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, UK.
martin.chapman,hana.chockler@kcl.ac.uk
[2] Department of Computer Science, University of 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, 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 behaviour 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 behaviour 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 [15] 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. [12], 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 PDA (representing software) and a DFA

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

(representing a 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 'localised' by comparing it to a trace that does not violate the property.

It is clear, then, 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 to present the user with a DFA that summarises 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.

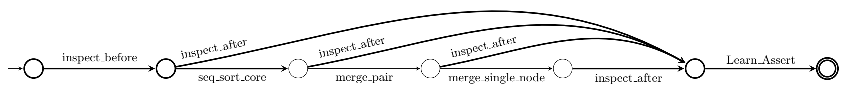


Fig. 1. An abstract description of a merge-sort program, where the letters are the function calls.

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 behaviour of which is communicated to L^* via an interface called the 'teacher'. L^* asks the teacher *membership* queries over Σ, namely whether $w \in L(\mathcal{U})$, where w is a word and $L(\mathcal{U})$ is the language of $\mathcal{U}$, and *conjecture* queries, namely whether for a given DFA $\mathcal{A}$, $L(\mathcal{A}) = L(\mathcal{U})$. The number of queries is bounded by $O(m^2n^3)$, where n is the number of states of the resulting (minimal) DFA, and m is the length of the longest feedback (counterexample). 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., [14], or to model the input-output relation in specific types of programs, in which that relation is sufficient for verifying certain properties [10].

Trivially, the language that describes a piece of a program, or the behaviours that fail an assertion, is neither finite or 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

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

given bound, corresponds to a real trace in the program. We will formalise this concept in Sect. 2. 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); }

void main() { // in is an input
  for (int i=0; i < in; i++) g(in);
  f();
}
```

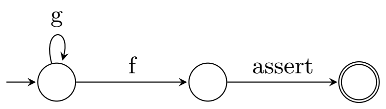


Fig. 2. A program and an automaton that we learn from it when using a low bound (< 100) on the word length. `in` models an input to the program.

In the next two sections, we define the language we learn precisely, and describe our method in detail, while assuming that the reader is somewhat familiar with L^*. L^* is mostly used as a black-box in our framework. We describe various aspects of our system and our empirical evaluation of it in Sec. 4, and conclude with some ideas for future research in Sec. 5.

2 The language we learn

Our learning scheme is based on user-defined *events*, which can be anything a user chooses as their atoms for describing the behaviours 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 IDs constitute 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, which we denote by $\alpha(\pi)$. The language of such a program, denoted $L(P)$, is defined naturally by

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

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 uncomputable 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 regular, finite 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 Sec. 4 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:

Theorem 1. *For all $\pi \in P$:*

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

A proof can be found in [1].

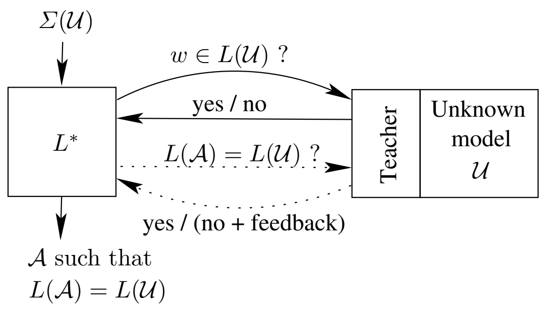
3 L^* and the queries

We assume the reader is somewhat familiar with L^* and only mention its essentials here, while focusing on how we answer the queries produced by this algorithm. In other words, we describe the so called 'teacher' and 'oracle', which are the terms used in Angluin's original description for the entities that answer the two queries. For simplicity we will unify them under the name 'teacher'.

Fig. 3 describes the interaction of L^* with the teacher. L^* learns $L(\mathcal{U})$, by querying the teacher with two types of questions:

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

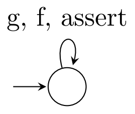
L^* maintains a table (called an 'observation table') that represents transitions and states. Let S be the set of states represented by this table. A table is said to be *closed* if $\forall s \in S.\ \forall \alpha \in \Sigma.\ \exists s' \in S.\ s' \equiv s \cdot \alpha$. In other words, the table is closed when it represents a complete transition function. To close its table, L^* asks the teacher multiple membership queries. For reasons that we will not detail here, 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 false, it analyses 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.



$\Sigma(\mathcal{U})$
$w \in L(\mathcal{U})$?
yes / no
L^*
$L(\mathcal{A}) = L(\mathcal{U})$?
yes / (no + feedback)
Teacher
Unknown model $\mathcal{U}$
$\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{U})$

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

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 $L(\mathcal{U})$. The answer is false 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 = $ `f`$\cdot$`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.

3.1 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 symbolises 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 [11]. 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 $idx \geq |w| \vee word[idx] \neq x$ then
 5:       assume(FALSE);                    ▷ Block paths incompatible with query
 6:    $idx = idx + 1$;
 7: function LEARN_ASSERT(bool $assertCondition$)
 8:    if $\neg assertCondition$ then                            ▷ Assertion fail
 9:       if $idx = |w| - 1$ then assert(FALSE);   ▷ $w \in L(\mathcal{A})$. 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.

Optimizations. We bypass a CBMC call and answer 'false' 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.

3.2 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.

Fig. 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 $\neg assertCondition$ 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 $assertCondition \wedge accepting[state]$ then assert(FALSE);      ▷ neg. feedback
9:     if $\neg assertCondition \wedge \neg accepting[state]$ 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 can belong to both a real 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’ in the end 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.

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

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.

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 in P to an exit point, which 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 $L(\mathcal{A}) - L_\Sigma(\mathcal{A}_{cfg})$ as the negative feedback.

4 System Description and Empirical Evaluation

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 8. 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 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 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 Sec. 2), 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, $\mathcal{A}$ 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).

Fig. 7 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

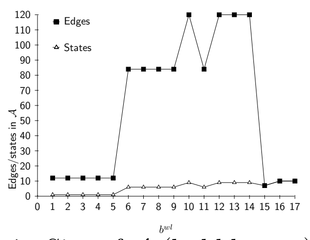


Fig. 7. Size of $\mathcal{A}$ (bubble sort).

state ('back edges'). Recall that a failing as-
sertion aborts execution, and hence no path can continue beyond it. Hence, the existence of such edges in $\mathcal{A}$ indicates that increasing b, b^{wl} or both should eventually remove them.

```
1: function LearnUptoBound(Program $P$)
2:     for $b \in [1 \ldots b_{\max}]$ do
3:         for $b^{wl} \in [b^{wl}_{\min} \ldots b^{wl}_{\max}]$ do
4:             $\mathcal{A}$ = learn $P$ with $b$ and $b^{wl}$;
5:             if $\mathcal{A} = \mathcal{A}_{prev}$ and $\mathcal{A}$ does not have back edges then return $\mathcal{A}$;
6:             $\mathcal{A}_{prev} = \mathcal{A}$;
```

Fig. 8. The autonomous discovery of the appropriate bounds.

Incrementality. The incremental nature of LearnUptoBound is exploited by our system for improving performance. We maintain a cache of words that have already already been proven to be in $L(\mathcal{U})$, 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. 3 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 32.28%.

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 from which the accepting state is inevitable [16].
- 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.

Evaluation. Our implementation of the process described so far is based on the automata library libalf [9] and on CBMC. We used it 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 [18], the Software-artifact Infrastructure Repository[2], and a model describing the behaviour of a space shuttle as it docks

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

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 summarised in Table 1. We also tested a strategy by which we do not return at Line 5 (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 benchmark in Table 1 reflect that.

					Time			CBMC Queries	
Target	$b^{wl}_{\min}$	b^{wl}	*It.*	b	[sec]	States	Edges	memb.	conj.
tcas	3	17	14	1	7.87	25	28	0.51%	34
bubble_sort	8	17	9	2	2.24	10	10	0.17%	69
bubble_sort	8	16	17	3	3.06	7	7	0.13%	89
bubble_sort	8	19	20	3	6.12	19	21	0.12%	99
merge_sort	4	8	4	1	0.11	4	3	0.92%	12
merge_sort	4	9	9	2	0.54	7	9	2.59%	32
sll_to_dll_rev	8	28	20	1	2.90	14	13	0.18%	39
sll_to_dll_rev	8	28	40	2	7.60	17	19	0.15%	78
defroster	25	29	4	1	36.31	14	18	0.01%	26
docking	5	8	3	1	0.47	7	6	0.86%	9
docking	5	8	6	2	0.76	7	6	0.86%	18
docking	5	11	12	2	1.63	11	11	1.07%	29

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.

Next, we present several examples of $\mathcal{A}$ from this benchmark set. Fig. 9 and 10 give $\mathcal{A}$ in the bubble sort benchmark with $b = 2$ and $b = 3$, where the value of b^{wl} satisfies the condition in Line 5. Bold edges indicate dominating events, e.g., the function `inspect` 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). Fig. 11 shows the docking benchmark, with $b = 2, b^{wl} = 11$. The source code and the corresponding learned automata of all our benchmarks are available online [1], where one may interactively change the bounds and see their effect.

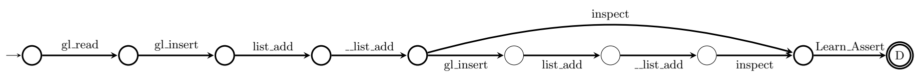


Fig. 9. Automaton produced for the 'bubble_sort' benchmark. $b^{wl} = 17$. $b = 2$.

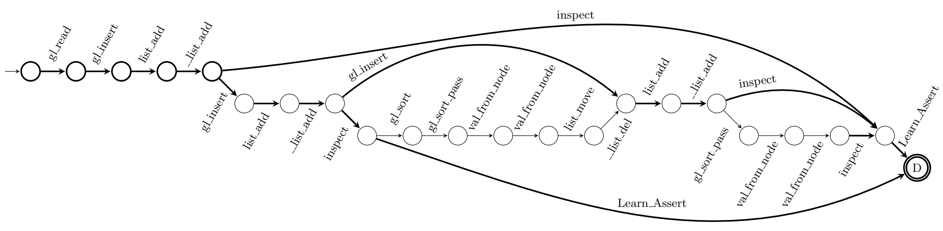


Fig. 10. Automaton produced for the 'bubble_sort' benchmark. $b^{wl} = 19$. $b = 3$.

5 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 to understand the cause of the error. More generally, the same technique can be used for visualising the behaviour of a program or parts thereof. This can be used, for example, for visually comparing different versions of a program (by presenting an automaton that captures the behaviour in the symmetric difference between them), or for assisting in merging several development branches.

Future directions. A possible extension is to adapt the framework to learn ω-regular languages, represented by Büchi automata (see [13, 4] for the extension of L^* to ω-regular languages). This extension would enable the learning of behaviours 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, for example, 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.

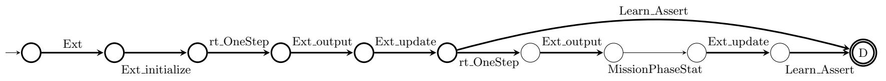


Fig. 11. Automaton produced for the 'docking' benchmark. $b^{wl} = 11$. $b = 2$.

One of the main goals of our framework is to present the language of error (or interesting behaviour) in a compact, easy to analyse 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 recognise three-valued answers (see [17] for a learning algorithm with inconclusive answers).

References

1. http://www.cprover.org/learning-errors/.
2. D. Angluin. Learning k-bounded context-free grammars. Technical report, Dept. of Computer Science, Yale University, 1987.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
4. D. Angluin and D. Fisman. Learning regular omega languages. In *Proc. of 25th ALT*, pages 125–139. Springer, 2014.
5. D. Angluin and M. Kharitonov. When won't membership queries help? (extended abstract). In *Proc. of 23rd STOC*, pages 444–454. ACM, 1991.
6. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proc. of POPL*, pages 97–105, 2003.
7. S. Basu, D. Saha, Y. Lin, and S. A. Smolka. Generation of all counter-examples for push-down systems. In *23rd FORTE*, volume 2767 of *LNCS*, pages 79–94, 2003.
8. I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler. Explaining counterexamples using causality. *Formal Methods in System Design*, 40(1):20–40, 2012.
9. B. Bollig, J. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon. libalf: The automata learning framework. In *Proc. of 22nd CAV*, pages 360–364, 2010.
10. M. Botinčan and D. Babić. Sigma*: symbolic learning of input-output specifications. In *Proc. of 40th POPL*, pages 443–456. ACM, 2013.
11. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. of 10th TACAS*, pages 168–176. Springer, 2004.
12. F. Copty, A. Irron, O. Weissberg, N. P. Kropp, and G. Kamhi. Efficient debugging in a formal verification environment. *STTT*, 4(3):335–348, 2003.
13. A. Farzan, Y. Chen, E. M. Clarke, Y. Tsay, and B. Wang. Extending automated compositional verification to the full class of omega-regular languages. In *Proc. of 14th TACAS*, pages 2–17. Springer, 2008.
14. D. Giannakopoulou, Z. Rakamaric, and V. Raman. Symbolic learning of component interfaces. In *Proc. of 19th SAS*, pages 248–264. Springer, 2012.
15. A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.
16. J. Hoenicke, K. Leino, A. Podelski, M. Schäf, and T. Wies. Doomed program points. *Formal Methods in System Design*, 37(2–3):171–199, 2010.
17. M. Leucker and D. Neider. Learning minimal deterministic automata from inexperienced teachers. In *Proc. of 5th ISoLA*, pages 524–538. Springer, 2012.
18. Competition on software verification 2015. http://sv-comp.sosy-lab.org/2015/.

A Proof of Theorem 1.

$\Rightarrow$ 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. For the third, falsely 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. For the third, falsely 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 false 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.

$\square$