of $\mathcal{A}_2$. 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 understanding the behavior of programs, we find that it is easier to analyze both sides of the difference separately, hence we do not produce the automaton for the union of differences (though it can easily be done).

The L^* algorithm, developed by Angluin [3], 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: *membership queries* and conjecture queries. Angluin's original description of the entities that answer the two queries uses the terms 'teacher' and 'oracle'; for simplicity we unify them here under the name 'teacher'. Figure 3 describes the interaction of L^* with the teacher. L^* learns $\mathcal{U}$, by querying the teacher with two types of questions:

- *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 $L(\mathcal{A}) = \mathcal{U}$. If the answer is yes, L^* terminates with $\mathcal{A}$ as the answer. Otherwise it expects the teacher to provide a counterexample string σ such that $\sigma \in \mathcal{U} \setminus L(\mathcal{A})$ or $\sigma \in L(\mathcal{A}) \setminus \mathcal{U}$. In the first case, we call σ *positive* feedback, because it should be added to $L(\mathcal{A})$. In the second case, we call σ *negative* feedback since it should be removed from $L(\mathcal{A})$. Based on the counterexample, L^* initiates a new series of queries, until it converges on a DFA $\mathcal{A}$ such that $L(\mathcal{A}) = \mathcal{U}$.

L^* maintains a table (called an 'observation table') that records transitions and states, and is used to construct the resulting automaton $\mathcal{A} = \langle S, init, \Sigma, \delta, F\rangle$.

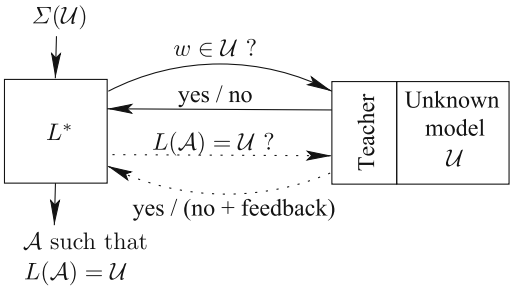


Fig. 3. The input and output of L^*, and its interaction with the teacher.

The states are defined by the prefixes they accept, and for each state s and each letter σ, the table defines whether $\delta(s, \sigma) \in F$. Let S' be a set of states currently in this table. A table is said to be *closed* if $\forall s \in S'.\ \forall \sigma \in \Sigma.\ \delta(s, \sigma) \in S'$. In other words, the table is closed when it represents a complete transition function. To close its table, L^* asks the teacher multiple membership queries. By construction, once the table is closed it represents a DFA. L^* presents this DFA as a conjecture query to the teacher. If the answer to the query is 'no', it analyzes the counterexample σ and adds states and transitions to accommodate it, which makes the table 'open' again. This leads to additional membership queries, and the process continues.

The underlying principle ensuring the convergence of the L^* algorithm is the Myhill-Nerode theorem [18], which provides a sufficient and necessary condition for a language to be regular. Since we assume that $\mathcal{U}$ is regular, L^* uses the Myhill-Nerode theorem to compute the equivalence classes of $\mathcal{U}$, which are mapped to the states of the final DFA. The number of queries is bounded by $O(km^2n^3)$, where k is the size of the alphabet, n is the number of states of the resulting (minimal) DFA, and m is the length of the longest feedback (counterexample).[2]

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

[2] This is a simplified upper bound of the complexity of the L^* algorithm.

(further denoted as $\pi \in P$) induces a Σ-word, 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 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 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

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

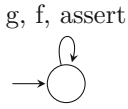
where $|\pi|$ denotes the maximal number of loop iterations or recursive calls made along π. Restricting the set of paths this way implicitly restricts the length of the abstract traces that we consider, i.e., $|\alpha(\pi)| \leq b'$, where b' can be computed from P and b. We also allow users to bound the word length $|\alpha(\pi)|$ directly with another value b^{wl}. In Sect. 6 we will describe strategies for obtaining such bounds automatically. Based on these bounds we define

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

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

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

4 L^* and the Queries



Consider the automaton in Fig. 2, which is learned by our system from the program on the left of the same figure, when $b = b^{wl} = 4$. This automaton is the second conjecture of L^*. Let us briefly review the steps L^* follows that lead to this conjecture. Initially it has a single state with no transitions. 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, L^* has a closed table corresponding to the automaton on the right: an automaton with one non-accepting state. It poses this automaton as a conjecture to the teacher, 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 L^* poses 12 more membership queries and conjectures the automaton in Fig. 2. The teacher answers 'yes', which terminates the algorithm. □

We continue by describing the teacher in our case, namely how we answer those queries. The source code of P is instrumented with two functions: LEARN(ID) at a 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 $\pi \not\models \varphi$?" Fig. 4 gives sample code 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 [12]. 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.

```
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 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.
```

Fig. 4. Sample (pseudo) code generated for a particular membership query.

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.

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 L^*, respectively.

Figure 5 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, and the accepting states of $\mathcal{A}$ are given in an array *accepting* (both are not shown here). *path* is 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 6–7) 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 L^*.

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

Fig. 5. Code added to P for checking conjecture queries.

Eliminating Spurious Words. The conjecture-query mechanism described above only applies to paths ending with LEARN_ASSERT. Other paths should be rejected, and for this we add a 'trap' at the exit points of the program. The implementation of this function appears in Fig. 6. 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 (word) can belong to both a real abstract path $p \in P$ and to a path $p' \notin P$ that we chose nondeterministically in this function (see line 9). For example, a path $p = 1 \cdot 1 \cdot 2 \cdot 0$ can exist in P (recall that the '0' at the end of this path means that it violates the assertion), but there is another path p' that does not go via any of these locations and reaches LEARN_TRAP, which nondeterministically chooses this path.

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

Theorem 1. *The implementations of Figs. 5 and 6 ensure that for all $\pi \in P$:*

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

```
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;
```

Fig. 6. 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.

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

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

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 Fig. 5. Since $\pi \models \varphi$ the guard in line 4 is false. In line 7 *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 8, and the conjecture is rejected. Contradiction.
- Otherwise, in the (last) conjecture query, π calls the trap function of Fig. 6. 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 Fig. 5. By our premise, the state is not accepting. Hence the condition in line 9 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 $w \in \Sigma^*$:

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

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.

Optimisation. We can bypass a CBMC call in the following case: consider an automaton $\mathcal{A}_{cfg}$ in which the states and transitions are identical to those of the control-flow graph (CFG) of P, and every state is accepting. Since the elements of Σ correspond to locations in the program we can associate them with nodes in the CFG. Hence, we can define $L_\Sigma(\mathcal{A}_{cfg})$, the language of $\mathcal{A}_{cfg}$ projected to Σ. Then if $L(\mathcal{A}) \not\subseteq L_\Sigma(\mathcal{A}_{cfg})$, return 'no', with an element of $L(\mathcal{A}) \setminus L_\Sigma(\mathcal{A}_{cfg})$ as the negative feedback.

5 Usage Scenarios for the Learning Framework

The framework presented in the paper is constructed in order to understand the language of software errors. That is, given a program with errors reported from the verification process, applying the learning algorithm results in a DFA that presents an abstraction of the bounded language of error traces of the program to important events, as defined in Theorem 1. The automaton represents the set of error traces in a concise and compact way and is amenable to standard analyses on DFAs, such as the computation of *dominators* and *doomed states and transitions*. These analyses aid in understanding the *root cause* of errors.

Beyond the language of software errors, our framework can be applied, without changes, to the task of *program explanation*. As illustrated by the Docking software example in Sect. 6.2, modern-day programs are often very 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, we extend our framework to assist in merging several 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 Fig. 7, representing an original program, its two branches developed independently, and the result of a widely used automatic merge (this example is taken from [20]). Note that the merge operation creates an unexpected behavior, where `functionZ()` calls `functionC()`, despite this not being either developer's intent.

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 between automata (see Sect. 2). Figures 8 and 9 draw attention to the new behavior introduced by the automatic merge.

```
main {
  ...
  functionA();
  functionB();
  ...}
```

(a) Source

```
main {
  ...
  functionA();
  ...
  functionZ();}

functionZ() {
  functionB();}
```

(b) Branch A

```
main {
  ...
  functionA();
  functionB();
  functionC();
  ... }
```

(c) Branch B

```
main {
  ...
  functionA();
  ...
  functionZ();
}

functionZ() {
  functionB();
  functionC();
  ...}
```

(d) Merged

Fig. 7. The effects of automatic merge in a version control system.

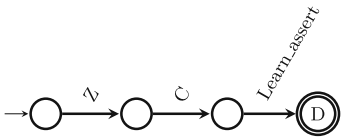


Fig. 8. Behavior not in Branch A

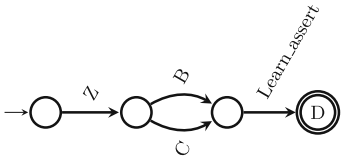


Fig. 9. Behavior not in Branch B

The changes are easy to see in small examples; 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 System Description and Empirical Evaluation

In this section, we discuss the possible optimisations of the algorithm and present experimental results of executing our framework on standard benchmarks.

6.1 Optimisations

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 Fig. 10. 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, with CBMC low values of b are sufficient for triggering the error. As an initial value for b^{wl} ($b^{wl}_{\min}$), we take a

```
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;
```

Fig. 10. The autonomous discovery of the appropriate bounds.

conservative estimation of the shortest word possible, according to a light-weight analysis of 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 an estimation of how long these words can be before the explanation becomes unintelligible.

Recall that the value of b implies a bound on the word length (we denoted it b' in Sect. 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 11 demonstrates this fact for one of the benchmarks (bubble sort with $b = 2$). We are not aware of a way to detect convergence in Ptime, 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.

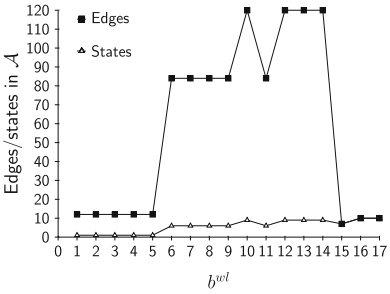


Fig. 11. Size of $\mathcal{A}$ (bubble sort example).

Incrementality. 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 Sect. 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 %.

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 [17].
- 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.

6.2 Implementation and Evaluation

Our implementation of the learning framework is based on the automata library libalf [10] as the L^* component and the bounded software model-checker for C CBMC [12] as the 'teacher' component and includes the optimisations described in Sect. 6.1. The modular implementation allows to replace both the L^* and the teacher component with other alternatives, which we discuss in Sect. 7.

We applied our framework to learn the language of error associated with a set of software verification benchmarks (that are relatively easy as verification targets for CBMC) drawn from three sources: the Competition on Software Verification [9], the Software-artifact Infrastructure Repository[3], and a 'docking' 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. Whilst learning, we record our estimated $b^{wl}_{\min}$, and when LearnUptoBound terminates we record the values of b^{wl}, the number of iterations, b, the total CPU time in seconds, the number of states and edges in $\mathcal{A}$, the number of calls to CBMC as a percentage of the total membership queries, and the total number of conjecture queries. All experiments were conducted on a computer with a 3.2 GHz quad-core processor and 6 GB of DDR3 RAM. The results are summarized in Table 1. We also tested a strategy by which we do not return at line 5 of Fig. 10 (recall that the condition there does not guarantee convergence), and rather only print $\mathcal{A}$. The multiple entries of b^{wl} and b for the same example in Table 1 reflect this.

Next, we present several examples of $\mathcal{A}$ from this benchmark set. Bold edges in our figures indicate dominating events, e.g., the function `inspect` in Fig. 12 is marked as dominating because a path to the error *must* call it. Doomed states are labelled with 'D'. (In this and later examples all states have paths to the non-accepting sink-state which we remove in post-processing, as explained above. Hence only the accepting state is marked doomed).

Figures 12 and 13 give $\mathcal{A}$ in the bubble sort example with bounds $(b, b^{wl}) = (2, 12)$ and $(b, b^{wl}) = (3, 15)$ respectively. The example constructs a linked list

[3] http://sir.unl.edu/.

Table 1. Experimental results. $b^{wl}_{\min}$ is our initial bound estimation. b^{wl}, *It.*, b and Time pertain to the process in LearnUptoBound, which produces $\mathcal{A}$. We give the number of states and edges of $\mathcal{A}$. We also list the percentage of membership queries made to CBMC, and the total number of conjecture queries.

Target	$b^{wl}_{\min}$	b^{wl}	*It.*	b	Time [sec]	States	Edges	C BMC Queries memb.	C BMC Queries conj.
tcas	3	17	14	1	7.76	25	28	0.51 %	34
bubble_sort	8	16	8	2	1.61	10	10	0.17 %	66
bubble_sort	8	19	19	3	4.24	19	21	0.13 %	96
merge_sort	4	8	4	1	0.13	4	3	0.92 %	12
merge_sort	4	12	12	2	0.74	7	9	2.90 %	40
sll_to_dll_rev	8	28	20	1	2.83	14	13	0.18 %	39
sll_to_dll_rev	8	28	40	2	7.28	17	19	0.15 %	78
defroster	25	29	4	1	32.92	14	18	0.01 %	26
docking	5	8	3	1	0.49	7	6	0.86 %	9
docking	5	8	6	2	0.72	7	6	0.86 %	18
docking	5	11	12	2	1.65	11	11	1.04 %	27

of non-deterministic size and contains a bug where the root node linkage is not initialized correctly. The bug can thus occur after an arbitrary amount of node insertions by the `gl_insert` operation. Using Fig. 12 we can conjecture that the bug either occurs after one or two insertions. In Fig. 13 L^* then correctly conjectures a loop and represents the whole nature of the bug with $b^{wl} = 15$. Whether L^* is able to conjecture a loop is dependent on multiple factors, but ultimately linked to the word length bound b^{wl}. Our membership query oracle will reject any word which is longer than b^{wl}. However, this limitation does not apply in the conjecture oracle. Since L^* at $b^{wl} = 15$ does not pose any membership queries exceeding this limit, the result in Fig. 13 ensues. The membership query list posed by L^* is dependent on the counter-examples provided by CBMC, which vary for different values of b and b^{wl}.

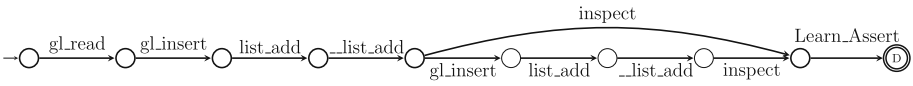


Fig. 12. Automaton produced for the 'bubble_sort' example. $b^{wl} = 12$. $b = 2$.

Figure 14 shows the 'docking' benchmark, with $b = 4, b^{wl} = 15$. This automaton is an example of a *program explanation* usage scenario. The C source code of the program was automatically generated from an existing MatLab module and is thus not optimized for readability. Furthermore, the original MatLab model may

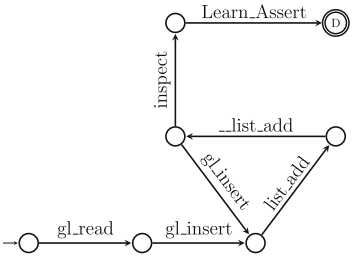


Fig. 13. Automaton produced for the'bubble_sort' example. $b^{wl} = 15$. $b = 3$.

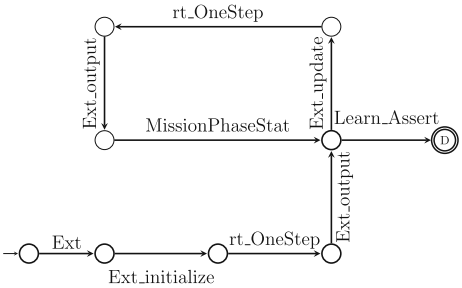


Fig. 14. Automaton produced for the 'docking' benchmark. $b^{wl} = 15$. $b = 4$.

not accurately describe how the respective program semantics was mapped to C. The automaton in Fig. 14 can explain the core behavior of the mapped program. It consists of a stepwise main simulation loop in which the logic of all mission-related phases is handled in the operation `MissionPhaseStat`. The source code and the learned automata of all our benchmarks are available online [1], where the reader may observe the effect of an interactive change of bounds.

7 Conclusions and Future Work

Our definition of $Fail(P)$ 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 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, hence aiding in program understanding – a direction that becomes especially relevant when the software in question is prohibitively large to be examined manually, or when the code owner is not available (or, as in our docking software example, the software was generated automatically). We demonstrated that the same technique can also be used for visually comparing different versions of a program (by presenting an automaton that captures the behavior in the symmetric difference between them), or for assisting in merging several development branches.

A possible extension is to adapt the framework to learn ω-regular languages, represented by Büchi automata (see [4,14] for the extension of L^* 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 [2], (though in general, the class of context-free grammars is not believed to be learnable [5]), providing us with the possibility of harnessing these algorithms in our framework.

As mentioned in Sect. 6.2, the modularity of our implementation allows us to replace CBMC with another component, acting as a 'teacher' in our framework. In particular, we can use a software testing tool as a 'teacher', thus potentially improving the scalability of the framework. The learned language, however, will likely differ from the one in Theorem 1 if the answers to queries are based on the results of software testing.

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. Hence, small automata are preferable, at least for manual analysis. 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 [19] for a learning algorithm with inconclusive answers).

References

1. http://www.cprover.org/learning-errors/
2. Angluin, D.: Learning k-bounded context-free grammars. Technical report, Dept. of Computer Science, Yale University (1987)
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
4. Angluin, D., Fisman, D.: Learning regular omega languages. In: Auer, P., Clark, A., Zeugmann, T., Zilles, S. (eds.) ALT 2014. LNCS, vol. 8776, pp. 125–139. Springer, Heidelberg (2014)
5. Angluin, D., Kharitonov, M.: When won't membership queries help? (extended abstract). In: Proceedings of 23rd STOC, pp. 444–454. ACM (1991)
6. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: Proceedings of 30th POPL, pp. 97–105 (2003)
7. Basu, S., Saha, D., Lin, Y.-J., Smolka, S.A.: Generation of all counter-examples. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 79–94. Springer, Heidelberg (2003)
8. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.J.: Explaining counterexamples using causality. Formal Methods Syst. Des. **40**(1), 20–40 (2012)
9. Beyer, D.: Software verification and verifiable witnesses. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 401–416. Springer, Heidelberg (2015)
10. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: the automata learning framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010)
11. Botinčan, M., Babić, D.: Sigma*: symbolic learning of input-output specifications. In: Proc. of 40th POPL, pp. 443–456. ACM (2013)
12. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)

13. Copty, F., Irron, A., Weissberg, O., Kropp, N.P., Kamhi, G.: Efficient debugging in a formal verification environment. STTT **4**(3), 335–348 (2003)
14. Farzan, A., Chen, Y.-F., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008)
15. Giannakopoulou, D., Rakamarić, Z., Raman, V.: Symbolic learning of component interfaces. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 248–264. Springer, Heidelberg (2012)
16. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. STTT **8**(3), 229–247 (2006)
17. Hoenicke, J., Leino, K., Podelski, A., Schäf, M., Wies, T.: Doomed program points. Formal Methods Syst. Des. **37**(2–3), 171–199 (2010)
18. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley, Reading (2000)
19. Leucker, M., Neider, D.: Learning minimal deterministic automata from inexperienced teachers. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 524–538. Springer, Heidelberg (2012)
20. The problem of automatic code merging (2012). http://www.personal.psu.edu/txl20/blogs/tks_tech_notes/2012/03/the-problem-of-automatic-code-merging.html