Active Learning of Symbolic Automata for Reactive Programs via Dynamic Symbolic Mapper

YOEL KIM, Kyungpook National University, Republic of Korea
YUNJA CHOI*, Kyungpook National University, Republic of Korea

Active learning of formal behavior models from program source code is a powerful approach for a wide range of software analysis, validation, and verification tasks, including understanding system intent, automating specification mining, generating test oracles, and checking formal properties. Recent advances in active learning of symbolic automata, powered by program synthesis and model checking, provide both rich expressiveness and soundness guarantees for the learned models. However, these techniques often encounter significant performance bottlenecks, particularly when dealing with reactive programs that expose many program variables with large value domains.

This work introduces an extended active learning algorithm tailored for reactive programs by incorporating a novel dynamic symbolic mapper for learning symbolic automata. The mapper abstracts program behavior using learner-inferred predicates over program variables, encodes each valuation of these variables as a Boolean vector induced by these predicates (Boolean abstraction), and dynamically refines the abstraction in response to missing behaviors identified by the teacher. The mapper is granularity-aware: for teaching, it employs the coarsest predicates sufficient to expose missing behaviors, enabling broad exploration; for learning, it refines the abstraction only to the finest predicates necessary to resolve the uncovered gaps, trying to avoid refinements that could otherwise be triggered by coarse abstraction.

We evaluated our approach on 120 benchmarks, including SV-COMP tasks, Simulink model-driven programs, LeetCode problems, and representative embedded control software. The results show that our method learns 32 more symbolic automata and reduces the average active learning time by 55.6% compared to the state-of-the-art.

CCS Concepts: • **Software and its engineering** → **Model checking**; **Software reverse engineering**; • **Theory of computation** → **Automata over infinite objects**; **Abstraction**.

Additional Key Words and Phrases: Active Learning, Symbolic Automata, Mapper, Reactive Program

ACM Reference Format:
Yoel Kim and Yunja Choi. 2026. Active Learning of Symbolic Automata for Reactive Programs via Dynamic Symbolic Mapper. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE147 (July 2026), 23 pages. https://doi.org/10.1145/3808154

1 Introduction

Reactive programs [13] interact continuously with their environment: they receive inputs (e.g., events, interrupts, messages), update internal states, and produce outputs that may serve, in turn, as future inputs. Typical examples include device drivers, network protocols, and embedded control systems that are often safety-critical or mission-critical. Their stateful nature—where the next output depends on both current inputs and internal states—typically necessitates the use of global

*Corresponding author.

Authors' Contact Information: Yoel Kim, School of Computer Science and Engineering, Kyungpook National University, Daegu, Republic of Korea, kimyoel2305@knu.ac.kr; Yunja Choi, School of Computer Science and Engineering, Kyungpook National University, Daegu, Republic of Korea, yuchoi76@knu.ac.kr.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/7-ARTFSE147
https://doi.org/10.1145/3808154

variables intertwined with complex control and data flows, often forming infinite execution loops. Gaining a clear understanding of such systems is highly challenging, particularly when formal models or documentation are absent.

Approaches such as specification mining [6, 27, 49, 51, 53, 54, 64, 67] and model learning [1, 15, 24, 25, 30, 42–44] have tried to address this challenge using techniques ranging from formal methods to even large language models (LLMs) [54, 64]. Among them, recent state-of-the-art work by Jeppu et al. [42–44] leverages program synthesis and model checking to learn symbolic automata directly from program source code. To the best of our knowledge, this work is the most advanced active learning approach to date—both theoretically sound (guaranteeing that the learned model encompasses all behaviors of the target program) and directly applicable to source code. This is especially significant for reactive programs in safety- or mission-critical domains, where formal documentation is often missing due to legacy development practices or other practical constraints.

However, this approach also faces severe performance bottlenecks when programs expose many variables with large and diverse domains (Boolean, enumerated, integer, and floating-point). Its performance is highly sensitive to the number of variables and the size of their domains, which amplifies the scalability challenges of automata learning and limits practical applicability.

To address this issue, we present an extended active symbolic automata learning algorithm tailored for reactive programs by introducing a novel *dynamic symbolic mapper*. The mapper abstracts program behavior using predicates inferred by the learner and dynamically refines the abstraction in response to missing behaviors revealed by the teacher. By mapping program variables with diverse value domains to Boolean variables induced by predicates, the mapper enables the active learner to operate within a Boolean vector space while preserving both the expressiveness and soundness of the learned model. This Boolean abstraction is then dynamically refined during active learning using information revealed by counterexample traces provided by the teacher. Our mapper is *granularity-aware*: for teaching, it adopts the coarsest predicates sufficient to expose missing behaviors, supporting broad exploration; for learning, it refines the abstraction only to the finest predicates necessary to resolve these missing behaviors, trying to minimize the number of refinements that could have been triggered by coarse abstraction.

The key novelty of our dynamic symbolic mapper is two-fold: (i) a two-layered trace abstraction, which first applies program synthesis to generate transition predicates from program execution traces and then abstracts these traces into Boolean vectors using the synthesized predicates; and (ii) fully automated granularity-aware abstraction strategies that control the level of abstraction on demand. Unlike prior methods that assume manually crafted abstractions [25, 30] or rely on domain-specific abstractions [8, 17, 31], our dynamic symbolic mapper can be fully automated within the active learning loop, without any predefined abstractions.

We developed an extended active symbolic automata learning tool, called AutoCL, that implements the proposed dynamic symbolic mapper. We evaluated AutoCL on 120 reactive C programs derived from SV-COMP benchmarks [12], Simulink models [55], LeetCode-inspired tasks [50], and representative embedded software [14, 20, 28, 32, 65]. Under a three-hour time limit per program, AutoCL successfully learned 100 sound symbolic automata, compared with only 68 learned by the baseline approach [42–44], while reducing the average active learning time by 55.6%.

The major contributions of this paper are summarized as follows:

- We introduce a novel *dynamic symbolic mapper* that enables active learning of symbolic automata for reactive programs to operate within a Boolean vector space.
- We propose a *granularity-aware* extension of a state-of-the-art active symbolic automata learning algorithm that alternates between coarse-grained abstraction for teaching and fine-grained abstraction for learning.

- We develop a prototype tool AutoCL and evaluate it on 120 reactive C programs drawn from four benchmark suites, demonstrating improved active learning effectiveness and efficiency.

The remainder of this paper is organized as follows: §2 introduces the formal background for symbolic automata learning and the notation used throughout the paper; §3 presents the dynamic symbolic mapper, including its abstraction and refinement operations and granularity-aware strategies; §4 describes the overall workflow and active learning algorithm of AutoCL; §5 evaluates the effectiveness and efficiency of AutoCL on 120 benchmarks; §6 discusses related work; and §7 concludes with a discussion of limitations and future work.

2 Preliminaries

This section first introduces basic definitions (§2.1) and a running example (§2.2) used throughout the paper to explain our ideas. We also review the baseline active learner [42–44], outlining its overall process and technical approach (§2.3), and discuss the limitations of this baseline (§2.4).

2.1 Definitions

Definition 1 (Reactive Programs). A reactive program $\mathcal{P}$ is a tuple $(Q, X_{in}, X_{out}, f, q_0)$, where Q is a set of states, X_{in} and X_{out} are finite sets of input and output variables, and $f : Q \times Val(X_{in}) \rightarrow Q \times Val(X_{out})$ is a transition function mapping a current state and an input valuation to a next state and an output valuation, and $q_0 \in Q$ is the initial state. It allows a *feedback loop*, i.e., $X_{in} \cap X_{out} \neq \emptyset$.

Here, let $Dom(x)$ denote the value domain of a variable x. For a set of variables X, we write $Val(X) = \prod_{x \in X} Dom(x)$ for the set of all valuations over X. We refer to $X = X_{in} \cup X_{out}$, the set of input and output variables, as *observable variables*, enumerate them as $X = \{x_1, \ldots, x_k\}$, and write $X' = \{x'_1, \ldots, x'_k\}$ for their updated counterparts after applying the transition function f.

We write $X_{st} = X_{in} \cap X_{out}$ for the set of *observable state variables* (e.g., global variables), whose current values are used in the next iterations via the feedback loop.

An *observation* o_i is a valuation over $X \cup X'$ collected at the i-th feedback loop iteration, and an (execution) *trace* π is a finite sequence of such observations, $\pi = o_1, o_2, \ldots, o_n$.

Definition 2 (Symbolic Finite Automata). A symbolic finite automaton (SFA) $\mathcal{M}$ is a tuple (S, Φ, Δ, s_0) where S is a finite set of states, Φ is a set of predicates over $X \cup X'$, $\Delta \subseteq S \times \Phi \times S$ is the transition relation, and $s_0 \in S$ is the initial state.

Each $(s_i, \phi, s_{i+1}) \in \Delta$ represents a transition from the source state s_i to the target state s_{i+1}, where ϕ is a transition predicate consisting of variables in $X_{in} \subset X$, variables in $X_{out} \subset X'$, basic arithmetic operators, logical operators, and constant symbols.

A *symbolic path* of $\mathcal{M}$ is a finite sequence of transitions $\widehat{\pi} = s_0, \phi_1, s_1, \phi_2, \ldots, s_{n-1}, \phi_n, s_n$ such that $(s_{i-1}, \phi_i, s_i) \in \Delta$ for all $1 \leq i \leq n$. The set of concretized traces of $\widehat{\pi}$ is given by: $[[\phi_1]] \times [[\phi_2]] \times \cdots \times [[\phi_n]]$, where $[[\phi]]$ denotes the concretization operator mapping a predicate ϕ to the set of all observations that satisfy ϕ. The union of all such traces over all paths of $\mathcal{M}$ forms the *language* of $\mathcal{M}$, denoted $\mathcal{L}(\mathcal{M})$. A trace π is *accepted* by $\mathcal{M}$ if $\pi \in \mathcal{L}(\mathcal{M})$.

An SFA $\mathcal{M}$ is a *sound abstraction* of a reactive program $\mathcal{P}$ if it accepts the language of $\mathcal{P}$, i.e., $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{M})$, where $\mathcal{L}(\mathcal{P})$ is the set of all traces obtained by executing $\mathcal{P}$.

In this paper, we consider an SFA $\mathcal{M}$ whose states are identified by the value of a designated variable $st \in X_{st}$, called the *primary state variable*. We say that each state $s \in S$ represents a value $v \in Dom(st)$ of the primary state variable st, written as $st = v$. Accordingly, the next value of st after applying the transition function f determines the target state of each transition. Concretely, we restrict transitions to the form $(s,\ \varphi \wedge st' = v',\ s') \in \Delta$, where φ is a *guard predicate* over $X \cup X' \setminus \{st, st'\}$, and $st' = v'$ specifies the next value of the primary state variable.

2.2 Running Example

```
1 : while (true) {
2 :   w := ReadWater();
3 :   m := ReadMethane();

4 :   if (st = Ready) {
5 :     h := 0;
6 :     if (m < 600 ∧ w ≥ 70) {
7 :       st := Pump; }}
8 :   else if (st = Pump) {
9 :     h := h + 1;
10 :    if (m ≥ 600) {
11 :      st := Locked; a := true; }
12 :    else if (w ≤ 30) {
13 :      st := Ready; }
14 :    else if (h ≥ 3 ∧ w ≤ 50) {
15 :      st := Ready; }}
16 :  else if (st = Locked) {
17 :    if (h > 0) {
18 :      h := h − 1; }
19 :    if (¬a) {
20 :      st := Pump; }
21 :    if (m ≤ 550) {
22 :      a := false; }}
23 : }
```

Fig. 1. Pseudocode of the mine-pump controller used as a running example.

Fig. 1 shows pseudocode for a mine-pump controller [47]. The controller repeatedly reads water level w and methane level m (lines 2–3) and updates the primary state variable st together with the alarm a and the heat h. In *Ready*, it switches to *Pump* when w is high (lines 4–7); in *Pump*, it returns to *Ready* when w is low (lines 12–13), or when h becomes high even if w is only moderate (lines 14–15), and it moves to *Locked* when m is high (lines 10–11). In *Locked*, once m becomes safe again, it turns off a and switches back to *Pump* in the next cycle (lines 16–22).

Following Def. 1, we view the controller in Fig. 1 as a *reactive program* $\mathcal{P}$ that runs a *feedback loop*. In this program, we set $X_{in} = \{w, m, st, a, h\}$ and $X_{out} = \{st, a, h\}$, hence $X_{st} = \{st, a, h\}$. One loop iteration yields an *observation* $o \in Val(\{st, w, m, a, h, a', h', st'\})$; since f updates only X_{out}, $w, m \in (X_{in} \setminus X_{out})$ implies $w' = w$ and $m' = m$, and we omit w' and m' in the observation set. For example, $o := (st = Pump \wedge w = 68 \wedge m = 610 \wedge \neg a \wedge h = 1 \wedge a' \wedge h' = 2 \wedge st' = Locked)$.

2.3 The Baseline Active Learner

Jeppu et al. [42–44] proposed an innovative active learning approach that automatically infers symbolic automata from program source code. Its novelty lies in combining two techniques: (1) *inductive program synthesis* [5, 9], which automatically synthesizes expressive predicates from input-output examples, and (2) *software model checking* [19], which ensures the soundness of the learned automaton. Unlike prior approaches that either rely on simple partitioning schemes (e.g., decision tree splits [15] or interval-based abstractions [25]) with limited expressiveness or require user-specified atomic propositions [25]—thereby shifting the burden of expressiveness to the user—program synthesis enables the automated construction of rich predicates directly over program variables of heterogeneous types. Similarly, black-box active learning techniques typically approximate the correctness of the learned SFA using a finite set of test oracles, whereas integrating software model checking provides a rigorous soundness guarantee for the learned SFA with respect to the program.

Algorithm 1 illustrates the overall algorithm of the baseline approach.[1] Procedure ActiveLearn takes a target reactive program $\mathcal{P}$, a set of initial traces $\mathcal{T}$ obtained by running $\mathcal{P}$ with some program inputs, and a time limit t_{limit} as input. Then, the learner infers a candidate SFA $\mathcal{M}$ from $\mathcal{T}$ (line 3), while the teacher checks the soundness of $\mathcal{M}$ against $\mathcal{P}$ (line 4). If any missing behavior is detected, the teacher returns counterexamples (CEs) $\mathcal{T}_{new} \subset \mathcal{L}(\mathcal{P}) \setminus \mathcal{L}(\mathcal{M})$, after which the procedure updates the trace set $\mathcal{T} := \mathcal{T} \cup \mathcal{T}_{new}$ (line 7) and refines $\mathcal{M}$ in the next iteration. This procedure repeats until $\mathcal{M}$ achieves soundness (i.e., $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{M})$) (line 6). Otherwise, the procedure returns the best-effort SFA if it exceeds the time limit t_{limit} (line 8).

[1] This algorithm is our interpreted version based on the papers [42–44] and their implementation [41].

Algorithm 1 Baseline Active Learning Algorithm

1: **procedure** ActiveLearn($\mathcal{P}$: reactive program, $\mathcal{T}$: set of initial traces, t_{limit}: time limit)
2:   **while** $t_{\text{elapsed}} < t_{\text{limit}}$ **do**
3:     $\mathcal{M}$:= Learn($\mathcal{T}$);
4:     $\mathcal{T}_{new}$:= Teach($\mathcal{M}, P$);
5:     **if** $\mathcal{T}_{new} = \emptyset$ **then**
6:       **return** $\mathcal{M}$;
7:     $\mathcal{T} := \mathcal{T} \cup \mathcal{T}_{new}$;
8:   **return** $\mathcal{M}$;

9: **procedure** Learn($\mathcal{T}$: set of traces)
10:   $\Phi := \emptyset$; // transition predicate set
11:   **for each** $v \in Dom(st)$ **do**
12:     G_v := Group($\mathcal{T}, v$);
13:     $\Phi := \Phi \cup$ Synth(G_v);
14:   $\mathcal{T}_{sym}$:= Symbolic($\mathcal{T}, \Phi$);
15:   $\mathcal{M}$:= DFA($\mathcal{T}_{sym}$);
16:   **return** $\mathcal{M}$;

17: **procedure** Teach($\mathcal{M}$: SFA (S, Φ, Δ, s_0), $\mathcal{P}$: reactive program)
18:   $\mathcal{T}_{new} := \emptyset$;
19:   **for each** $(s, \phi, s') \in \Delta$ **do**
20:     $\mathcal{T}_{inf} := \emptyset$;
21:     $\mathcal{P}_s :=$ (**assume**(ϕ); $(q', X') := f(q, X)$; **assume**($\neg out(s')$););
22:     **while** *true* **do**
23:       τ := Select($\mathcal{L}(\mathcal{P}_s) \setminus \mathcal{T}_{inf}$);
24:       **if** $\tau = \perp$ **then**
25:         **break**;
26:       **if** (Feasible($\tau, \mathcal{P}$)) **then**
27:         $\mathcal{T}_{new} := \mathcal{T}_{new} \cup \{\tau\}$;
28:         **break**;
29:       **else**
30:         $\mathcal{T}_{inf} := \mathcal{T}_{inf} \cup \{\tau\}$;
31:   **return** $\mathcal{T}_{new}$;

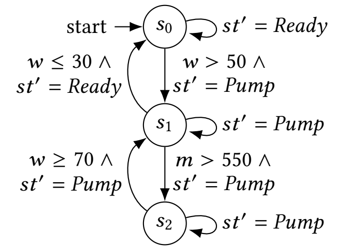
Given a set of traces $\mathcal{T}$, procedure Learn first groups the observations by the current value of the state variable st (line 12). For each group, it synthesizes guard predicates that distinguish different next values of st. Fig. 2a shows a trace $\pi = o_1, \ldots, o_8$ obtained from running the program in Fig. 1; for illustration, consider the group $G_{Ready} = \{o_1, o_2, o_4\}$ containing the observations whose current state is *Ready*. In this group, o_1 is followed by $st' = Ready$, whereas o_2 and o_4 are followed by $st' = Pump$. Procedure Synth synthesizes a distinguishing guard predicate (e.g., Synth$(G_{Ready}) = w > 50$) and then forms the corresponding transition predicate $w > 50 \land st' = Pump$ (line 13). Using the synthesized transition predicates, procedure Symbolic generates a symbolic lifting $\mathcal{T}_{sym}$ of $\mathcal{T}$ (as shown in Fig. 2b) by replacing each observation o with the transition predicate ϕ that o satisfies, i.e., $o \in [[\phi]]$ (line 14). Finally, procedure DFA applies a deterministic finite automaton (DFA) learning algorithm [7] that constructs an SFA $\mathcal{M}$ in Fig. 2c from $\mathcal{T}_{sym}$ (line 15).

i	st	w	m	a	h	a'	h'	st'
1	*Ready*	31	548	*F*	0	*F*	0	*Ready*
2	*Ready*	73	596	*F*	0	*F*	0	*Pump*
3	*Pump*	28	548	*F*	0	*F*	1	*Ready*
4	*Ready*	75	588	*F*	1	*F*	0	*Pump*
5	*Pump*	72	544	*F*	0	*F*	1	*Pump*
6	*Pump*	68	610	*F*	1	*T*	2	*Locked*
7	*Locked*	67	550	*T*	2	*F*	1	*Locked*
8	*Locked*	71	540	*F*	1	*F*	0	*Pump*

(a) Execution trace of Fig. 1

1 : $st' = Ready$
2 : $w > 50 \land st' = Pump$
3 : $w \leq 30 \land st' = Ready$
4 : $w > 50 \land st' = Pump$
5 : $st' = Pump$
6 : $m > 550 \land st' = Locked$
7 : $st' = Locked$
8 : $w \geq 70 \land st' = Pump$

(b) Symbolic lifting of (a)



(c) Learned SFA from (b)

Fig. 2. An example of symbolic automaton learning from an execution trace.

Procedure Teach iterates over each transition $(s, \phi, s') \in \Delta$ of $\mathcal{M}$. It first checks the *completeness* of each transition predicate ϕ using model checking to determine whether all possible transitions of the program $\mathcal{P}$ with the value pair $(st = v, st' = v')$ can be represented by (s, ϕ, s'); concretely, this is done by creating an annotated copy of $\mathcal{P}$, denoted $\mathcal{P}_s$, that assumes any observations satisfying ϕ, executes the transition function f, and then applies **assume**($\neg out(s')$), where $out(s')$ denotes the disjunction of all outgoing predicates of the target state s' (line 21). By model checking $\mathcal{P}_s$, it

can either identify a CE τ or conclude that the transition is complete if $\tau = \bot$. Each identified τ is subjected to *feasibility checking* against $\mathcal{P}$ (line 26): if feasible, τ is added to the new trace set $\mathcal{T}_{new}$ (line 27); otherwise, it is classified as infeasible and added to the infeasible set $\mathcal{T}_{inf}$ (line 30) and excluded from future checks (line 23). Teach returns the accumulated CE set $\mathcal{T}_{new}$ (line 31). If $\tau = \bot$ holds for all transitions (i.e., $\mathcal{T}_{new} = \emptyset$), $\mathcal{M}$ is proven to be a sound abstraction of $\mathcal{P}$.

2.4 Limitations of the Baseline Approach

The baseline active learner in §2.3 automatically synthesizes expressive transition predicates and provides soundness guarantees for the learned SFAs; however, these advantages come with significant *inefficiency*. Two major sources of inefficiency are (1) the repeated synthesis process and (2) the repeated feasibility checking process. These bottlenecks become more severe as the number of observable variables increases and their value domains expand, since each additional variable or domain value expands the search space for synthesizing candidate predicates and finding feasible CEs. In Learn, after each teaching step, the learner discards previously synthesized predicates—often computed at the cost of tens or hundreds of seconds—and recomputes them from scratch, and the growing number of traces further expands the synthesis search space. In Teach, the same expansion leads to a combinatorial explosion of candidate CEs that must be checked for feasibility. The nested loops at lines 19 and 22 iterate over transitions and candidate CEs, invoking model checking at every iteration and thus significantly increasing the overall cost.

Consequently, the overall performance of active learning for symbolic automata is highly sensitive to both the number of observable variables and the size of their value domains. In complex programs with many variables, these factors exacerbate inefficiency issues that the baseline approach cannot overcome. This limitation highlights the need for an effective *abstraction* of observable variables—one that reduces the search space without sacrificing the expressiveness and soundness.

3 Dynamic Symbolic Mapper

Our dynamic symbolic mapper abstracts program behavior represented as predicates over observable variables into *Boolean variables* (BVs), and dynamically refines the abstraction in response to missing behaviors revealed by the teacher. The mapper produces abstract traces and abstract SFAs over the Boolean vector space so that the learner synthesizes transition predicates using only logical operations, and the teacher checks feasibility over Boolean combinations, thereby reducing the number of infeasible CEs as well as the complexity of program synthesis.

3.1 Boolean Abstraction in Active Learning

Before presenting our dynamic symbolic mapper in full, we first highlight its key operation: *trace abstraction*. At a high level, given a finite set of BVs, trace abstraction maps each observation over observable variables to a Boolean valuation over BVs by assigning each BV the truth value of its corresponding predicate on the observation. We formalize this mapping and explain how it works for active learning.

Definition 3 (Trace Abstraction). Given a finite set of BVs $B = \{b_1, \dots, b_k\}$, where each b_i encodes the truth value of a unique predicate ψ_i over $X \cup X'$ (i.e., $b_i \Leftrightarrow \psi_i$), the *trace abstraction* is a function $\alpha : Val(X \cup X') \rightarrow Val(\{st, st'\} \cup B)$.

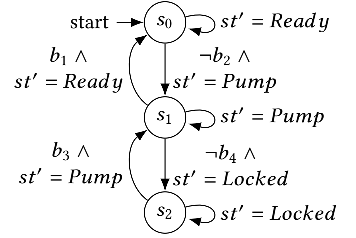
Here, $Val(\cdot)$ denotes the set of valuations over its argument. For an observation $o \in Val(X \cup X')$, the trace abstraction $\alpha(o)$ is the Boolean valuation over B induced by $\psi_1, \dots, \psi_k$, i.e., b_i is true in $\alpha(o)$ iff $o \models \psi_i$. We extend α to traces and trace sets as follows. For a trace $\pi = o_1 \cdots o_n$, we define $\alpha(\pi) = \alpha(o_1) \cdots \alpha(o_n)$. For a trace set $\mathcal{T}$, we define $\alpha(\mathcal{T}) = \{\alpha(\pi) \mid \pi \in \mathcal{T}\}$.

Trace abstraction enables both learning and teaching to operate in the Boolean vector space with lower cost. In the active learning workflow, we may first fix a BV set $B = \{b_1, \ldots, b_k\}$ and apply trace abstraction to an initial set of execution traces $\mathcal{T}$ to obtain an initial abstract trace set $\widehat{\mathcal{T}} = \alpha(\mathcal{T})$. The learner then infers an abstract SFA $\widehat{\mathcal{M}}$ from $\widehat{\mathcal{T}}$, whose guard predicates are Boolean formulas over B. The teacher checks $\widehat{\mathcal{M}}$ against a target reactive program $\mathcal{P}$ and returns an abstract CE trace $\hat{\tau}$ over $\{st, st'\} \cup B$.

We further illustrate this with the running example in Fig. 3: We instantiate $B = \{b_1, \ldots, b_8\}$ using predicates appearing in the code (Fig. 1), e.g., b_1 corresponds to the condition at line 12.

i	st	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	st'
1	*Ready*	*F*	*T*	*F*	*T*	*T*	*F*	*F*	*F*	*Ready*
2	*Ready*	*F*	*F*	*T*	*F*	*T*	*F*	*F*	*F*	*Pump*
3	*Pump*	*T*	*T*	*F*	*T*	*T*	*F*	*T*	*F*	*Ready*
4	*Ready*	*F*	*F*	*T*	*F*	*T*	*F*	*F*	*F*	*Pump*
5	*Pump*	*F*	*F*	*T*	*T*	*T*	*F*	*T*	*F*	*Pump*
6	*Pump*	*F*	*F*	*F*	*F*	*F*	*F*	*T*	*F*	*Locked*
7	*Locked*	*F*	*F*	*F*	*T*	*T*	*T*	*T*	*F*	*Locked*
8	*Locked*	*F*	*F*	*T*	*T*	*T*	*F*	*F*	*F*	*Pump*

(a) Abstract trace of Fig. 2a



(b) Abstract SFA learned from (a)

Fig. 3. An example of trace abstraction. Here, $b_1 \Leftrightarrow (w \leq 30)$, $b_2 \Leftrightarrow (w \leq 50)$, $b_3 \Leftrightarrow (w \geq 70)$, $b_4 \Leftrightarrow (m \leq 550)$, $b_5 \Leftrightarrow (m < 600)$, $b_6 \Leftrightarrow (a = T)$, $b_7 \Leftrightarrow (h' > 0)$, and $b_8 \Leftrightarrow (h' \geq 3)$.

Fig. 3a shows an *abstract trace* $\alpha(\pi)$ obtained by applying the trace abstraction to the execution trace π in Fig. 2a. Each BV specified in the caption of Fig. 3 is evaluated on every observation in π to determine its value. $\alpha(\pi)$ is then passed to the learner, which reasons only about logical operations over B instead of linear arithmetic over integer-valued observations (e.g., w, m, and h), thus reducing the cost of predicate synthesis and the overall learning cost.

Fig. 3b shows an *abstract SFA* $\widehat{\mathcal{M}}$ obtained by learning from Fig. 3a. This abstract SFA then enables the teacher to perform feasibility checking over B rather than over the observable variables $\{w, m, a, h\}$. For example, let $\hat{\tau} := (st = Pump \wedge b_1 \wedge \cdots \wedge \neg b_4 \wedge \cdots \wedge b_8 \wedge st' = Pump)$ be a candidate abstract CE obtained from completeness checking. The teacher then invokes a model checker to check whether $\hat{\tau}$ is feasible in the program $\mathcal{P}$. If $\hat{\tau}$ is feasible, it is added to the existing abstract trace set for the next learning iteration. Otherwise, $\hat{\tau}$ is discarded from further checks, excluding all observations mapped to $\hat{\tau}$. In contrast to teaching the SFA in Fig. 2c, the search space for the teacher is reduced to $2^{|B|}$ from $Dom(w) \times Dom(m) \times Dom(a) \times Dom(h)$.

It is clear that trace abstraction can reduce the cost of active learning if $2^{|B|}$ is significantly smaller than $Dom(w) \times Dom(m) \times Dom(a) \times Dom(h)$. However, it is non-trivial whether there is such a B and how we can find it. BVs can be manually crafted by a domain expert or obtained automatically by analyzing the program code (e.g., by extracting predicates such as branch conditions), but it is highly likely that the fixed BV set becomes either unnecessarily large or insufficiently small. In Fig. 3b, $\{b_5, \ldots, b_8\}$ do not occur in any transition predicate; nevertheless, they still expand the search space of candidate CEs, since the number of Boolean combinations grows exponentially as $2^{|B|}$. Conversely, if B is insufficient, it may induce an *abstract trace conflict*, where two observations have identical valuations over $\{st, b_1, \ldots, b_k\}$ but different st'. In this situation, the learner cannot synthesize a distinguishing predicate using only the given B. We address this issue by introducing a dynamic mapper that maintains BVs on demand during active learning.

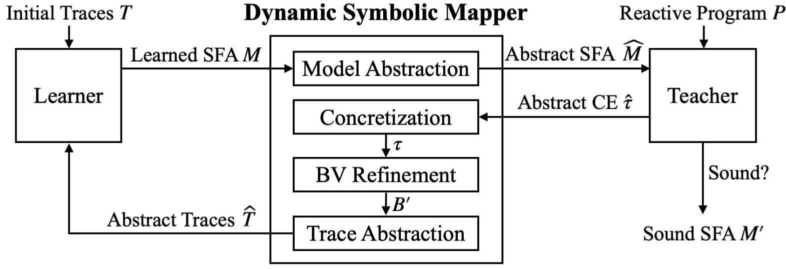


Fig. 4. Overview of dynamic symbolic mapper.

3.2 The Mapper Operations

We aim to dynamically identify a necessary and sufficient set of BVs for the Boolean abstraction at each active learning iteration. To this end, we need the following operations in addition to trace abstraction: (i) *model abstraction* to identify necessary BVs from a learned SFA, and (ii) *concretization* and *BV refinement* to identify missing BVs from abstract CEs generated by the teacher. Fig. 4 outlines the workflow of our dynamic symbolic mapper, which provides four operations: *model abstraction*, *concretization*, *BV refinement*, and *trace abstraction*.

The mapper starts with an empty BV set $B = \emptyset$, so the learner first infers an SFA $\mathcal{M}$ directly from the initial trace set $\mathcal{T}$. The *model abstraction* operation then extracts guard predicates from $\mathcal{M}$ to initialize B and rewrites $\mathcal{M}$'s guard predicates into Boolean expressions over B, yielding an abstract SFA $\widehat{\mathcal{M}}$. The mapper passes $\widehat{\mathcal{M}}$ to the teacher and receives an abstract CE $\hat{\tau}$; the *concretization* operation then produces a concretized CE $\tau \in [[\hat{\tau}]]$. If $\hat{\tau}$ exposes an abstract trace conflict, the mapper applies the *BV refinement* operation: it (i) collects a conflict group O_τ sharing the same Boolean valuation as $\hat{\tau}$ (thus $\tau \in O_\tau$), (ii) synthesizes a predicate over $X \cup X'$ that distinguishes O_τ, and (iii) introduces this predicate as a new BV. With the refined BV set B', the mapper applies the *trace abstraction* operation to the extended trace set $\mathcal{T} := \mathcal{T} \cup \{\tau\}$, producing $\widehat{\mathcal{T}}$, so the learner can infer a refined SFA using only logical operations over B' without any abstract trace conflicts. The model abstraction operation also prunes any BVs that no longer appear in the refined SFA, updating B' accordingly for the next teaching round.

We now formalize the dynamic symbolic mapper. It keeps a separate BV set B_v for each value $v \in Dom(st)$, enabling active learning to adapt to different regions of the Boolean vector space.

Definition 4 (Dynamic Symbolic Mapper). A *dynamic symbolic mapper* is a tuple $\mathcal{A} = (\mathcal{B}, \alpha, \alpha^{-1}, \text{Rewrite}, \text{Refine})$, where

- $\mathcal{B} = \{B_v\}_{v \in Dom(st)}$ is a family of BV sets, where each B_v is the finite set of BVs indexed by v (so B_v may differ across v). We write $B = \bigcup_{v \in Dom(st)} B_v$ to represent the global BV set.
- $\alpha : Val(X \cup X') \rightarrow Val(\{st, st'\} \cup B)$ is the *trace abstraction* function.
- $\alpha^{-1} : Val(\{st, st'\} \cup B) \rightharpoonup 2^{Val(X \cup X')}$ is the *concretization* function, where $\alpha^{-1}(\hat{o}) = \{\, o \in Val(X \cup X') \mid \alpha(o) = \hat{o} \,\}$ (and $\alpha^{-1}(\hat{o}) = \emptyset$ if $\hat{o}$ is infeasible).
- Rewrite : $\mathcal{M} \rightarrow \mathcal{B}' \times \widehat{\mathcal{M}}$ is the *model abstraction* function.
- Refine : $\mathcal{T}_{new} \rightarrow \mathcal{B}'$ is the *BV refinement* function.

Trace Abstraction and Concretization. Given an observation $o \in Val(X \cup X')$, let $o(x)$ denote the value of an observable variable x in o. Then, the trace abstraction of the observation $\alpha(o) \in Val(\{st, st'\} \cup B_{o(st)})$ is obtained by assigning each BV $b \in B_{o(st)}$ the truth value of b's corresponding predicate evaluated on o. We extend α to traces and trace sets as in Def. 3.

The concretization function α^{-1} is extended symmetrically, mapping abstract traces back to their corresponding concrete ones.

Model Abstraction. Given a learned SFA $\mathcal{M} = (S, \Phi, \Delta, s_0)$, for each value $v \in Dom(st)$ let $\Phi_v = \{\varphi \mid (s, \varphi \wedge st' = v', s') \in \Delta$ and s represents $st = v\}$ be the subset of guard predicates in Φ. Then, Rewrite($\mathcal{M}$) produces a family of BV sets $\mathcal{B} = \{B_v\}_{v \in Dom(st)}$ and an abstract SFA $\widehat{\mathcal{M}}$: it constructs each $B_v = \{(b_\psi \Leftrightarrow \psi) \mid \psi \in Sub(\Phi_v)\}$, where $Sub(\cdot)$ is the set of all subformulas occurring in its argument. It then obtains the abstract SFA $\widehat{\mathcal{M}}$ by rewriting each guard predicate $\varphi \in \Phi_v$ into a Boolean expression $Expr(B_v)$ via *syntactic substitution*—replacing every occurrence of each such subformula ψ in φ with its corresponding BV $b_\psi \in B_v$.

For example, if B_v consists of $b_2 \Leftrightarrow (w \leq 50)$ and $b_4 \Leftrightarrow (m \leq 550)$, then a guard predicate $\varphi = (w \leq 50) \wedge (m > 550)$ is rewritten as $b_2 \wedge \neg b_4$.

BV Refinement. Given a set of new abstract CEs $\mathcal{T}_{new}$, Refine($\mathcal{T}_{new}$) produces a refined family of BV sets $\mathcal{B}' = \{B'_v\}_{v \in Dom(st)}$ where each B'_v is the refined set of BVs. B'_v is introduced as follows: For each concretized CE $\tau \in \alpha^{-1}(\hat{\tau})$ with $v = \tau(st)$, (i) a conflict group $O_\tau = \{o \mid \alpha(o) = \hat{\tau}$ except for the value of $st'\}$ is collected; (ii) a distinguishing predicate ψ is synthesized by applying Synth(O_τ), which is the same procedure used in Algorithm 1 at line 13; and (iii) a refined set of BVs $B'_v = B_v \cup \{(b_\psi \Leftrightarrow \psi) \mid \psi \in Sub(\text{Synth}(O_\tau))\}$ is introduced, where B_v is the set of existing BVs.

$\widehat{G}_{Pump}$	st	b_1	b_4	st'
$\alpha(o_3)$	*Pump*	*T*	*T*	*Ready*
$\alpha(o_5)$	*Pump*	*F*	*T*	*Pump*
$\alpha(o_6)$	*Pump*	*F*	*F*	*Locked*
$\hat{\tau}$	*Pump*	*F*	*F*	*Pump*

(a) $\widehat{G}_{Pump}$ after teaching

O_τ	st	w	m	a	h	a'	h'	st'
o_6	*Pump*	68	610	*F*	1	*T*	2	*Locked*
τ	*Pump*	68	599	*F*	1	*F*	2	*Pump*

(b) Conflict group O_τ satisfying $\neg b_1 \wedge \neg b_4$

$\widehat{G}_{Pump}$	st	b_1	b_4	b_5	st'
$\alpha(o_3)$	*Pump*	*T*	*T*	*T*	*Ready*
$\alpha(o_5)$	*Pump*	*F*	*T*	*T*	*Pump*
$\alpha(o_6)$	*Pump*	*F*	*F*	*F*	*Locked*
$\hat{\tau}$	*Pump*	*F*	*F*	*T*	*Pump*

(c) Refined $\widehat{G}_{Pump}$ by adding b_5

Fig. 5. Synthesis of a new BV b_5 to resolve the abstract trace conflict in (a) by synthesizing a predicate from the conflict group in (b). The refined abstract observation group $\widehat{G}_{Pump}$ after refining B_{Pump} is shown in (c).

Running Example. We use the same running example in Fig. 1 to walk through the dynamic mapper step by step, showing how it dynamically updates the BV sets.

Suppose that the learner is given the execution trace in Fig. 2a and infers the initial SFA $\mathcal{M}_1$ as shown in Fig. 2c. From $\mathcal{M}_1$ we have $\Phi_{Ready} = \{(w > 50)\}$, $\Phi_{Pump} = \{(w \leq 30), (m > 550)\}$, and $\Phi_{Locked} = \{(w \geq 70)\}$. Φ_{Ready} includes the guard $(w > 50)$ because the initial state s_0 represents *Ready*, and $\mathcal{M}_1$ contains an outgoing transition of s_0: $(s_0, w > 50 \wedge st' = Pump, s_1) \in \Delta$. Starting from $\mathcal{B} = \emptyset$, applying the model abstraction Rewrite($\mathcal{M}_1$) yields $(\mathcal{B}, \widehat{\mathcal{M}_1})$, where $\mathcal{B} = \{B_{Ready}, B_{Pump}, B_{Locked}\}$, with $B_{Ready} = \{b_2 \Leftrightarrow (w \leq 50)\}$, $B_{Pump} = \{b_1 \Leftrightarrow (w \leq 30), b_4 \Leftrightarrow (m \leq 550)\}$, and $B_{Locked} = \{b_3 \Leftrightarrow (w \geq 70)\}$. The resulting abstract SFA $\widehat{\mathcal{M}_1}$ is shown in Fig. 3b.

Suppose the teacher returns new CEs $\mathcal{T}_{new}$ from $\widehat{\mathcal{M}_1}$. For simplicity, we assume that the teacher checked only one transition $(s_0, \neg b_2 \wedge st' = Pump, s_1)$; thus $\mathcal{T}_{new} = \{\hat{\tau}\}$, where $\hat{\tau} := (st = Pump \wedge \neg b_1 \wedge \neg b_4 \wedge st' = Pump)$. Fig. 5a shows an abstract observation group obtained by applying the trace abstraction using B_{Pump} after teaching, $\widehat{G}_{Pump} = \{\alpha(o_3), \alpha(o_5), \alpha(o_6), \hat{\tau}\}$. This group exhibits an *abstract trace conflict*: $\hat{\tau}$ shares the same valuation $\neg b_1 \wedge \neg b_4$ as $\alpha(o_6)$ but with a different value of st', indicating that $B_{Pump} = \{b_1, b_4\}$ is insufficient. To resolve this conflict, we apply the BV refinement Refine($\mathcal{T}_{new}$) as follows. The mapper (i) concretizes $\hat{\tau}$ into an observation $\tau \in \alpha^{-1}(\hat{\tau})$, e.g., $\tau := (st = Pump \wedge w = 68 \wedge m = 599 \wedge \neg a \wedge h = 1 \wedge \neg a' \wedge h' = 2 \wedge st' = Pump)$; (ii) constructs a conflict group O_τ as shown in Fig. 5b; and (iii) synthesizes a distinguishing predicate $m < 600$ by Synth(O_τ) and introduces it as a new BV b_5, yielding $B'_{Pump} = \{b_1, b_4, b_5\}$ and the refined observation group in Fig. 5c.

In the same way, after teaching the remaining six transitions, we can extend the CE set $\mathcal{T}_{new} = \{\hat{\tau}_1, \ldots, \hat{\tau}_7\}$. Applying Refine($\mathcal{T}_{new}$) then refines the BV sets to $B_{Ready} = \{b_2, b_5\}$, $B_{Pump} = \{b_1, b_4, b_5, b_8\}$, and $B_{Locked} = \{b_3, b_7\}$. After abstracting the extended trace set $\mathcal{T} := \mathcal{T} \cup \alpha^{-1}(\mathcal{T}_{new})$ with these refined BV sets, the learner infers the second SFA $\mathcal{M}_2$, as shown in Fig. 6a. The mapper then applies the model abstraction operation Rewrite($\mathcal{M}_2$) again to prune unnecessary BVs that, due to newly introduced BVs, no longer appear in any transition predicate in $\mathcal{M}_2$. As a result, we obtain $B_{Ready} = \{b_2, b_5\}$, $B_{Pump} = \{b_1, b_5, b_8\}$, and $B_{Locked} = \{b_7\}$.

3.3 Granularity-Aware Strategies

We further extend the dynamic symbolic mapper with *granularity strategies*, which control the granularity at which predicates are encoded into BVs: they either introduce one BV for a composite predicate as a whole or introduce multiple BVs for its atomic predicates.

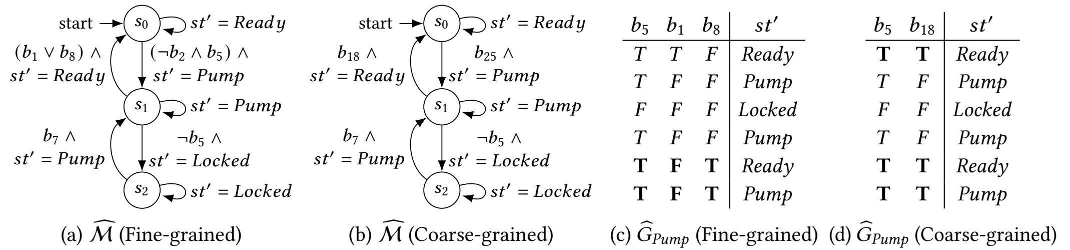
Definition 5 (Granularity-Aware Dynamic Symbolic Mapper). A *granularity-aware dynamic symbolic mapper* is an extension of the dynamic mapper $\mathcal{A}$ (Def. 4), where the two functions

$$\text{Rewrite} : \mathcal{M} \times G \rightarrow \mathcal{B}' \times \widehat{\mathcal{M}} \quad \text{and} \quad \text{Refine} : \mathcal{T}_{new} \times G \rightarrow \mathcal{B}'$$

are extended to incorporate a *granularity strategy* G, which specifies how BVs should be composed or split. Here, we explain the two basic granularity strategies: *split-to-atoms* and *never-split*:

Split-to-Atoms. All newly introduced BVs encode atomic predicates, i.e., we introduce one BV per atomic predicate. Let $Atom(\Phi_v) = \{\psi \mid \psi \text{ is an atomic formula of some } \varphi \in \Phi_v\}$. Then, Rewrite($\mathcal{M}$, *"split-to-atoms"*) rewrites each $\varphi \in \Phi_v$ by syntactic substitution, replacing every $\psi \in Atom(\Phi_v)$ in φ with b_ψ for $B_v = \{\, b_\psi \mid \psi \in Atom(\Phi_v)\}$. Likewise, Refine($\mathcal{T}_{new}$, *"split-to-atoms"*) refines B_v to $B'_v = B_v \cup \{b_\psi \mid \psi \in Atom(\text{Synth}(O_\tau))\}$.

Never-Split. All newly introduced BVs encode whole predicates, without decomposing them into atomic predicates. In this case, Rewrite($\mathcal{M}$, *"never-split"*) rewrites each guard predicate $\varphi \in \Phi_v$ as b_φ, and Refine($\mathcal{T}_{new}$, *"never-split"*) refines B_v to $B'_v = B_v \cup \{b_\psi \mid \psi = \text{Synth}(O_\tau)\}$.



b_5	b_1	b_8	st'
T	T	F	Ready
T	F	F	Pump
F	F	F	Locked
T	F	F	Pump
T	**F**	**T**	Ready
T	**F**	**T**	Pump

b_5	b_{18}	st'
T	**T**	Ready
T	F	Pump
F	F	Locked
T	F	Pump
T	**T**	Ready
T	**T**	Pump

(a) $\widehat{\mathcal{M}}$ (Fine-grained) (b) $\widehat{\mathcal{M}}$ (Coarse-grained) (c) $\widehat{G}_{Pump}$ (Fine-grained) (d) $\widehat{G}_{Pump}$ (Coarse-grained)

Fig. 6. Abstraction results under two BV granularity strategies. Here, $b_{18} \Leftrightarrow (b_1 \vee b_8)$ and $b_{25} \Leftrightarrow (\neg b_2 \wedge b_5)$.

As defined in Def. 5, we support four combinations of granularity strategies. Among these, two fixed strategies are straightforward: applying *never-split* for both teaching and learning (i.e., coarse-grained strategy), or *split-to-atoms* for both (i.e., fine-grained strategy).

Fig. 6 compares the two fixed granularity strategies. In the coarse-grained one, the mapper applies Rewrite($\mathcal{M}$, *"never-split"*), introducing BVs such as $b_{18} \Leftrightarrow (b_1 \vee b_8)$ and $b_{25} \Leftrightarrow (\neg b_2 \wedge b_5)$; this yields $\mathcal{B} = \{B_{Ready}, B_{Pump}, B_{Locked}\}$ with $B_{Ready} = \{b_{25}\}$, $B_{Pump} = \{b_5, b_{18}\}$, and $B_{Locked} = \{b_7\}$, and the resulting abstract SFA $\widehat{\mathcal{M}}$ is shown in Fig. 6b. The teacher then searches for abstract CEs involving only b_{25} when $st = Ready$. If, instead, the teacher worked directly with b_2 and b_5 under the fine-grained strategy, i.e., applying Rewrite($\mathcal{M}$, *"split-to-atoms"*), whose result is shown in Fig. 6a, it would need to consider four Boolean combinations, requiring additional feasibility checks.

Suppose that the two abstract observation groups $\widehat{G}_{Pump}$ in Fig. 6c and Fig. 6d are constructed from the same concrete observations under the fine- and coarse-grained strategies, respectively. Both strategies expose abstract trace conflicts, highlighted in bold, but the resulting conflict groups differ in size. Under the fine-grained strategy, only two observations satisfying $(b_5 \wedge \neg b_1 \wedge b_8)$ are required for BV refinement. Under the coarse-grained strategy, however, the BV $b_{18} \Leftrightarrow (b_1 \vee b_8)$ merges the three valuations $(b_1 \wedge b_8)$, $(\neg b_1 \wedge b_8)$, and $(b_1 \wedge \neg b_8)$, so an additional observation satisfying $(b_5 \wedge b_1 \wedge \neg b_8)$ is also required. Consequently, the coarse-grained strategy may increase both the synthesis search space and the BV refinement cost.

Overall, these fixed strategies expose different bottlenecks, while offering complementary strengths. This motivates us to alternate between coarse- and fine-grained abstractions to combine their benefits, maintaining BVs more adaptively.

4 AutoCL: An Extended Active Learner

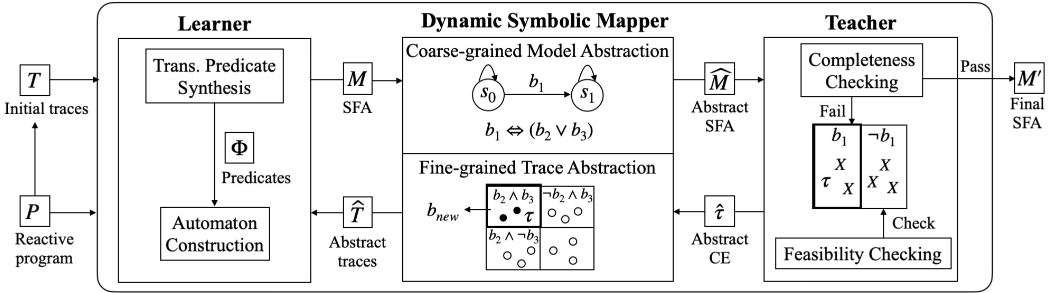


Fig. 7. Overall workflow of AutoCL.

In this section, we introduce AutoCL, an extended active symbolic automata learner designed for reactive programs with the aid of the dynamic symbolic mapper. The core idea is an *alternating granularity strategy*. For teaching, AutoCL applies the *never-split* strategy: BVs are kept as coarse predicates that are still sufficient to reveal missing behaviors, enabling broad exploration. For learning, it switches to the *split-to-atoms* strategy: BVs are split into the finest predicates so that refinement is restricted to only the observations necessary to resolve the exposed missing behavior, thereby minimizing the number of observations involved in BV refinement that would otherwise be included due to the coarse-grained abstraction.

Fig. 7 illustrates the workflow of AutoCL. Initially, we collect initial traces $\mathcal{T}$ by executing a reactive program $\mathcal{P}$, and the learner infers an initial SFA $\mathcal{M}$. The mapper then initializes a BV family $\mathcal{B}$ by extracting guard predicates from $\mathcal{M}$ and constructs an initial abstract SFA $\widehat{\mathcal{M}}$ using the *never-split* strategy, which keeps $\mathcal{B}$ as *coarse-grained BVs* (*coarse-grained model abstraction*). The teacher then checks $\widehat{\mathcal{M}}$ against $\mathcal{P}$ and returns a set of abstract CEs $\mathcal{T}_{new}$ expressed over these *coarse-grained BVs*, which is sent back to the mapper. Upon receiving $\mathcal{T}_{new}$, the mapper switches to the *split-to-atoms* strategy and produces *fine-grained BVs* by splitting $\mathcal{B}$, which reduces the sizes of conflict groups used for BV refinement. Accordingly, the mapper refines $\mathcal{B}$ to $\mathcal{B}'$ by synthesizing new predicates that resolve abstract trace conflicts for each abstract CE $\hat{\tau} \in \mathcal{T}_{new}$. The mapper then abstracts the traces $\mathcal{T}$ into the abstract traces $\widehat{\mathcal{T}}$ with $\mathcal{B}'$ (*fine-grained trace abstraction*). After that, the learner re-learns $\mathcal{M}$ using $\widehat{\mathcal{T}}$ expressed over *fine-grained BVs*. Finally, the mapper reconstructs $\widehat{\mathcal{M}}$ by switching to the *never-split* strategy for the next teaching round.

4.1 Algorithm

Algorithm 2 describes the active learning procedure of AutoCL. The algorithm takes as input a target reactive program $\mathcal{P}$, an initial set of traces $\mathcal{T}$, and a time limit t_{limit}; it outputs a learned SFA $\mathcal{M}$ and terminates if it does not find any more CEs or the time limit is reached. In more detail:

Algorithm 2 AutoCL's Active Learning Algorithm

```
1:  procedure AutoCL(𝒫: target reactive program, 𝒯: set of initial traces, t_limit: time limit)
2:      𝒜 := (ℬ, α, α⁻¹, Rewrite, Refine); // initialize mapper with empty BV family (ℬ := ∅)
3:      ℳ := Learn(𝒯); // learn initial SFA
4:      while t_elapsed < t_limit do
5:          𝒜.ℬ, ℳ̂ := 𝒜.Rewrite(ℳ, "never-split"); // switch to coarse-grained BVs and generate abstract SFA
6:          𝒯_new := Teach(ℳ̂, 𝒫); // teach SFA
7:          if 𝒯_new = ∅ then
8:              return ℳ;
9:          𝒜.ℬ, _ := 𝒜.Rewrite(ℳ, "split-to-atoms"); // switch to fine-grained BVs
10:         for each τ̂ ∈ 𝒯_new do
11:             𝒯 := 𝒯 ∪ {τ ∈ 𝒜.α⁻¹(τ̂)}; // concretize the abstract CE
12:         𝒜.ℬ := 𝒜.Refine(𝒯_new, "split-to-atoms"); // synthesize new BVs
13:         𝒯̂ := 𝒜.α(𝒯); // generate abstract traces
14:         ℳ := Learn(𝒯̂); // re-learn SFA
15:     return ℳ; // return best-effort SFA
```

(1) *Initialization* (lines 2–3): The mapper $\mathcal{A}$ starts with the empty family $\mathcal{B} = \emptyset$, and the learner infers an initial SFA $\mathcal{M}$ derived from the initial trace set $\mathcal{T}$.
(2) *Coarse-grained model abstraction for teaching* (lines 5–6): Applying the *never-split* strategy, the mapper creates coarse-grained BVs from the given SFA $\mathcal{M}$ and abstracts it into the abstract SFA $\widehat{\mathcal{M}}$. The teacher then returns a set of abstract CEs $\mathcal{T}_{new}$.
(3) *Fine-grained trace abstraction for learning* (lines 9–14): Switching to the *split-to-atoms* strategy, the mapper utilizes the fine-grained BVs. For each abstract CE $\hat{\tau} \in \mathcal{T}_{new}$, it synthesizes new BVs that resolve an abstract trace conflict. After that, using the refined BV family $\mathcal{B}$, it generates the new abstract traces $\widehat{\mathcal{T}}$. The learner then takes $\widehat{\mathcal{T}}$ and re-learns the SFA $\mathcal{M}$.
(4) *Termination* (lines 7–8 and 15): The algorithm terminates when one of two conditions is met: (i) no new CEs are found, or (ii) the time limit t_{limit} is reached, in which case the current best-effort SFA is returned.

Soundness Proof (Sketch). We informally argue that Algorithm 2 returns a sound SFA when Teach$(\widehat{\mathcal{M}}, \mathcal{P})$ returns no abstract CEs, under the assumption that the baseline approach (Algorithm 1) produces a sound SFA. Suppose that the baseline produces a sequence of SFAs during the iterations of Algorithm 1 and terminates at the n_{th} iteration without any CEs: $\mathcal{L}(\mathcal{M}_1) \subseteq \mathcal{L}(\mathcal{M}_2) \subseteq \cdots \subseteq \mathcal{L}(\mathcal{M}_n)$ and $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{M}_n)$. At any iteration $1 \leq i \leq n-1$ where the teacher returns a CE τ, by definition we have $\tau \in \mathcal{L}(\mathcal{P}) \setminus \mathcal{L}(\mathcal{M}_i)$, and the subsequent Learn yields a new SFA that subsumes this CE, i.e., $\tau \in \mathcal{L}(\mathcal{M}_{i+1})$. We now show that Algorithm 2 produces an abstract SFA $\widehat{\mathcal{M}_i}$ for the i-th iteration that satisfies $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\widehat{\mathcal{M}_i})$ as follows: Initially, lines 2–3 of Algorithm 2 create the same initial SFA as that of the baseline; Rewrite operations at lines 5 and 9 rewrite a guard predicate φ as a Boolean expression $Expr(B_v)$ that is syntactically (and thus semantically) equivalent to φ for any transition $(s, \varphi \wedge st' = v', s') \in \Delta$, where s represents $st = v$; for any observation o with $o(st) = v$, we have $o \models \varphi \Leftrightarrow \alpha(o) \models Expr(B_v)$.

Base Case. If Teach returns an empty set at line 6 in the first iteration, the SFA returned at line 8 is the same as that of the baseline. Therefore, $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{M}_1) \subseteq \mathcal{L}(\widehat{\mathcal{M}_1})$ holds.

Inductive Hypothesis. Suppose that $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\widehat{\mathcal{M}_i})$ after the i-th iteration.

Inductive Step. If Teach returns an empty set at line 6 in the $(i+1)$-th iteration, the baseline can only return an empty set due to the inductive hypothesis. Otherwise, suppose that a CE $\tau \in \mathcal{L}(\mathcal{M}_{i+1}) \setminus \mathcal{L}(\mathcal{M}_i)$ such that $\tau \not\models \varphi$ is identified from $(i+1)$-th teaching process using the baseline algorithm. We note that $\tau \not\models \varphi$ implies $\tau \not\models Expr(B)$ and $\hat{\tau} \not\models Expr(B)$, where $\hat{\tau} = \alpha(\tau) \notin \mathcal{L}(\widehat{\mathcal{M}_i})$, by construction, as α is a Boolean abstraction of τ and $Expr(B)$ is a simple syntactic rewrite of φ. As such $\hat{\tau}$ exists, Teach returns a non-empty set at line 6 the $(i+1)$-th iteration. $\hat{\tau} \not\models Expr(B')$ also holds after B is refined into B' after line 9 because the rewriting does not change the semantics of the Boolean expression. Therefore, we see that $\forall \tau \in \mathcal{L}(\mathcal{M}_{i+1}) \setminus \mathcal{L}(\mathcal{M}_i)$, $\exists \hat{\tau} = \alpha(\tau) \in \mathcal{L}(\widehat{\mathcal{M}_{i+1}}) \setminus \mathcal{L}(\widehat{\mathcal{M}_i})$. Together with the inductive hypothesis $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\widehat{\mathcal{M}_i})$, this proves that $\mathcal{L}(\mathcal{M}_{i+1}) \subseteq \mathcal{L}(\widehat{\mathcal{M}_{i+1}})$ for all i; if $\mathcal{M}_n$ is a sound abstraction of $\mathcal{P}$, so is $\widehat{\mathcal{M}_n}$.

4.2 Implementation

We implemented our approach AutoCL in Java, building on Trace2Model [44] for learning symbolic automata, an SMT solver, CVC5 (v1.1.2) [9] for synthesizing predicates, and a bounded model checker, CBMC (v5.43) [19] for checking model completeness and the feasibility of CEs during teaching. Concretization is also performed by CBMC as it automatically generates concrete witnesses for each feasibility check of candidate abstract CEs. We also implemented an automated instrumentation tool to identify feedback loops and to trace the values of observable variables. The tool is written in C++, performing static analysis with Clang. Given a primary state variable st, it analyzes control and data dependencies of all assignment statements to st, allowing us to prune irrelevant observable variables that do not affect any update of st. A more detailed description of this component is omitted due to space constraints and lies outside the scope of this paper.

5 Evaluation

To evaluate our approach, we assess its performance with three research questions:

RQ1: AutoCL vs. Baseline. Does AutoCL improve the efficiency and effectiveness of active learning compared to the baseline [43]? We evaluate the active learning completion rate within the given time budget and the correctness of the learned SFAs.

RQ2: Dynamic vs. Static Symbolic Mappers. How much overhead does the dynamic mapper introduce through its operations during active learning? To quantify this overhead, we perform an ablation study by implementing a *static symbolic mapper* that performs only trace abstraction with a fixed BV set (as defined in §3.1).

RQ3: Coarse- vs. Fine-grained AutoCL Variants. How effective is AutoCL's alternating granularity strategy? To assess its impact, we perform an ablation study with two AutoCL variants, AutoCL-Coarse and AutoCL-Fine, which implement the two fixed granularity strategies in §3.3.

5.1 Benchmarks

We constructed four benchmark sets consisting of a total of 120 reactive C programs. The benchmark sets are organized as follows: *Simulink* (45), *SV-COMP* (25), *LeetCode* (25), and *Embedded* (25).

Simulink Benchmarks. We use 45 benchmark programs from the baseline approach [43], derived from Simulink Stateflow examples [55], to check the correctness of the AutoCL implementation in comparison with the baseline approach.

SV-COMP Benchmarks. We randomly selected 25 programs from seven SV-COMP [12] benchmark categories that are inherently reactive or stateful: (1) *ntdrivers-simplified*, (2) *product-lines*, (3) *systemc*, (4) *seq-mthreaded*, (5) *eca-rers2012*, (6) *psyco*, and (7) *openssl-simplified*. For example, *systemc* (SystemC) [60] is a modeling language for embedded systems; these systems are naturally reactive;

eca-rers2012 originates from the Rigorous Examination of Reactive Systems (RERS) Challenge [38]; *psyco* includes benchmarks generated by PSYCO [37], an active learner for symbolic interfaces from software components; and *openssl-simplified* is derived from OpenSSL [59], a stateful network protocol implementation.

LeetCode Benchmarks. We generated 25 programs from programming assignment descriptions in LeetCode [50] by prompting ChatGPT (GPT-4o) [58] to confirm the general applicability of AutoCL. Most LeetCode problems are designed to implement a single function that produces an output based on given inputs, and thus are not inherently reactive. Therefore, we instructed GPT-4o to transform these problems into reactive programs.

Embedded Software Benchmarks. We selected 25 embedded control benchmarks, comprising two groups. The first group consists of five representative reactive controllers: (1) a garbage-collector robot [14], (2) a collision-avoidance controller[2], (3) an elevator controller [20], (4) an automotive object-following controller [32], and (5) an automotive window-lift controller [65]. The second group consists of 20 benchmarks drawn from TACLeBench [28], which notably includes satellite on-board software (e.g., DEBIE) and UAV autopilot control software (e.g., PapaBench). Since many TACLeBench programs are multi-tasking, we learn an SFA for each task separately, treating variables updated by other tasks as input variables. Overall, these programs represent typical reactive and stateful control logic in embedded systems, where external events (e.g., button presses or sensor inputs) trigger different state transitions depending on various conditions.

Table 1 shows the summary: from left to right, it shows the number of programs ($|P|$); the number of programs containing nested loops (# Nest); per-program averages of lines of code (LoC), domain size of the primary state variable ($|Dom(st)|$), and the number of observable variables ($|X|$) and their type distribution—the numbers of Boolean (# Bool), enumerated (# Enum), and numeric (integer or floating-point) (# Num) variables.

Table 1. Summary of the benchmark sets.

Benchmark	$\|P\|$	# Nest	LoC	$\|Dom(st)\|$	$\|X\|$	# Bool	# Enum	# Num
Simulink	45	5	236	3.7	7.0	1.2	4.3	1.4
SV-COMP	25	5	1,526	4.6	34.8	1.6	11.0	22.2
LeetCode	25	17	187	5.2	25.8	4.4	1.5	19.9
Embedded	25	16	1,242	5.0	19.1	1.3	3.5	14.3
Avg.			704	4.5	19.2	2.0	5.0	12.3

5.2 Experimental Setup

All experiments were conducted on an AMD Ryzen Threadripper PRO 7995WX CPU (96 cores, 2.5 GHz) with 256 GB of RAM, running Ubuntu 22.04. The following are the details:

- *Selection of primary state variables*: We prompted GPT-4o on each program to obtain a list of the most important variables of interest, from which we selected one as the primary variable.
- *Generation of initial traces*: We generated initial traces by running each program with a grey-box fuzzer AFL++ [29] for 600 seconds.
- *Options for bounded model checking*: We set the bound k (i.e., the maximum number of feedback-loop iterations explored by CBMC) per benchmark. Starting from an initial value

[2]It was originally provided as one of the mini-projects for the course at https://www.it.uu.se/edu/course/homepage/modbasutv/, but the page is no longer accessible.

(e.g., $k = 10$), we increased k stepwise in preliminary runs until the model structure (i.e., the set of states and transitions) no longer changed.
- *Options for predicate synthesis*: We adopted the default grammar provided in Trace2Model (with linear integer arithmetic theory) for transition predicate synthesis. Additionally, all constant values used in each program were extracted and supplied to CVC5.
- *Measurement and time limit for active learning*: We set a 3-hour wall-clock timeout per benchmark. For each run, we separately measured the time spent on Learn (SFA inference, trace abstraction, and BV refinement) and on Teach (model abstraction and concretization). We report average times over all benchmarks, counting timed-out runs as 3 hours.

5.3 RQ1: AutoCL vs. Baseline

Overall Results. Aggregated over all runs, AutoCL spent 2,184 seconds per benchmark on average (1,896 teaching + 288 learning), while the baseline spent 4,915 seconds (4,587 + 328). Thus, AutoCL reduced the average time by 55.6%, with most of the reduction coming from the teaching phase. Notably, AutoCL executed more active learning iterations in total (2,998 vs. 979), yet each iteration was substantially cheaper: the learning cost decreased from 40.2 seconds to 11.5 seconds per iteration (3.5×) and the teaching cost decreased from 562.2 seconds to 75.9 seconds (7.4×).

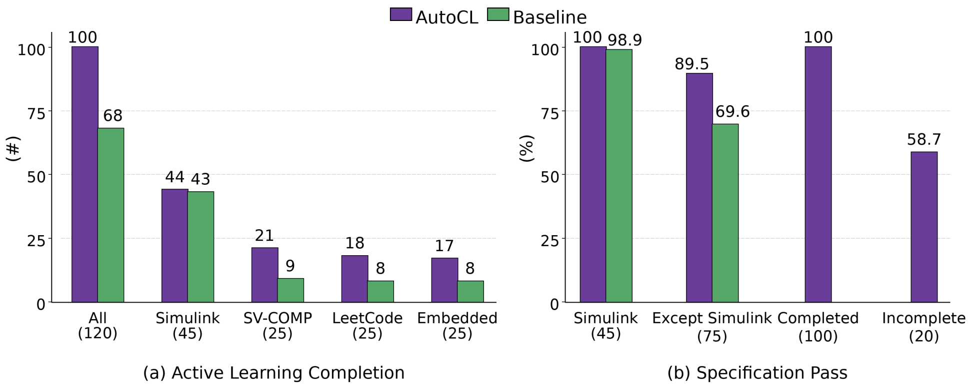


Fig. 8. (a) Number of completed benchmarks and (b) specification pass rate across benchmark sets.

Fig. 8a indicates the number of completions over benchmark sets (i.e., the number of benchmarks on which active learning succeeded within the 3-hour time limit). AutoCL consistently completed more benchmarks than the baseline: AutoCL succeeded in 100 out of 120 benchmarks, while the baseline completed only 68. On the Simulink set, which was used in the baseline's original evaluation, both approaches completed almost all benchmarks (44/45 vs. 43/45). However, on the remaining benchmarks, the baseline's performance dropped significantly: it succeeded in only 9 + 8 + 8 = 25 out of 75 (33.3%), compared to 21 + 18 + 17 = 56 (74.7%) for AutoCL.

Table 2. Comparison between completed vs. incomplete cases by AutoCL.

Benchmark	$\|P\|$	# Nest	LoC	$\|Dom(st)\|$	$\|X\|$	# Bool	# Enum	# Num	# Iter	Learn	Teach	$\|S\|$	$\|\Delta\|$
Completed	100	27	527	4.1	11.6	2.3	4.2	5.2	15.4	49s	412s	5.3	10.9
Incomplete	20	16	1,591	6.3	57.1	0.5	8.8	47.9	72.7	1,485s	9,314s	8.1	23.2

Table 2 presents a comparison between the benchmarks that AutoCL successfully completed and those it did not. The columns up to the number of numeric variables (# Num) are the same

categories as in Table 1. The remaining columns report the average outcomes, namely the number of active learning iterations (# Iter), the time spent in learning and teaching (in seconds), and the resulting model size in terms of the number of states $|S|$ and transitions $|\Delta|$. Across all these dimensions, the incomplete benchmarks exhibit substantially higher cost and complexity than the completed ones. A more detailed analysis is provided in §7.

Correctness. While completion results demonstrate the effectiveness of AutoCL, they do not reveal whether the learned SFAs are correct. We therefore assessed correctness by checking, with CBMC, transition relations and invariants extracted from each SFA on the target program. Each transition relation has the form $assert((st = v \wedge \varphi) \rightarrow (st' = v'))$, where st is the primary state variable, v and v' are its current and next values, and φ is the corresponding guard predicate. Transition invariants are formulated in the form $assert(\neg(st = v \wedge st' = v'))$ where $v, v' \in Dom(st)$ and there is no transition from v to v' in the SFA. On average, we generated 23.8 specifications per SFA on Embedded, 36.8 on LeetCode, 27.5 on SV-COMP, and 8.8 on Simulink; for the incomplete runs (timeouts), we used the best-effort SFA.

Fig. 8b shows the average pass rate of these specifications. On the Simulink set, both approaches showed nearly the same result (AutoCL at 100% and the baseline at 98.9%). However, when considering the remaining 75 benchmarks, the baseline pass rate dropped to 69.6%, whereas AutoCL maintained 89.5%. To further assess the quality of the learned SFAs, we aggregated the pass rates of all 100 completed SFAs, all of which achieved 100% pass rates. For the 20 incomplete SFAs, the average pass rate was 58.7%.

5.4 RQ2: Dynamic vs. Static Symbolic Mappers

Table 3. Ablation study of dynamic and static mappers over all benchmarks, excluding trace-conflict cases†.

Mapper	# Conflicts	Completed	# BV (L/T)	# Iter	Learn	Teach	$\alpha(\mathcal{T})$	Rewrite	Refine	Overhead
Dynamic	–	87	6.4/1.5	17.9	110s	1,548s	3.8s	8.4s	18.8s	31.0s
Static	21	63	34.4	2.5	4s	3,989s	0.2s	–	–	0.2s

† Excluding 21 trace-conflict cases, we report statistics on the remaining 99 benchmarks for both mappers.

Table 3 reports an ablation study of dynamic (AutoCL) and static mappers over all benchmarks, excluding trace-conflict cases exposed by the static mapper. We omit the definitions of the columns already introduced in the previous tables and focus on the additional columns. # Conflicts reports the number of trace-conflict benchmarks observed in the static mapper. # BV denotes the size of the BV set used by each mapper. For the dynamic mapper, it defines the average of $|B_v|$ over all $v \in Dom(st)$ for each active learning iteration. # BV (L) and # BV (T) are measured during the learning and teaching phases, respectively. For the static mapper, # BV simply denotes the size of its fixed BV set $|B|$. $\alpha(\mathcal{T})$, Rewrite, and Refine report, on average, the time spent in trace abstraction, model abstraction, and BV refinement, respectively; Overhead is their sum.

We implemented the static mapper by statically collecting predicates over the observable variables X appearing in conditional and loop statements, introducing one BV for each collected atomic predicate, and fixing the resulting BV set B throughout active learning. The static mapper exhibited 21 trace-conflict cases (# Conflicts), so the aggregated statistics in Table 3 exclude these cases. On the remaining benchmarks, the static mapper completed 63, while the dynamic mapper completed 87 out of 99. The static mapper spent substantially more time in teaching than the dynamic mapper (3,989 vs. 1,548 seconds), which is consistent with the BV set size (# BV: 34.4 vs. 6.4/1.5). While the static mapper has nearly no mapper overhead (0.2 seconds) and small learning time (4 seconds),

the large fixed BV set makes teaching dominate the overall cost. In contrast, the dynamic mapper reduces teaching time by maintaining the BV sets compact, at the price of additional mapper operations: its average overhead is 31.0 seconds per benchmark, consisting of $\alpha(\mathcal{T})$ (3.8), Rewrite (8.4), and Refine (18.8), which is negligible compared to the cost reduction obtained in teaching.

5.5 RQ3: Coarse- vs. Fine-Grained AutoCL Variants

Table 4. Ablation study of granularity variants (AutoCL, AutoCL-Coarse, AutoCL-Fine) over all benchmarks.

Variant	Completed	# BV (L/T)	# Iter	Learn	Teach	Refine (#/time)	$\lvert O_\tau \rvert$ (%)	$\lvert \mathcal{T}_{inf} \rvert$
AutoCL	100	7.6/1.7	25.0	288s	1,896s	13.5/65s	15.2%	31.2
AutoCL-Coarse	96	2.7/1.7	20.2	856s	1,589s	19.0/702s	49.9%	27.3
AutoCL-Fine	95	8.8/8.5	19.9	289s	2,410s	10.1/86s	17.5%	616.7

Table 4 reports an ablation study on the three granularity strategies: AutoCL, AutoCL-Coarse, and AutoCL-Fine. We omit the trace and model abstraction costs because they are almost the same across the three variants. Refine (#/time) reports the number of BV refinements and the total time spent on refinement per benchmark. $\lvert O_\tau \rvert$ (%) denotes the average ratio of observations used for BV refinements to the total number of collected observations. $\lvert \mathcal{T}_{inf} \rvert$ reports the average number of infeasible CEs.

Coarse-grained Strategy. AutoCL-Coarse spent, on average, 568 seconds more on learning than AutoCL, and most of this time was consumed by Refine (82.0%, 702/856). The main reason is that AutoCL-Coarse maintains fewer BVs (# BVs), which increases the ratio of observations included in resolving an abstract-trace conflict ($\lvert O_\tau \rvert$ (%)); this, in turn, suggests a higher predicate synthesis burden. In contrast, AutoCL completed 4 more benchmarks and spent 65 seconds on refinement (22.6% of learning), averaging 4.8 seconds per refinement. Moreover, AutoCL invoked Refine in only 13.5 out of 25 iterations on average, meaning that the remaining iterations proceeded without any conflicts and thus required no Refine.

Fine-grained Strategy. AutoCL-Fine spent 514 seconds more on teaching than AutoCL on average, which largely accounts for its higher overall cost. The main reason is the increase in infeasible CEs ($\lvert \mathcal{T}_{inf} \rvert$) during teaching: AutoCL-Fine maintains a much larger number of BVs in teaching (# BV (T); 8.5 vs. 1.7), which produces far more infeasible CEs (616.7 vs. 31.2) and thus increases the teaching burden. As a result, AutoCL completed five more benchmarks than AutoCL-Fine and reduced the average time by 19.1%.

Overall, these results indicate that AutoCL's alternating strategy maintains BVs adaptively across learning and teaching, making it more effective than either fixed-granularity variant.

5.6 Threats to Validity

5.6.1 External Validity. Our benchmark coverage is constrained primarily by the external backends that AutoCL relies on (CBMC and CVC5). Because these tools do not robustly support certain C features, AutoCL currently cannot handle programs with dynamic memory allocation, inline assembly, function pointers, or multi-threading, which may limit the representativeness of our benchmark set; these limitations could be mitigated by (i) adopting a program synthesizer that can reason about dynamic heap operations, (ii) translating inline assembly into semantically equivalent C code so that CBMC can analyze such blocks, and (iii) considering model checkers specialized for multi-threading (e.g., [66]). In addition, since CVC5 with linear integer arithmetic (LIA) theory assumes unbounded integers when synthesizing predicates, it cannot capture overflows/underflows, and its limited floating-point precision may lead to loss of precision.

5.6.2 Internal Validity. First, the use of CBMC as a teacher may produce unsound SFAs if an insufficient bound k is used (e.g., with $k = 2$ for the code in Fig. 1, an unsound SFA that contains no transition into $st = Locked$ will be inferred). For this, we selected k for each benchmark via extensive preliminary active learning runs, tuning it to avoid any missing states or transitions. Second, the number of conflicts exposed by the static mapper may depend on how its BV set is constructed, which can affect the comparison. Finally, implementation errors in our tool cannot be entirely ruled out; at least for the Simulink set, we manually confirmed that the SFAs learned by the baseline and AutoCL are semantically equivalent.

6 Related Work

6.1 Abstraction Mapper

The role of a mapper is to abstract a (possibly infinite) set of concrete alphabets into a small, finite set of abstract alphabets, which can be viewed as trace abstraction. While active automata learning frameworks such as LearnLib [40], RALib [24], and AALpy [57] support a variety of learning algorithms and automata classes, the mapper is typically implemented manually by users [22, 36, 61, 63], making abstraction design time-consuming and error-prone; when the abstraction turns out to be inadequate, users must refine it and restart the learning process. Several domain-specific approaches ease or automate this effort by leveraging domain knowledge—e.g., for web applications [8, 10, 62], mobile apps [17], and network protocols [23, 31]—by predefining domain-specific keywords or analyzing domain-specific patterns (e.g., event sequences, packets, and interaction behavior) to define abstract symbols or merge states in automata. In contrast, our approach fully automates trace abstraction and its refinement for active learning over observable variables in reactive C programs, without application-domain assumptions.

Our closest line of work is the general-purpose, automated refinement of mappers [1, 2, 39]. These techniques can be viewed as a CEGAR approach [18], where CEs trigger the refinement of the trace abstraction. These approaches refine the abstraction in a coarse-to-fine manner by progressively splitting abstract symbols (making the partition of the concrete alphabet finer) using simple (in)equalities, and are primarily tailored to learning classical automata over a finite abstract alphabet. Our approach follows the CEGAR approach in principle, but it refines a predicate-based Boolean abstraction for symbolic automata learning by synthesizing new predicates, rather than simply splitting symbols by (in)equalities. Aarts et al. [2] were the first to define a symbolic mapper for symbolic automata; however, they provided only a manual example of abstraction, leaving the automated initialization and refinement of symbolic mappers as an open problem.

6.2 Symbolic Automata Learning

The key advantage of symbolic automata over classical automata is their ability to map concrete alphabets—valuations over observable variables—into symbolic alphabets defined by predicates. Although predicates can be highly expressive in principle, existing approaches [15, 16, 25, 26, 30] typically support only simple partitioning rules (e.g., (in)equality, intervals, or a restricted fragment of linear arithmetic such as simple additions), thereby limiting expressiveness. Alternatively, users may manually supply expressive predicates, which learners can then combine logically. However, this imposes a burden on the user, who may need to provide additional predicates to ensure the correct learning of symbolic automata. This is closely related to the static mapper, which relies on a fixed set of BVs. The baseline approach of Jeppu et al. [42–44] instead employs program synthesis [5, 9, 35] to automatically infer expressive predicates directly from program variables of diverse types. However, our evaluation showed that this approach is inefficient and incurs a high computational cost. In contrast, our approach takes a two-layered approach that first

synthesizes predicates from traces over observable variables and then abstracts them into BVs, and also dynamically refines them, achieving both expressiveness and efficiency without manual effort.

6.3 Specification Mining

Automata learning can be a powerful means for automated specification mining, which has been actively investigated [6, 11, 15, 24, 25, 30, 45, 49, 52, 53]. The works in [6, 11, 43, 45, 49, 52, 53] use execution traces whose observations are API-call names or event symbols, which typically results in learning a finite-state machine (FSM) over such symbols. Specifications extracted from these FSMs capture temporal aspects of system-level behavior (e.g., API ordering constraints), but abstract away the program's internal conditions and relations over program variables. By contrast, our approach uses execution traces whose observations are valuations of observable variables and learns symbolic automata; the resulting specifications can describe functional relations among inputs, internal states, and outputs, making them more suitable for deriving system-level functional properties.

Several works [15, 24, 30, 42, 43] also take execution traces whose observations are valuations over variables and construct symbolic automata, often referred to as extended finite-state machines (EFSMs) or register automata. Since they learn automata whose transitions are guarded by predicates over variables, their extracted specifications are similar to those of our approach. However, except for the baseline [42, 43], these techniques do not provide formal correctness guarantees; most operate in a black-box setting and employ conformance testing [3, 4, 15, 34] to approximate the correctness of the learned models. By contrast, our approach theoretically guarantees the learning of sound automata, ensuring that the specifications extracted from them are likewise correct.

Other lines of work for specification mining include invariant inference [21, 27, 48, 54, 64] and LTL specification mining [51, 67]. Daikon [27] and its successors [21, 48] mine invariants from program states (i.e., valuations over variables), but these invariants are typically scoped to unit level rather than system level. More recently, AutoSpec [64] and SpecGen [54] use LLMs to propose candidate invariants that are validated and refined by a verifier; however, such approaches do not generally provide a convergence guarantee for the refinement loop and typically capture method-level behavior. In contrast, our approach aims to generate correct specifications that capture system-level behavior of the reactive program. We also note that LTL specification mining can be achieved via automata learning, since LTL specifications can be derived from the learned automata.

7 Discussion

We introduced a novel dynamic symbolic mapper for active symbolic automata learning from reactive programs. Our dynamic symbolic mapper abstracts observable variables over diverse domains into BVs, and dynamically and iteratively refines this abstraction using CEs identified by the teacher. The use of the dynamic mapper improves both efficiency and effectiveness of active learning by reducing the search space from numeric vector space to Boolean vector space and by applying predicate-to-Boolean mapping on demand, introducing only the necessary and sufficient number of BVs for each active learning iteration. This strategy is essential and unavoidable as identifying necessary and sufficient BVs up-front is very difficult, if not impossible. As shown in Table 3, the use of a static mapper often results in trace conflicts or inefficient/incomplete automata learning. Our alternating-granularity strategy, which adopts coarse-grained abstraction for teaching and fine-grained abstraction for learning, also helps improve the performance of symbolic automata learning by maintaining BVs more adaptively. The dynamic mapper with an alternating-granularity strategy is implemented in our active learner AutoCL, which outperforms the baseline active learner by successfully learning 32 more sound symbolic automata within the 3-hour time limit and by reducing the average active learning time by 55.6%.

7.1 Practical Value

7.1.1 System Understanding. Our approach learns formal behavioral models for the reactive system as a whole, at the system level. Even when the learned model is incomplete due to scalability limitations, it still provides a useful means of understanding the system's overall behavior. We can also generate multiple models from the same source code with varying choices of primary state variables, with each model highlighting a distinct aspect of the system. This enables both a focused understanding of specific components and an integrated view of complex interrelations through the composition of models. Such behavior models can serve as a starting point for reverse system documentation, supplemented by manual validation.

7.1.2 Functional Specification Mining. Among the various use cases of formal behavioral models, automated specification mining is particularly valuable for systematic software verification. As mentioned in §5.3, we generated functional specifications from the learned models in the form of transition relations and transition invariants, and employed them in model checking to confirm that the generated models are faithful representations of their corresponding reactive programs. These functional properties can further serve not only for formal verification but also as test oracles for systematic testing of target programs under evolution.

7.2 Limitations

Despite encouraging results, our approach still suffers from the issues that naturally come with the use of formal approaches such as program synthesis and model checking.

(1) *Bounded model checking*. The use of bounded model checking as a teacher may produce unsound automata if an insufficient bound k is used.
(2) *Inefficient handling of programs with many observable variables.* Our BV refinement operation requires synthesizing new BVs as predicates over the observable variables; when their number is large, this process can become costly and may degrade the efficiency.
(3) *Inefficient handling of large domains of primary state variables.* Large domains of primary state variables ($Dom(st)$) greatly deteriorate the efficiency and effectiveness of both learning and teaching due to the increase in the number of transitions to be learned and taught.

7.3 Future Work

We plan to extend our approach in a couple of directions. First, to mitigate the limitation of bounded model checking, we are investigating stronger verification back-ends for feasibility checking, such as k-induction [33] or interpolation-based reasoning [56]. Second, we plan to improve the efficiency of learning symbolic automata for programs with a large number of observable variables by pruning irrelevant variables before BV refinement in each iteration. Third, we will investigate automatic abstraction of the value domain $Dom(st)$ to enable state merging. Lastly, but most importantly, we aim to explore automated compositional learning that infers each component of reactive programs, such as a thread or a task, individually, and formally composes them at the system level.

Data Availability

Our reproduction package (the Docker image of AutoCL) is available at [46].

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2021R1A5A1021944).

References

[1] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. 2012. Automata learning through counterexample guided abstraction refinement. In *FM 2012: Formal Methods: 18th International Symposium, 2012. Proceedings 18*. Springer, 10–27. doi:10.1007/978-3-642-32759-9_4

[2] Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits Vaandrager. 2015. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design* 46, 1 (2015), 1–41. doi:10.1007/s10703-014-0216-x

[3] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Sicco Verwer. 2014. Improving active Mealy machine learning for protocol conformance testing. *Machine learning* 96, 1 (2014), 189–224. doi:10.1007/s10994-013-5405-0

[4] Bernhard K Aichernig and Martin Tappler. 2019. Efficient active automata learning via mutation testing. *Journal of Automated Reasoning* 63, 4 (2019), 1103–1134. doi:10.1007/s10817-018-9486-0

[5] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8. doi:10.1109/FMCAD.2013.6679385

[6] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16. doi:10.1145/565816.503275

[7] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106. doi:10.1016/0890-5401(87)90052-6

[8] Alexander Bainczyk, Alexander Schieweck, Malte Isberner, Tiziana Margaria, Johannes Neubauer, and Bernhard Steffen. 2016. ALEX: mixed-mode learning of web applications at ease. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 655–671. doi:10.1007/978-3-319-47169-3_51

[9] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[10] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. 2009. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 141–150. doi:10.1145/1595696.1595719

[11] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 267–277. doi:10.1145/2025113.2025151

[12] Dirk Beyer. 2024. State of the art in software verification and witness validation: SV-COMP 2024. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 299–329. doi:10.1007/978-3-031-57256-2_15

[13] Frédéric Boussinot. 1991. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience* 21, 4 (1991), 401–428. doi:10.1002/spe.4380210406

[14] Brobot. 2016. Garbage Colletor Robot Program. Retrieved September, 2025 from https://github.com/stvhwrd/Brobot/

[15] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2016. Active learning for extended finite state machines. *Formal aspects of computing* 28, 2 (2016), 233–263. doi:10.1007/s00165-016-0355-5

[16] Yu-Fang Chen, Ondřej Lengál, Tony Tan, and Zhilin Wu. 2017. Register automata with linear arithmetic. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–12. doi:10.1109/LICS.2017.8005111

[17] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices* 48, 10 (2013), 623–640. doi:10.1145/2544173.2509552

[18] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*. Springer, 154–169. doi:10.1007/10722167_15

[19] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, Berlin, Heidelberg, 168–176. doi:10.1007/978-3-540-24730-2_15

[20] Elevator controller. 2020. Elevator Controller Program. Retrieved September, 2025 from https://github.com/Sandbergo/elevator-controller/

[21] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*. 281–290. doi:10.1145/1368088.1368127

[22] Lesly-Ann Daniel, Erik Poll, and Joeri De Ruiter. 2018. Inferring OpenVPN state machines using protocol state fuzzing. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 11–19. doi:10.1109/EuroSPW.2018.00009

[23] Joeri de Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 193–206. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter
[24] Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. 2024. Scalable tree-based register automata learning. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 87–108. doi:10.1007/978-3-031-57249-4_5
[25] Samuel Drews and Loris D'Antoni. 2017. Learning symbolic automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 173–189. doi:10.1007/978-3-662-54577-5_10
[26] Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. 2019. Symbolic register automata. In *International Conference on Computer Aided Verification*. Springer, 3–21. doi:10.1007/978-3-030-25540-4_1
[27] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45. doi:10.1016/j.scico.2007.01.015
[28] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis*. doi:10.4230/OASIcs.WCET.2016.2
[29] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi
[30] Dana Fisman, Hadar Frenkel, and Sandra Zilles. 2023. Inferring symbolic automata. *Logical Methods in Computer Science* 19 (2023). doi:10.46298/LMCS-19(2:5)2023
[31] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification*. Springer, 454–471. doi:10.1007/978-3-319-41540-6_25
[32] Object follower. 2013. Object-Following Automotive Multitasking Program. Retrieved September, 2025 from https://github.com/addud/object-follower/
[33] Mikhail YR Gadelha, Hussama I Ismail, and Lucas C Cordeiro. 2017. Handling loops in bounded model checking of C programs via k-induction. *International journal on software tools for technology transfer* 19, 1 (2017), 97–114. doi:10.1007/s10009-015-0407-9
[34] Bharat Garhewal and Carlos Diego N Damasceno. 2023. An experimental evaluation of conformance testing techniques in active automata learning. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 217–227. doi:10.1109/MODELS58315.2023.00012
[35] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. doi:10.1561/2500000010
[36] Jiaxing Guo, Chunxiang Gu, Xi Chen, and Fushan Wei. 2019. Model Learning and Model Checking of IPSec Implementations for Internet of Things. *IEEE Access* 7 (2019), 171322–171332. doi:10.1109/ACCESS.2019.2956062
[37] Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamarić. 2013. Hybrid learning: interface generation through static, dynamic, and symbolic analysis. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 268–279. doi:10.1145/2483760.2483783
[38] Falk Howar, Malte Isberner, Maik Merten, Bernhard Steffen, Dirk Beyer, and Corina S Păsăreanu. 2014. Rigorous examination of reactive systems: The RERS challenges 2012 and 2013. *International Journal on Software Tools for Technology Transfer* 16, 5 (2014), 457–464. doi:10.1007/s10009-014-0337-y
[39] Falk Howar, Bernhard Steffen, and Maik Merten. 2011. Automata learning with automated alphabet abstraction refinement. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 263–277. doi:10.1007/978-3-642-18275-4_19
[40] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The Open-Source LearnLib. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I 27*. Springer, 487–495. doi:10.1007/978-3-319-21690-4_32
[41] Natasha Yogananda Jeppu. 2023. Active Learning Implementation. University of Oxford. doi:10.5287/ora-aownkwvym
[42] Natasha Yogananda Jeppu, Tom Melham, and Daniel Kroening. 2022. Active learning of abstract system models from traces using model checking. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 100–103. doi:10.23919/DATE54114.2022.9774595
[43] Natasha Yogananda Jeppu, Tom Melham, and Daniel Kroening. 2022. Enhancing active model learning with equivalence checking using simulation relations. *Formal Methods in System Design* 61, 2 (2022), 164–197. doi:10.1007/s10703-023-00433-y

[44] Natasha Yogananda Jeppu, Thomas Melham, Daniel Kroening, and John O'Leary. 2020. Learning concise models from long execution traces. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6. doi:10.1109/DAC18072.2020.9218613

[45] Hong Jin Kang and David Lo. 2021. Adversarial specification mining. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–40. doi:10.1145/3424307

[46] Yoel Kim and Yunja Choi. 2026. Reproduction Package for the FSE 2026 Article 'Active Learning of Symbolic Automata for Reactive Programs via Dynamic Symbolic Mapper'. doi:10.6084/m9.figshare.30069472.v2

[47] Jeff Kramer, Jeff Magee, Morris Sloman, and Andrew Lister. 1983. Conic: an integrated approach to distributed computer control systems. *IEE Proceedings E (Computers and Digital Techniques)* 130, 1 (1983), 1–10. doi:10.1049/ip-e.1983.0001

[48] Markus Kusano, Arijit Chattopadhyay, and Chao Wang. 2015. Dynamic generation of likely invariants for multithreaded programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 835–846. doi:10.1109/ICSE.2015.95

[49] Tien-Duy B Le and David Lo. 2018. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 106–117. doi:10.1145/3213846.3213876

[50] LeetCode. 2025. Online Programming Assignment Platform. https://leetcode.com/.

[51] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL specification mining (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 81–92. doi:10.1109/ASE.2015.71

[52] David Lo and Siau-Cheng Khoo. 2006. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 265–275. doi:10.1145/1181775.1181808

[53] Weilin Luo, Tingchen Han, Junming Qiu, Hai Wan, Jianfeng Du, Bo Peng, Guohui Xiao, and Yanan Liu. 2025. NADA: Neural Acceptance-Driven Approximate Specification Mining. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1795–1817. doi:10.1145/3728956

[54] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 16–28. doi:10.1109/ICSE55347.2025.00129

[55] MATLAB. 2025. Simulink Stateflow Examples. https://uk.mathworks.com/help/stateflow/examples.html?s_tid=CRUX_topnav.

[56] Kenneth L McMillan. 2006. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*. Springer, 123–136. doi:10.1007/11817963_14

[57] Edi Muškardin, Bernhard K Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. 2022. AALpy: an active automata learning library. *Innovations in Systems and Software Engineering* 18, 3 (2022), 417–426. doi:10.1007/s11334-022-00449-3

[58] OpenAI. 2025. ChatGPT. https://chat.openai.com/.

[59] OpenSSL. 2025. Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[60] Preeti Ranjan Panda. 2001. SystemC: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th international symposium on Systems synthesis*. 75–80. doi:10.1145/500001.500018

[61] Andrea Pferscher and Bernhard K Aichernig. 2022. Stateful black-box fuzzing of bluetooth devices using automata learning. In *NASA Formal Methods Symposium*. Springer, 373–392. doi:10.1007/978-3-031-06773-0_20

[62] Matthias Schur, Andreas Roth, and Andreas Zeller. 2013. Mining behavior models from enterprise web applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 422–432. doi:10.1145/2491411.2491426

[63] Martin Tappler, Bernhard K Aichernig, and Roderick Bloem. 2017. Model-based testing IoT communication via active automata learning. In *2017 IEEE International conference on software testing, verification and validation (ICST)*. IEEE, 276–287. doi:10.1109/ICST.2017.32

[64] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. In *International Conference on Computer Aided Verification*. Springer, 302–328. doi:10.1007/978-3-031-65630-9_16

[65] Winlift. 2020. ColHC12 OSEKturbo WindowLift demo and True-Time Simulator. Retrieved September, 2025 from https://github.com/sselab-office/HC12_C_WinLift/

[66] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2020. On Scheduling Constraint Abstraction for Multi-Threaded Program Verification. *IEEE Transactions on Software Engineering* 46, 5 (2020), 549–565. doi:10.1109/TSE.2018.2864122

[67] Changjian Zhang, Parv Kapoor, Ian Dardik, Leyi Cui, Rômulo Meira-Góes, David Garlan, and Eunsuk Kang. 2025. Constrained LTL Specification Learning from Examples. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 629–641. doi:10.1109/ICSE55347.2025.00160

Received 2025-09-12; accepted 2026-03-24