Discovering Likely Invariants for Distributed Systems through Runtime Monitoring and Learning

Yuan Xia[1], Deepayan Sur[1], Aabha Shailesh Pingle[1], Jyotirmoy V. Deshmukh[1], Mukund Raghothaman[1], and Srivatsan Ravi[1]

University of Southern California, Los Angeles, CA, USA
{yuanxia,deepayan,apingle,jdeshmuk,raghotha,srivatsr}@usc.edu

Abstract. Characterizing the set of reachable states of a distributed protocol that uses asynchronous message-passing communication is difficult due to the exponential number of possible interleavings of local executions. Any syntactic expression overapproximating the set of reachable states is an *invariant formula* of the system, and is a valuable tool that can aid programmers in understanding global program behavior. In this paper, we propose a method for obtaining a formula that approximates the set of reachable states; we call this formula a likely invariant, and we learn it using information only obtained from system executions. Our method doubles up as a way for identifying states that may not be known to be reachable (based on the best-known likely invariant) and hence may appear anomalous to the system designer. In some cases, they may be actually anomalous and may indicate a lurking (*heisenbug*). Our method has the following main steps: (1) we observe the global states of the system reached during its execution, (2) we asynchronously learn a *likely invariant* from the observed global states, (3) we monitor the learned likely invariant for the system states that do not satisfy it, and (4) if such states are found, we *revise* the likely invariant. We implement our overall methodology for a number of distributed protocols written in the Promela language and show that our technique can learn useful information about the system from just runtime executions.

1 Introduction

Distributed systems serve as the backbone of most real-world computing applications. These systems are modeled as a collection of concurrent processes that rely on local computations and asynchronous message-passing to achieve their objectives, involving multiple functions, e.g., consensus, coordination, memory coherence, decentralized computation, and database consistency. Therefore, the inherent nondeterminism can exponentially increase the number of possible execution sequences and can significantly expand the space of reachable system states, making it challenging to formally reason about system behaviors.

A Boolean-valued expression over program variables that is true for every reachable state of the program is called an *invariant formula* or simply, a program *invariant*. Learning program invariants is valuable as it enhances the user

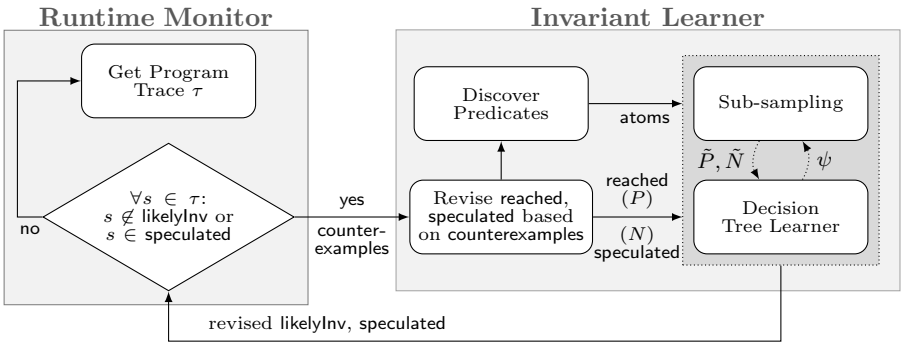

Fig. 1: High-level Algorithm 1 for learning and online monitoring likely invariants

understanding of the set of reachable states of the program, and can also be used to prove safety of the program w.r.t. a user-specified safety specification [2]. We define a *likely invariant* as a syntactic expression which holds true for all *observed* states of a program (in general, a subset of the reachable states[1]).

While many formal techniques [7, 12, 18, 19, 29, 35, 36, 42, 45] including model checking [2] have been effective for invariant learning, they usually struggle when applied to large-scale distributed systems. Some approaches require prior knowledge from the user; for instance, some invariant-learning techniques such as ICE-learning [18,19] require a user-provided safety property as a vital step in learning the invariant. Other invariant synthesis techniques make use of model checkers, SMT solvers, theorem provers, or constrained Horn clause solvers [36, 41, 47, 51] to verify if a synthesized invariant is valid. Unfortunately, this restricts the use of invariant synthesis to programs for which the above tools are tractable. These limitations motivate our use of *dynamic invariant generation*, where we approximate reachable states during execution, examples include approaches based on Daikon [11, 25]. Given a sequence of program states, Daikon uses predefined expression templates to synthesize invariant expressions derived from these templates. However, its reliance on specific syntactic expressions may result in expressions that are too conservative (i.e., with large error in overapproximating the reachable state set).

To address these challenges, this paper introduces the LIDO framework (Likely Invariant Discovery through Online monitoring) shown in Fig. 1. LIDO has two main components: a runtime monitor that continuously monitors program executions for counterexample states and an invariant learning engine. We first describe the invariant learner, which is a data-driven step: we use the observed states of the program as positive examples which must be satisfied by any candidate invariant expression. Learning such an expression usually follows the Occam's Razor principle of finding the shortest syntactic expression that explains the (positive) examples. However, as *true* is a trivial expression that is an invariant, we also need negative examples to get non-trivial expressions. To do

[1] Determining if a Boolean expression holds for every reachable state is undecidable for general programs by a reduction from the halting problem for Turing machines. Likely invariants are thus a best effort solution to approximate reachable states.

so, we use a "guess and check" approach, i.e., we guess unreachable states (by random sampling) and use them as negative examples in the decision-tree based learning method. Checking if the speculated states is part of online validation of the invariant, and incorrectly guessed unreachable states form one kind of counterexample. Existing approaches [3, 8, 19–21, 25, 48, 50, 51] use formal tools to verify or refute a synthesized invariant; however, LIDO employs online monitoring to find states that should be included in the invariant (but are not). These form the other kind of counterexample states. Any counterexample state discovered by the runtime monitor leads to a *revision* of the likely invariant.

A key challenge in the decision-tree based invariant synthesis method is identifying atomic predicates that should constitute the invariant expression. In this work, we assume that the user provides predicate templates of the form $\texttt{size(a)} > \texttt{c}_1$, $\texttt{x}_1 + \texttt{x}_2 < \texttt{c}_2$, etc. Here, a, x_1, x_2 are program variables, and c_1, c_2 are symbolic constants. LIDO discovers concrete predicates from such symbolic predicates automatically.

We provide experimental evaluation of our technique for several distributed protocols written in the Promela modeling language. Through the SPIN execution environment [23], we can generate runtime traces for Promela programs. Our technique itself is language-agnostic, and with some engineering effort can be applied to programs written in other languages such as Java or P [9]. Through our experiments on Promela, we demonstrate the potential of our technique to work on more complex programming and modeling languages. As our technique for learning the invariant is a purely data-driven technique with only runtime validation, an important question is whether the likely invariants that we learn are indeed true invariants. Here, we leverage the Spin model checker that can verify Promela programs to empirically evaluate the invariants learned by LIDO. We show that *in spite of being purely data-driven*, we can learn invariants that are valid.

Contributions. To summarize, this paper makes the following contributions:

(i) A decision-tree based method to learn invariants only from observed system states and speculated unreachable states.
(ii) An online monitoring framework to validate the learned invariant and revise the candidate invariant.
(iii) A runtime technique to discover concrete predicate expressions from user-supplied template predicates.
(iv) A novel evaluation measurement for likely invariants' quality based on three metrics: soundness, tightness, and safety.
(v) Demonstrations of the practical usefulness of our approach for distributed protocols modeled in the Promela language [22].

2 Preliminaries

In this section, we formalize the terminology needed to explain our approach with Peterson's algorithm, which presents mutual exclusive access to the critical

```
// Peterson's solution to the mutual exclusion problem - 1981
bool turn, flag[2];
byte ncrit;
active [2] proctype user() {
    assert(_pid == 0 || _pid == 1);
again:
    flag[_pid] = 1;
    turn = _pid;
    (flag[1 - _pid] == 0 || turn == 1 - _pid);
    ncrit++;
    assert(ncrit == 1); // critical section
    ncrit--;
    flag[_pid] = 0;
    goto again }
```

Fig. 2: Peterson's mutual exclusion algorithm for 2 processes in Promela

section for two concurrent processes [37] (henceforth denoted as $\mathcal{P}_{\mathsf{mutex},2}$ for brevity). We show $\mathcal{P}_{\mathsf{mutex},2}$ modeled using the Promela language [22] in Fig. 2.

2.1 Program structure and modeling assumptions

Program variables, the program state, and program execution traces are all standard notions, we define them formally so that we can precisely state the problem we wish to solve.

Definition 1 (Variables, State). *A type t refers to a finite or an infinite set. A program variable v of type t is a symbolic name that takes values from t. We use* $\texttt{type}(v)$ *to denote variable types. We use V to denote the set of program variables. A* valuation *ν maps a variable v to a specific value in* $\texttt{type}(v)$*. $\nu(V)$ denotes the tuple containing valuations of the program variables; a program state is a specific valuation of its variables during program execution.*

The following types are included: $\texttt{bool} = \{true, false\}$, $\texttt{int} = \mathbb{Z}$, $\texttt{byte} = \{0, -1, 3^2 + 1\}$, finite enumeration types which are some finite set of values, the program counter `pc` type that takes values in the set of line numbers of the program. We assume that the grammar of our modeling language provides syntactically accurate expressions made up of operators and program variables, and that the well-defined semantics of the language defines the valuation of an expression based on the variable valuations. Given a set of variables $\{v_1, \ldots, v_k\}$ and an expression $e(v_1, \ldots, v_k)$, we use $\nu(e)$ to denote the valuation of the expression when each v_i is substituted by $\nu(v_i)$.

Procedure calls and procedure-local variables are present in the majority of real-world programs; we assume that the program is modeled in a language that abstracts away such details. Real-world concurrent programs typically use either a shared memory or message-passing paradigm to perform concurrent operations of individual *threads* or *processes*[2]. We assume that the number of concurrent

[2] In most modern programming languages, processes and threads have been used to mean different abstract units of concurrent operation. For the purpose of this paper,

processes in the program, N, is known and fixed throughout program execution. Thus, our modeling language does not have statements to start a new process or terminate one. So, a concurrent program for us is a set of N processes, with each process being a sequence of atomic statements. We include statements such as assignments and conditional statements (which execute sequentially), and `goto` statements to alter the sequential control flow. An assignment statement is of the form $v_i \leftarrow e$, where e is an expression whose valuations must be in $\texttt{type}(v_i)$. The formal semantics of an assignment statement is that for all variables that do not appear on the LHS of the assignment, the valuation remains unchanged, while the valuation of v_i changes to $\nu(e)$.

Example 1. For $\mathcal{P}_{\mathsf{mutex},2}$, $V = \{\ell, \texttt{turn}, \texttt{flag}, \texttt{ncrit}\}$, where $\texttt{type}(\ell, turn, flag[i], ncrit) = (\texttt{pc} \times \texttt{pc}, \texttt{bool}, \texttt{bool} \times \texttt{bool}, \texttt{byte})$. The variable ℓ is a pair that maintains program counters for the two processes. The statements $\texttt{flag}[\texttt{\_pid}] = \texttt{1}$ and $\texttt{turn} = \texttt{\_pid}$ are assignments. The state s_0 is

$$s_0 = \begin{pmatrix} \ell_1 & \mapsto 7, \\ \ell_2 & \mapsto 7, \\ \texttt{turn} & \mapsto 0, \\ \texttt{flag}[0] & \mapsto 0, \\ \texttt{flag}[1] & \mapsto 0, \\ \texttt{ncrit} & \mapsto 0 \end{pmatrix}$$

After Process 0 ($\texttt{\_pid} = 0$) executes two assignment statements ($\texttt{flag}[0] \leftarrow 1$ and $\texttt{turn} \leftarrow 0$), the program state changes to the following:

$$s_2 = \begin{pmatrix} \ell_1 & \mapsto 9, \\ \ell_2 & \mapsto 7, \\ \texttt{turn} & \mapsto 0, \\ \texttt{flag}[0] & \mapsto 1, \\ \texttt{flag}[1] & \mapsto 0, \\ \texttt{ncrit} & \mapsto 0 \end{pmatrix}$$

The execution semantics of a program are typically explained using the notion of a labeled transition system.

Definition 2 (Labelled Transition System (LTS) for a concurrent program). *A labelled transition system is a tuple* (S, L, T, Init) *where* S *is a set of system states,* $T \subseteq S \times L \times S$ *is the transition relation,* L *is the set of program statements that serve as labels for elements in* T*, and* Init *is a set of initial program states as* $\mathsf{Init} \subseteq S$*.*

Labels identify the program statement that caused the transition, and transitions describe how a system changes from one state to another. Converting a program to its corresponding LTS is a well-defined process (see [2]); where transition is added based on the effect of the corresponding program statement on

we assume that we are using a modeling language such as Promela or P that abstracts away from these finer details.

the program variables (including the program counter). Assignment statements involve updating the program counter and the variables on the LHS. For conditional statements, a transition is added to the next state only if the valuation corresponding to the current state satisfies the condition. For `goto` statements, only the program counter is updated. Computing the LTS for a concurrent program is a bit more tricky. We typically use an interleaved model of concurrency, so from every state, we consider N next states corresponding to each of the N concurrent processes executing. We assume that each of the sequential processes involved in the computation is *deterministic*, and the only source of nondeterminism is context switches due to an external scheduler[3]. We also remark that an LTS is finite if the type of each program variable is finite, otherwise an LTS can have an infinite set of states.

Definition 3 (Reachable States). *For a program $\mathcal{P}$ as a labeled transition system, its set of reachable states* $\mathsf{Reach}(\mathcal{P}, \mathsf{Init})$ *is defined as the set of all states that can be reached from an initial state $s_0 \in S$ through the execution of the program statements. Formally,* $\mathsf{Reach}(\mathcal{P}, \mathsf{Init})$ *is the smallest set R that satisfies the following:*

1. $s_0 \in \mathsf{Init} \implies s \in R$, *and,*
2. $s \in R \wedge (s, s') \in T \implies s' \in R$.

Example 2. We remark that the state s_2 in Example 1 is reachable from s_0, since: $s_0 \xrightarrow{flag[\_pid]=1;turn=\_pid} s_2$. States where $ncrit = 2$ are unreachable from s_0.

Definition 4 (Program Trace). *A program trace σ of length $k = \mathsf{len}(\sigma)$ is a sequence of states $s_0, s_1, \ldots, s_{k-1}$, s.t., $s_0 \in \mathsf{Init}$, and for all $j \in [1, k-1]$, $(s_{j-1}, s_j) \in T$.*

Example 3. A possible trace σ of $\mathcal{P}_{\mathsf{mutex},2}$ with $\mathsf{len}(\sigma) = 5$ is shown below.

$$\left(\begin{array}{l} \underbrace{\langle 7,7,0,0,1,0\rangle}_{s_0}, \\ \underbrace{\langle 7,8,1,0,1,0\rangle}_{s_1}, \\ \underbrace{\langle 7,9,1,0,1,0\rangle}_{s_2}, \\ \underbrace{\langle 7,11,1,0,1,1\rangle}_{s_3}, \\ \underbrace{\langle 7,13,1,0,0,0\rangle}_{s_4}, \end{array}\right)$$

[3] Our techniques can also handle nondeterminism in the sequential processes. If the number of nondeterministic choices available is fixed and known *a priori*, then the procedure in Fig. 1 will converge. If the nondeterminism is unbounded then the convergence of the procedure depends on the generalizability of the learning procedure and the kind of counterexamples obtained by the testing procedure.

Remark 1. To obtain a program trace, starting from a random $s_0 \in \mathsf{Init}$, we can randomly sample a successor s_1 from all possible pairs $(s_0, s') \in T$, and repeat this procedure from each subsequent s_i. Some of the successor states for a given state s correspond to a context switch for the distributed program (as it may require a different process to execute its atomic instruction than the one that executed to reach the state s). To randomly sample the initial or successor states, we need a suitable distribution over the initial states and outgoing labelled transitions from a given state. This distribution is defined by the scheduler and assumed to be *unknown* to the program developer.

Definition 5 (Monitor). *A* monitor *is to observe a finite prefix of a program trace $\sigma = (s_0, s_1, \ldots, s_k)$, where k is the length of the prefix. The monitor evaluates whether the observed trace σ satisfies a given set of properties ϕ and yields a prediction.*

Runtime monitoring refers as monitoring the system's states during its execution and detecting whether the behaviors align with predefined specifications. Runtime monitoring is a popular approach for dynamic verification [28]. The *offline monitoring* [17] collects runtime information during the system's execution and stores it for later analysis. The analysis is performed offline once the system has completed execution. In contrast, *online monitoring* [17] refers to the simultaneous observation and analysis of system states as it is running. The concurrent monitor enables early detection and response, which is essential for critical systems requiring immediate corrective actions.

Definition 6 (Invariants). *An invariant I is a Boolean-valued formula over program variables and constants that is satisfied by every reachable program state.*

Example 4. For instance, in the Peterson's model, an invariant is $0 \leq ncrit \leq 1$. This invariant is useful to show that at most one process is in the critical section at any given time.

Remark 2 (About inductiveness of invariants). Invariants and *inductive invariants* are sometimes confused. An inductive invariant is defined as a set of states s.t., after executing any program statement from any state within this set, the resulting state also belongs to the same set. The exact set of reachable states of a program is an inductive invariant. However, obtaining a formula that accurately characterizes all reachable states can be challenging. Instead, it is often more practical to find a formula that over-approximates the reachable states to facilitate safety verification. While any formula that over-approximates the reachable states is considered an invariant, it may not be an inductive invariant. Many verification methods prefer inductive invariants because they can be verified using automated tools like SMT solvers. However, deriving inductive invariants using data-driven techniques requires is more complex. In our approach, we would require an execution engine that uses a labeled transition system representation of the given protocol to execute arbitrary transitions from states in the likely invariant. As the invariant expressions that we obtain are adequate to perform safety proofs, we defer learning inductive invariants to future work.

Algorithm 1: Pseduo-code for LIDO

input : Number of points randomly sampled ℓ
Precision threshold δ
Depth of already constructed decision tree d
Maximum depth of decision tree k
Ratio of speculative negative to positive examples α
output: likelyInv ϕ

```
1  reached ← {}, speculated ← {}, φ ← false, n ← 0
2  repeat
3      τ ← get_trace
4      ce ← {s | s ∈ τ ∩ [[¬φ]]}  // reached state in ¬φ
5      ce ← ce ∪ {s | s ∈ τ ∩ speculated}  // reached state assumed unreachable
6      reached ← reached ∪ ce  // update reached states
7      speculated ←
         (speculated\ce) ∪ randomSample(S \ ((speculated\ce) ∪ reached), ℓ)
8                  s.t. |speculated|/|reached| < α
9      if ce ≠ ∅ then
10         atoms ← predicates_discovery(reached, template_predicates)
11         φ ← Learner(atoms, reached, speculated, d, k, δ)
12 until true
13 return φ
```

Problem statement. Let $\mathcal{P}$ be a distributed program. Let $\mathsf{Reach}(\mathcal{P}, \mathsf{Init})$ be the set of reachable states of $\mathcal{P}$. Let G denote a user-defined grammar to specifies (a possibly infinite) set of Boolean-valued expressions over program variables, and let $\mathcal{L}(G)$ denote this set. The objective of this paper is to design an runtime algorithm able to learn a program likely invariant ϕ without interrupting the system execution. The invariant has the following properties:

1. Soundness of the likely invariant:

$$((s \in \mathsf{Reach}(\mathcal{P}, \mathsf{Init})) \Rightarrow (s \in \phi)) \tag{1}$$

2. Tightness of the likely invariant:

$$\phi = \underset{\phi \in \mathcal{L}(G)}{\arg\min} \; |\{s \mid s \notin \mathsf{Reach}(\mathcal{P}, \mathsf{Init}) \wedge s \in \phi\}| \tag{2}$$

3 Data-driven Invariant Generation

In this section, we present the overall algorithm for synthesizing likely invariants in an *online* learning setting in Algorithm 1. We use the notation $[\![\phi]\!]$ to represent the set of states that satisfy ϕ. Initially, the likely invariant is set to *false*, so in Line 4, the set ce contains all states encountered in the sampled trace T, which are then added to the set reached in Line 6. The next step involves sampling a set of states speculated to be unreachable, represented by the set speculated.

We sample ℓ states from the state space S (excluding those already in reached or speculated) and add them to speculated as long as the ratio $\frac{|\mathsf{speculated}|}{|\mathsf{reached}|}$ remains below a user-specified threshold α (Line 8). In the first iteration, since ce is non-empty, the algorithm invokes the Learner procedure with the sets reached (positive examples), speculated (negative examples), and hyper-parameters d, k, and δ (elaborated in the next section) (Line 11). The main loop of the algorithm (Lines 2-12) iteratively checks if the likely invariant from the previous iteration survives. This is done by sampling a new trace T (Line 3) and verifying whether T contains any states not included in ϕ (Line 4) or any previously speculated unreachable states (Line 5). If either condition is met, a revision is triggered in Line 11. As shown in Fig.1, we have a monitor and a generator to keep monitoring the system states and generate positive states P and negative states N. The states are then forwarded to Learner to further learn/revise the likelyInv.

Discovering Concrete Predicates. Before each revision, we first discover a dynamic set of concrete predicates (Line 10). The predicates_discovery function assumes a grammar (similar to the one shown in Table 1) with predicate templates. It constructs a dynamic set of concrete predicates based on the observed states of the system; essentially, it replaces parameters in the predicate templates with variable values in the set of observed states. We remark that for any invariant expression to be tight, one the concrete predicates thus obtained must be present in the final synthesized predicate, and thus it is enough to only consider as many concrete predicates as the number of observed states. Thus, though there may be a very large number of predicates associated with a given set of variables, we can restrict the set of concrete atomic predicates to a finite number.

Example 5. We now give an example run of the algorithm on $\mathcal{P}_{\mathsf{mutex},2}$. Recall that the state of $\mathcal{P}_{\mathsf{mutex},2}$ is a valuation of $(\ell_1, \ell_2, \texttt{turn}, \texttt{flag}[0], \texttt{flag}[1], \texttt{ncrit})$. The trace of valuations of these state variables from Example 3 is a possible sampled trace of $\mathcal{P}_{\mathsf{mutex},2}$ obtained by using $\mathsf{RecordStates}(\mathcal{P}_{\mathsf{mutex},2})$, reproduced here for ease of exposition:

$$\langle 7,7,0,0,1,0\rangle, \langle 7,8,1,0,1,0\rangle, \langle 7,9,1,0,1,0\rangle, \langle 7,11,1,0,1,1\rangle, \langle 7,13,1,0,0,0\rangle, \langle 7,7,0,0,0,0\rangle$$

In Line 8, we speculatively add following states to speculated:

$$\mathsf{speculated} = \{\langle 8,7,0,1,0,0\rangle, \langle 9,7,0,1,0,0\rangle\} \tag{3}$$

We then use the positive (reached) and negative (speculated) examples to learn a candidate likely invariant ϕ that over-approximates reached but has minimal overlap with speculated. Suppose we learn the likely invariant $\phi_1 \equiv \texttt{flag}[0] = 0$. We can check that all states in reached satisfy ϕ_1 and none of the states in speculated satisfy it. In the next iteration, suppose we sample the following trace next:

$$\langle 7,7,0,1,0,0\rangle, \langle 8,7,0,1,0,0\rangle, \langle \mathbf{9},\mathbf{7},\mathbf{0},\mathbf{1},\mathbf{0},\mathbf{0}\rangle, \langle 11,7,0,1,0,1\rangle, \langle 13,7,0,0,0,0\rangle, \langle 7,7,0,0,1,0\rangle$$

$$
\begin{array}{rcl}
\textit{ArrayExpr}\ (ae) & := & a \mid \textit{subset}(a) \\
\textit{ArrayArithExprs}\ (aae) & := & \textit{index}(a,i) \mid \lambda(a,f) \\
\textit{ArrayFunc}\ (f) & := & \textit{len}(a) \mid \textit{sum}(a) \mid \textit{min}(a) \mid \textit{max}(a) \\
\textit{ArithExprs}\ (e) & := & v \mid e \diamond e \mid aae \diamond aae \mid aae \diamond e \mid e \diamond aae \\
\textit{Operator}\ (\diamond) & := & + \mid - \mid \times \mid \div \mid \textit{mod} \\
\textit{Comparator}\ (\circ) & := & = \mid > \mid < \mid \leq \mid \geq \mid \neq \\
\textit{Predicate Template} & := & e \circ p
\end{array}
$$

Table 1: Grammar for predicate templates; v is an arbitrary numeric variable, a an array variable, and p a parameter. Here, i is an integer.

Clearly, the states in this trace refute the likely invariant ϕ_1 because $\texttt{flag}[0] \neq 0$, which means that these states will appear in the set ce. We note that the second state in the set speculated (shown in (3)) was speculated to be unreachable but is actually reached in the trace (shown in bold). Thus, this state also gets added to ce, and removed from speculated. With the revised speculated and reached, we can re-learn the likely invariant. □

4 Decision Tree Learning for Invariants

In this section, we describe the details of Learner as presented in Algorithms 2,3. The Learner algorithm combines decision tree learning, syntax-guided synthesis, and a novel subsampling technique to enable efficient inference of likely invariants at runtime.

Syntax-guided Synthesis. Our method for learning likely invariants is motivated by syntax-guided synthesis (SyGuS) [1]. We assume that a grammar G is provided, which specifies Boolean-valued formulas constructed from user-defined parametric atomic predicates: $\phi ::= \mathsf{atom}(\mathbf{p}) \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$. In this context, $\mathsf{atom}(\mathbf{p})$ denotes Boolean-valued expressions involving program variables and parameters, referred to as parametric atomic predicates. The parameters $\mathbf{p}$ in atom act as placeholders for constant values of corresponding types. Replacing a parametric predicate with a suitable constant results in a concrete atomic predicate. Following predicate_discovery, we obtain a set of concrete atomic predicates, represented by atoms, which correspond to a finite collection of instantiated atomic predicates. The pre-defined parametric atomic predicates are sufficient for exploring concrete atoms, as they are derived from combinations of global variables and potential grammar rules. The grammar for atomic predicates that we use for invariant synthesis is provided in Table 1.

Example 6. Consider the following grammar for parameterized atom symbols.

$$\mathsf{atom}(c) ::= (x \leq c) \mid (x \geq c) \mid (y \leq c) \mid (y \geq c)$$

Here, the program variables V is the set $\{x, y\}$ (say of type `byte`), and c is a parameter of type `byte`. Substituting c with values, e.g., 2, -3, etc., gives atomic predicates $x \leq 2$, $y \geq -3$, etc. □

Next, we present the concept of a signature for a well-formed formula in G. We assume that we have an ordered set of positive examples $P = \langle e_1, \ldots, e_{|P|} \rangle$ and an ordered set of negative examples $N = \langle f_1, \ldots, f_{|N|} \rangle$.

Definition 7 (Formula signature). *Given ordered sets P and N, the signature of a formula ϕ is a $(|P| + |N|)$-bit vector σ^ϕ, where for each $i \in [\,1, |P|\,]$, $\sigma^\phi_i = 1$ iff $e_i \models \phi$ and 0 otherwise, and for $i \in [\,|P| + 1, |P| + |N|\,]$, $\sigma^\phi_i = 1$ iff $f_{i-|P|} \models \phi$ and 0 otherwise.*

Assuming the sets P and N are concatenated, the bit at a given index in σ is 1 *iff* the corresponding example in the concatenated set satisfies ϕ. It is important to recognize that, with the formula signatures for specific atomic predicates, we can easily derive the signatures for more complex formulas in the grammar by recursively applying the following rules, where bw_op represents the bitwise application of the corresponding logical operation.

$$\sigma^{\neg\phi} = \mathsf{bw\_not}(\sigma^\phi),\ \sigma^{\phi_1 \wedge \phi_2} = \mathsf{bw\_and}(\sigma^{\phi_1}, \sigma^{\phi_2}),\ \sigma^{\phi_1 \vee \phi_2} = \mathsf{bw\_or}(\sigma^{\phi_1}, \sigma^{\phi_2})$$

Using the formula signature, we can also calculate the precision and recall of a formula. In the definitions below, we first define precision and recall, and then provide the expressions over σ. Precision and recall play a role in our invariant learning and the stopping criteria of subsampling, which will be explained in the following subsections.

$$\begin{aligned} \mathsf{precision}(\phi, P, N) &= \frac{|P \cap [\![\phi]\!]|}{|(P \cup N) \cap [\![\phi]\!]|} \\ &= \frac{\left|\left\{ i \mid i \leq |P| \wedge \sigma^\phi_i = 1 \right\}\right|}{\mathrm{sum}(\sigma^\phi)} \end{aligned} \tag{4}$$

$$\begin{aligned} \mathsf{recall}(\phi, P, N) &= \frac{|P \cap [\![\phi]\!]|}{|P|} \\ &= \frac{\left|\left\{ i \mid i \leq |P| \wedge \sigma^\phi_i = 1 \right\}\right|}{|P|} \end{aligned} \tag{5}$$

Learner is presented in two parts: a decision tree learner and a sub-sampling procedure to improve the scalability and generalizability of decision-tree learning. Previous work such as [46] frequently employs enumerative solvers for learning expressions. Although these solvers perform effectively in practice for learning expressions over more complex types, we have found that they do not scale as effectively when it comes to learning Boolean-valued functions. In contrast, the decision tree model is particularly effective to tackle the combinatorial aspect of learning Boolean combinations of atomic predicates.

Algorithm 2: DecisionTreeLearner(atoms, P, N, d, k, δ)

input :
- atoms: a set of atoms
- P : set of positive examples, N : set of negative examples
- d : depth of already constructed decision tree
- k : maximum allowed depth for the decision tree

output: likelyInv ϕ

1 $\delta \leftarrow$ MinPrecisionThreshold
2 **if** $d < k$ **then**
3 $\quad$ atom $= \underset{\text{atom} \in \text{atoms}}{\arg\max}\,(\text{IG}(\text{atom}, P, N))$
4 $\quad$ $P' \leftarrow P \cap [\![\text{atom}]\!]$, $N' \leftarrow N \cap [\![\neg \text{atom}]\!]$
5 $\quad$ atoms$'$ = atoms $\setminus$ {atom}
6 $\quad$ $\phi \leftarrow$ (atom $\wedge$ DecisionTreeLearner(atoms$'$, $P', N', d+1, k, \delta$) $\vee$
7 $\quad\quad$ ($\neg$atom $\wedge$ DecisionTreeLearner(atoms$'$, $P \setminus P', N \setminus N', d+1, k, \delta$)
8 **if** (precision$(\phi, P, N) > \delta \ \wedge$ recall$(\phi, P, N) = 1$) **then return** ϕ
9 **else** report error: k is too low

Decision Tree Learner. Algorithm 2 presents the decision tree learning technique. Each internal node in our decision tree model represents an atom with its formula signature. Given our speculative sampling technique, ensuring that our decision tree has an appropriate depth bound is critical to avoid over-fitting (if the depth is too high) or over-generalizing (if the depth is too low). The key steps are shown in Algorithm 2. The DecisionTreeLearner procedure is a recursive algorithm; each recursive instance is invoked with sets of states P and N, and it constructs a sub-tree that best partitions states in P and N. The subtree is then returned to the caller, where it is added as a child to the partial tree being constructed by the caller. The parameter d equals the depth of the decision tree constructed by the caller. Each recursive call finds the *best* atom to use as the root of the sub-tree – we explain how we define the best atom below. All examples from P that satisfy atom are removed to get P', and those from N not satisfying atom are removed to get N' (Line 4). The predicate atom is itself removed from the set atoms to get atoms$'$ (Line 5). Then DecisionTreeLearner is recursively invoked on the sets P' and N' (to form the left sub-tree) and on the sets $P \setminus P'$ and $N \setminus N'$ (to form the right sub-tree); the recursive invocations are with an incremented value of d (Line 7). The likely invariant expression is constructed using tail recursion. If the learned likely invariant lacks sufficient precision (i.e., too many negative states are included in the likely invariant), then either the maximum allowed depth k is too low, or the choice of atoms is insufficient to learn a good invariant, and the procedure fails. We remark that we use a precision threshold of less than 1 because states in N are not known to be truly unreachable, but are speculative. Therefore, it is reasonable to allow a certain number of speculative negative states to be included in the invariant.

To select the best atom in each recursive call, we choose the atom with the highest *information gain* over the sets P and N (denoted IG(atom, P, N)) in

Algorithm 3: Learner(P, N, δ)

input : P : set of positive examples
N: set of negative examples
δ : precision threshold
output: likelyInv ψ

1 $\tilde{P}, \tilde{N} = \mathsf{subSample}(P, N)$
2 **while** $\tilde{P} \neq P \vee \tilde{N} \neq N$ **do**
3 $\quad \psi = \mathsf{DecisionTreeLearner}(\tilde{P}, \tilde{N}, 0, k, \mathsf{atoms})$
4 $\quad$ **if** $\mathsf{precision}(\psi, P, N) > \delta \wedge \mathsf{recall}(\psi, P, N) = 1$ **then return** ψ
5 $\quad$ **else** $\tilde{P} = \tilde{P} \cup \mathsf{subSample}(P), \tilde{N} = \tilde{N} \cup \mathsf{subSample}(N)$
6 increase tree depth k

Line 3. This is a commonly used metric in decision tree algorithms [31]. To define information gain, we make use of Shannon entropy. Given sets P and N, consider the set $P \cup N$. The probability of a randomly drawn state being in P is $p_P = \frac{|P|}{|P \cup N|}$ and being in N is $p_N = \frac{|N|}{|P \cup N|}$. Then the Shannon entropy is defined as:

$$H(P, N) = -p_P \log_2(p_P) - p_N \log_2(p_N) \tag{6}$$

Lower entropy signifies a higher level of separation between positive and negative points in the dataset. In other words, when the dataset is more homogeneous and primarily consists of one class, the entropy will be lower. Conversely, higher entropy signifies a greater level of randomness (or uncertainty) in the dataset, characterized by a more equal distribution of class labels. In our context, this means that a more balanced set of points is being separated. Information Gain with Shannon Entropy (as proposed in [52]) is then defined as follows:

$$\begin{aligned} \mathsf{IG}(P, N, \mathsf{atom}) = H(P, N) &- p_P H(P \cap [\![\mathsf{atom}]\!], N \cap [\![\mathsf{atom}]\!]) \\ &- p_N H(P \cap [\![\neg\mathsf{atom}]\!], N \cap [\![\neg\mathsf{atom}]\!]) \end{aligned} \tag{7}$$

Subsampling for Efficiency. Now we explain how the overall Learner procedure works. In Line 1 of Algorithm 3, we use subSample to sample states from the positive examples P and negative examples N. Recall that Learner is invoked with $P = \mathsf{reached}$ and $N = \mathsf{speculated}$. The main idea is to only use the subsets $\tilde{P}$, $\tilde{N}$ of P, N to invoke DecisionTreeLearner (Line 3), which returns ψ. Once we identify a candidate ψ, we check if its recall is 1 (i.e., all reached states are included in the likely invariant) and if its precision is above a pre-defined threshold δ. If yes, the algorithm returns ψ as the candidate likelyInv (Line 4). If it fails to find a likely invariant with desired precision/recall, subSample gathers more samples and invokes DecisionTreeLearner (Line 5).

Grammar. The figure illustrates the template-free grammar $\mathcal{G}$ designed for atomic predicates to express invariants. This grammar provides an extensive range of constructs for logical and arithmetic expressions, array manipulations, and comparison operations. The grammar is also compatible with standard SMT

solvers, enabling users to verify invariants when necessary. By limiting expressiveness to linear integer arithmetic and simple array manipulations, the grammar **G** ensures that invariants can be synthesized within a feasible computational framework while still capturing the essential properties needed by programmers for practical use.

5 Experimental Evaluation

To demonstrate the feasibility of our framework and the quality of the learned likely invariants, we evaluated LIDO for distributed protocols, which are modeled in the Promela language. Although our framework can accommodate systems in other languages such as P [9], TLA+ [34], and others, we selected Promela for two main reasons: firstly, numerous descriptions of distributed protocols are readily accessible in Promela. Secondly, Promela programs are compatible with the Spin model checker [23] its capability to sample finite-length execution traces of programs. As a comparative baseline, we employed Daikon [11] a widely used dynamic invariant generation tool that is known for its ability to identify likely invariants based on runtime traces. Daikon can serve as a front-end invariant synthesis in the LIDO framework as well. In our experiments, Daikon is used to substitute Learner, while the operational framework LIDO remained unchanged.

Python implementation. The implementation of the LIDO framework is with two concurrent Python processes, one to continuously monitor system states in real-time using the trace generation functionality of the Spin model checker, while the other to asynchronously synthesize likely invariants based on the monitored program traces. The two processes: the Runtime Monitor process monitors global program states. It does so by collecting a trace of the global system state of a fixed length. It then checks if any state in the trace violates the invariant or invalidates the speculated negative examples. If true, then the likely invariant needs to be revised, and the process writes the counterexample state(s) on a shared channel. The Invariant Learner process subscribes to this channel, and whenever a new counterexample is published to this channel, it invokes the invariant learning procedure. This process terminates when a likely invariant is synthesized, and writes the new likely invariant and the updated set of speculated unreachable states to the shared channel.

In Python, the ready method is used to determine if a subsequent read on the shared communication channel would block; we use this to check if the invariant is synthesized before issuing a get on the shared channel to obtain the updated likely invariant and the set of speculated unreachable states. Both processes are guaranteed to run concurrently in two separate Python interpreters, thus avoiding potential performance degradation caused by Python's Global Interpreter Lock (GIL). The Runtime Monitor process is not expected to stop; however, the Invariant Learner process does terminte

5.1 Benchmarks and Measurements

To investigate our research goals, we formulated three key questions:

- **RQ1**: Is the previous leading method, Daikon, capable of producing high-quality likely invariants during the runtime monitoring of real-world distributed systems?
- **RQ2**: How does LIDO compare to Daikon in terms of the quality of likely invariants generated during online monitoring of distributed systems?
- **RQ3**: Is LIDO able to effectively learn likely invariants in large-scale distributed systems at runtime with minimal overhead?

We assess the quality of a learned likely invariant using three distinct metrics. The first metric measures the tightness of the invariant by counting the total number of states that satisfy it. This can be estimated through model counting, which helps determine if the invariant is overly broad. A maximum bound can also provide insights into the total number of states, especially when the invariant implies an infinite number.

The second and third metrics focus on the ability of a posteriori verification tool to validate that the synthesized invariant aligns with a user-defined safety property. The soundness of an invariant reflects its accuracy in representing the distribution of reachable states without being overly aggressive. However, soundness alone does not guarantee utility; for example, a broadly defined invariant like $True$ may still be considered sound. Furthermore, the safety aspect of an invariant serves as an important indicator of its practical value, as it suggests that the system being analyzed remains safe.

Our benchmarks include 13 distributed systems modeled in Promela, drawn from resources related to Spin and other distributed systems literature. These systems primarily involve conditional invariants that require more intricate combinations of atomic expressions than those typically available in Daikon's templates. Notably, our approach to invariant generation successfully identified a bug in one of Spin's official systems. Additionally, we effectively generated invariants for a large-scale, real-world smart contract system modeled in Spin [49], which were subsequently verified to ensure the system's safety properties were maintained.

5.2 Results for Quality Evaluation

In order to address research questions **RQ1** and **RQ2**, we assessed the performance of LIDO through the Learner procedure, which incorporates both positive and negative examples during the learning process. Daikon* was employed as our baseline, integrating it into our monitoring framework as the inference engine to derive invariant expressions. For the ablation study, we ensured that both Daikon* and LIDO generated invariants within the same execution time, simulating conditions from Spin. The likely invariants produced by both methods were subsequently validated using the Spin model checker. The safety of these invariants was confirmed through an SMT solver, which checked if the intersection of the invariant ϕ with a user-defined set of unsafe states was empty. To maintain a consistent comparison, both Daikon* and LIDO utilized identical execution traces from Spin. This allowed for a fair evaluation of their respective performances and effectiveness.

Table 2 presents the results of our quality assessment for the learned invariant. In all cases, the likely invariants generated by LIDO with Learner were confirmed as true invariants. This trend was similarly observed with Daikon, which is known for producing overly conservative invariants. However, due to speculative synthesis, LIDO achieved precise invariants than those generated by Daikon, while maintaining a high level of soundness by monitoring system states. Notably, in each case study, the synthesized likely invariant also aligned with the system's safety property, as indicated by ✓in Table 2. In contrast, the likely invariants derived from the combination of LIDO and Daikon were only able to confirm system safety in one case study and failed in others (marked by ✗), further highlighting Daikon's tendency to generate overly conservative invariant expressions. These empirical findings suggest that, with sufficient monitoring time, LIDO can develop a robust understanding of system behavior.

We also modeled a smart contract for the Ethereum commodity market in Spin to simulate execution and verify compliance with specifications. This model plays a crucial role in enhancing the credibility of smart contracts by detecting vulnerabilities, as errors can lead to significant financial losses due to their immutable nature once deployed on the blockchain. Unlike static analysis tools such as OYENTE, Osiris, and Gasper, which focus on pre-execution analysis, our approach with LIDO allows for dynamic monitoring and synthesis of system behavior in real-time. For example, we identified a likely invariant $fa\_Acc + su\_Acc + t\_Acc + sca\_Acc + scb\_Acc = invar0$, which pertains to the total account balance and is recognized as a key safety property.

Additionally, LIDO discovered the likely invariant $((seen \leq n \wedge tour \leq max) \vee (tour > max \wedge seen \leq n))$ for the *salesman* system [23], which was verified by Spin. However, the official specification stated $seen < n \vee tour > max$, which Spin did not confirm as a true invariant. This discrepancy revealed a flaw in the official specification. By adjusting the assertion to $seen \leq n \vee tour > max$, the system was successfully verified by Spin. Thus, through our tool LIDO, we not only identified this bug but also provided a resolution.

5.3 Results for Evaluating the Scalabiilty and Efficiency

Learning likely invariants efficiently during runtime is critical for large-scale distributed systems, which often operate continuously and cannot afford to be paused for traditional offline verification methods. Given the complexity of nondeterminism inherent in distributed systems, the key question is whether LIDO can effectively handle large-scale environments while minimizing its impact on system performance. **RQ3** explores whether LIDO can balance accuracy and efficiency with minimal disruption to the system while providing meaningful invariants. To benchmark the *efficiency* of LIDO, we measured the overhead of tracking state change log information on Spin, which is the only interruption event on Spin systems. We quantify the performance impact, or overhead incurred by monitoring system of 10000 states. The overhead is defined in terms of resource usage—such as CPU, memory, and I/O operations—that the monitoring process

Table 2: **Likely Invariant Quality Evaluation**

System	Tightness		Soundness		Safety	
	LIDO	Daikon*	LIDO	Daikon*	LIDO	Daikon*
Peterson(Binary processes) [23]	✓	✗	✓	✓	✓	✗
Peterson(N processes) [23]	✓	✗	✓	✓	✓	✗
Bakery [23]	✓	✗	✓	✓	✓	✗
Hajek [23]	✓	✗	✓	✓	✓	✗
Manna Pnuelli [23]	✓	✗	✓	✓	✓	✗
Traffic Lights [13]	✓	✗	✓	✓	✓	✗
Producer Consumer [4]	✓	✗	✓	✓	✓	✗
Alternative Bit Protocol [23]	✓	✗	✓	✓	✓	✗
Leader Election [23]	✓	✗	✓	✓	✓	✓
UPPAAL Train/Gate [23]	✓	✗	✓	✓	✓	✗
Salesman [23]	✓	✗	✓	✓	✓	✗
Distributed Lock Server [30]	✓	✗	✓	✓	✓	✗
Smart Contract(ETH) [49]	✓	✗	✓	✓	✓	✗

consumes while recording and storing state information. By evaluating this overhead, we seek to determine whether the monitoring system can effectively scale to larger state spaces without introducing significant performance degradation. Based on the results, the overhead ranges between 0.008 to 0.5 seconds, which indicates that the monitoring system can efficiently handle detailed logging and state observation without significantly affecting the system's overall performance or responsiveness. The outcomes for each distributed system are outlined in Table 3. With an execution time (T_r) of at most or around $1s$, and an approximate ratio of observed to reachable states $\frac{|V|}{|R|} \approx 0$ on large-sacle systems, our tool scales effectively. Notably, even with the significantly low ratio, our tool can still infer likelyInv. The data presented in Table 3 affirm the validity of **RQ3**. It is important to note that while we were able to obtain likely invariants that were sound invariants, LIDO does not guarantee soundness as we *do not use* a model checker in the learning process. This approach aligns with traditional dynamic techniques, such as Daikon, avoiding the use of model checking. Model checking typically encounters the common issues of reduced generality and scalability.

6 Related Work and Conclusions

Runtime Monitoring. Runtime monitoring that leverages invariant inference techniques supports various applications, from software verification to error detection and enhancing system reliability. One of the pioneering and impactful tools in this domain is Daikon, which dynamically identifies likely invariants from program execution traces. Daikon has been extensively utilized for debugging and improving testing and verification processes by generating invariants that describe system behavior [5, 48].While Daikon has been integrated into a

Table 3: **Evaluation results on large-scale distributed systems** Overhead: the interruption time of system execution due to the monitoring event; T_r: average execution time for each revision.

Distributed Program	LoC	Shared Vars.	No. of Reachable States (R)	$\frac{\lvert\text{Visited}\rvert}{\lvert R\rvert}$	Overhead(s) (log info) /10000 states	$T_r(s)$
Peterson(Binary Processes) [23]	20	3	16	1	0.111	0.012
Bakery [23]	24	2	8	1	0.125	0.084
Manna Pnueli [23]	29	3	18	0.61	0.092	0.015
Hajek [23]	68	2	256	0.13	0.115	0.015
Traffic Lights [13]	33	2	200	0.065	0.178	0.004
Producer Consumer [4]	37	4	1.03M	0.00044	0.135	0.049
Peterson(N Processes) [23]	45	3	19.5M	0.00018	0.088	0.070
Alternative Bit Protocol [23]	42	3	∞	≈ 0	0.154	0.013
Leader Election [23]	127	3	26K	0.0024	0.008	0.015
UPPAAL train/gate [23]	78	7	16.8M	0.000024	0.192	0.021
Salesman [23]	54	6	∞	≈ 0	0.012	0.033
Distributed Lock Server [30]	100	4	12.2K	0.015	0.503	0.024
Smart Contract(ETH) [49]	962	21	∞	≈ 0	0.172	0.022

runtime monitoring framework, as discussed in previous work [5], our research indicates that the invariants produced by Daikon are often overly conservative.

Alongside Daikon, DIDUCE [20] also significantly influenced the field by employing template enumeration and checking invariants by mapping state information to bits, where unchanged bits during execution are treated as invariants. Building on these foundational efforts, Artemis [16] was introduced as an acceleration technique for dynamic monitoring, specifically applied to DIDUCE in C programs, enhancing its efficiency. Recent studies have also focused on inferring FSM (Finite State Machine) models [10, 40] to develop effective methodologies and frameworks for evaluating specification miners, while ours focuses on logic expressions. [6] Another study developed an SVM-based model through runtime monitoring in cyber-physical systems. Despite the contributions of these early approaches, they often face challenges related to the quality and generality of the invariants produced. Our work aims to address these issues, improving both the generality and quality of dynamic invariant generation.

Dynamic Invariant Synthesis. The main advantages of these works are scalability and generality, while the main limitation is its sensitivity to the initial pool of templates and its inability to learn interesting and non-trivial invariants, including properties with disjunction [11]. A few invariant generators [25, 26] build on Daikon. ContExt [26] combines static analysis of program properties and dynamic analysis by Daikon to generate disjunctive constraints. The work in [25] proposes an LLVM-based code instrumentation frontend on top of Daikon to achieve invariant inference on multithreaded programs. Some solvers focus on specific invariant types, such as LinearAirbitrary [52] for linear inequalities

and algebraic equation invariants [43]. Other dynamic generation tools include DIDUCE [20], DySy [8], Agitator [3], and Iodine [21], but they are often limited to certain languages, programs, or invariant types. Existing dynamic generation tools compromise applicability across diverse scenarios, lagging behind static techniques. Our tool aims to retain generality like Daikon while producing higher-quality invariants through a data-driven method, complementing existing approaches by enhancing their strengths and addressing limitations.

Model Checker Assisted Invariant Synthesis. Early attempts at automatic invariant generation used first-order theorem provers like Vampire [24,38], limited by their scalability. Subsequent work includes approaches guaranteeing provably correct invariants and data-driven methods for *likely invariants*. Counter-example guided invariant generation (CEGIR) combines inductive synthesis and model checker verification, exemplified by ICE [18], ICE-DT [19], and FreqHorn [14,15]. DistAI [51] and DuoAI [50] observe system executions but are guided by a target property verified with IVy [32], potentially causing infinite loops when the property does not hold. Techniques using theorem provers and model checkers can be accelerated through additional verifier information [33] or machine learning, such as Code2Inv [44], counter-example guided neural synthesis [39], and ACHAR [27]. However, the cost of model checking and the rapid growth of the space of possible invariants limit the complexity of systems to which these techniques can be applied. On the other hand, our approach in LIDO can thus be freely applied to complex systems.

7 Conclusion

Fundamental research in runtime monitoring has paved the way for innovations in dynamic invariant generation and automated program verification. Our approach, LIDO, is an automatic and practical framework for learning likely invariants for distributed protocols. We utilize counterexamples and speculative negative states to guide the invariant learning process. We also dynamically discover new atomic predicates from the observed states; essentially replacing the parameters in the chosen grammar by values matching the observed states.

Our framework successfully learns likely invariants, preserving the with same level of scalability as widely used tools such as Daikon, without being restricted to any particular programming language, without relying on exhaustive tools such as model checkers, or necessitating prior knowledge of the system's safety properties. Our method can provide three valuable outcomes for system developers: (1) a true overapproximation of the reachable states, which yields a valid invariant learned purely from program traces and validated only through online monitor- ing, (2) a summary of the most commonly observed states, where any violation highlights rarely encountered system behaviors or anomalies, and (3) potential verification of system safety. In future work, we will explore replacing hte decision tree learning method with more advanced generative learning models.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. 2013 Formal Methods in Computer-Aided Design pp. 1–8 (2013), https://api.semanticscholar.org/CorpusID:6705760
2. Baier, C., Katoen, J.P.: Principles of Model Checking (Representation and Mind Series). The MIT Press (2008)
3. Boshernitsan, M., Doong, R., Savoia, A.: From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis. p. 169–180. Association for Computing Machinery (2006). https://doi.org/10.1145/1146238.1146258
4. Byrd, G., Flynn, M.: Producer-consumer communication in distributed shared memory multiprocessors. Proceedings of the IEEE **87**(3), 456–466 (1999). https://doi.org/10.1109/5.747866
5. Chen, Y., Ying, M., Liu, D., Alim, A., Chen, F., Chen, M.H.: Effective online software anomaly detection. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 136–146. ISSTA 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3092703.3092730
6. Chen, Y., Poskitt, C.M., Sun, J.: Learning from Mutants: Using Code Mutation to Learn and Monitor Invariants of a Cyber-Physical System . In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 648–660. IEEE Computer Society, Los Alamitos, CA, USA (May 2018). https://doi.org/10.1109/SP.2018.00016
7. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (Sep 2003), https://doi.org/10.1145/876638.876643
8. Csallner, C., Tillmann, N., Smaragdakis, Y.: DySy: dynamic symbolic execution for invariant inference. In: Proceedings of the 30th International Conference on Software Engineering. p. 281–290. ICSE '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1368088.1368127
9. Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., Zufferey, D.: P: safe asynchronous event-driven programming. ACM SIGPLAN Notices **48**(6), 321–332 (2013)
10. Dianlin, W., Ziying, D., Donghong, L., Xiaoguang, M.: Automatic online specification mining. In: Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE). pp. 253–258 (2011). https://doi.org/10.1109/TMEE.2011.6199191
11. Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**, 35–45 (12 2007). https://doi.org/10.1016/j.scico.2007.01.015
12. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-Ice learning for synthesizing invariants and contracts. Proc. ACM Program. Lang. **2**(OOPSLA) (Oct 2018). https://doi.org/10.1145/3276501
13. Faye, S., Chaudet, C., Demeure, I.: A distributed algorithm for adaptive traffic lights control. In: 2012 15th International IEEE Conference on Intelligent Transportation Systems (2012)
14. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences

on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24. pp. 251–269. Springer (2018)
15. Fedyukovich, G., Kaufman, S.J., Bodík, R.: Learning inductive invariants by sampling from frequency distributions. Form. Methods Syst. Des. **56**(1–3), 154–177 (Dec 2020). https://doi.org/10.1007/s10703-020-00349-x
16. Fei, L., Midkiff, S.P.: Artemis: practical runtime monitoring of applications for execution anomalies. SIGPLAN Not. **41**(6), 84–95 (Jun 2006). https://doi.org/10.1145/1133255.1133992
17. Gao, L., Lu, M., Li, L., Pan, C.: A survey of software runtime monitoring. In: 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS). pp. 308–313 (2017). https://doi.org/10.1109/ICSESS.2017.8342921
18. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: a robust framework for learning invariants. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 69–87. Springer International Publishing, Cham (2014)
19. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. SIGPLAN Not. **51**(1), 499–512 (2016). https://doi.org/10.1145/2914770.2837664
20. Hangal, S., Lam, M.: Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. pp. 291–301 (2002). https://doi.org/10.1145/581376.581377
21. Hangal, S., Narayanan, S., Chandra, N., Chakravorty, S.: Iodine: a tool to automatically infer dynamic invariants for hardware designs. In: Proceedings. 42nd Design Automation Conference, 2005. pp. 775–778 (2005). https://doi.org/10.1109/DAC.2005.193920
22. Holzmann, G.J., Lieberman, W.S.: Design and validation of computer protocols, vol. 512. Prentice hall Englewood Cliffs (1991)
23. Holzmann, G.: The model checker spin. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
24. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. pp. 1–35. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
25. Kusano, M., Chattopadhyay, A., Wang, C.: Dynamic generation of likely invariants for multithreaded programs. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, pp. 835–846 (2015). https://doi.org/10.1109/ICSE.2015.95
26. Kuzmina, N., Paul, J., Gamboa, R., Caldwell, J.: Extending dynamic constraint detection with disjunctive constraints. In: Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008). p. 57–63. WODA '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1401827.1401839
27. Lahiri, S., Roy, S.: Almost correct invariants: synthesizing inductive invariants by fuzzing proofs. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 352–364. ISSTA 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3533767.3534381
28. Leucker, M., Schallhart, C.: A brief account of runtime verification. The journal of logic and algebraic programming **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
29. Li, J., Sun, J., Li, L., Le, Q.L., Lin, S.W.: Automatic loop-invariant generation and refinement through selective sampling. In: Proceedings of the 32nd IEEE/ACM

International Conference on Automated Software Engineering. p. 782–792. ASE '17, IEEE Press (2017)
30. Ma, H., Goel, A., Jeannin, J.B., Kapritsos, M., Kasikci, B., Sakallah, K.A.: I4: incremental inference of inductive invariants for verification of distributed protocols. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. pp. 370–384 (2019)
31. Maimon, O.Z., Rokach, L.: Data mining with decision trees: theory and applications, vol. 81. WORLD SCIENTIFIC (2014). https://doi.org/10.1142/9097
32. McMillan, K.L., Padon, O.: Ivy: A multi-modal verification tool for distributed algorithms. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32. pp. 190–202. Springer (2020)
33. Neider, D., Parthasarathy, M., Saha, S., Garg, P., Park, D.: A learning-based approach to synthesizing invariants for incomplete verification engines. Journal of Automated Reasoning **64** (10 2020). https://doi.org/10.1007/s10817-020-09570-z
34. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**(4), 66–73 (Mar 2015). https://doi.org/10.1145/2699417
35. Padhi, S., Sharma, R., Millstein, T.: Data-driven precondition inference with learned features p. 42–56 (2016). https://doi.org/10.1145/2908080.2908099
36. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 614–630 (2016)
37. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**, 115–116 (1981), https://api.semanticscholar.org/CorpusID:45492619
38. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS. Lecture Notes in Computer Science, vol. 2031, pp. 82–97. Springer (2001), http://dblp.uni-trier.de/db/conf/tacas/tacas2001.htmlPnueliRZ01
39. Polgreen, E., Abboud, R., Kroening, D.: Counterexample guided neural synthesis. ArXiv (2020), https://api.semanticscholar.org/CorpusID:210920369
40. Pradel, M., Bichsel, P., Gross, T.R.: A framework for the evaluation of specification miners based on finite state machines. In: 2010 IEEE International Conference on Software Maintenance. pp. 1–10. IEEE (2010), https://api.semanticscholar.org/CorpusID:6673177
41. Riley, D., Fedyukovich, G.: Multi-phase invariant synthesis. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 607–619 (2022)
42. Ryan, G., Wong, J., Yao, J., Gu, R., Jana, S.S.: Cln2inv: Learning loop invariants with continuous logic networks. ArXiv (2019), https://api.semanticscholar.org/CorpusID:202749930
43. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22. pp. 574–592. Springer (2013)
44. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Neural Information Processing Systems (2018), https://api.semanticscholar.org/CorpusID:53319040

45. Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2Inv: A deep learning framework for program verification. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II. p. 151–164. Springer-Verlag, Berlin, Heidelberg (2020), https://doi.org/10.1007/978-3-030-53291-8_9
46. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M., Alur, R.: Transit: specifying protocols with concolic snippets. ACM SIGPLAN Notices **48**(6), 287–296 (2013)
47. Wang, J., Wang, C.: Learning to synthesize relational invariants. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–12 (2022)
48. Wang Bo, Lu Sirui, J.J.X.Y.: Survey of dynamic analysis based program invariant synthesis techniques. Journal of Software **31**(6), 1681–1702 (2020)
49. Yang, Z., Dai, M., Guo, J.: Formal modeling and verification of smart contracts with spin. Electronics p. 3091 (2022). https://doi.org/10.3390/electronics11193091
50. Yao, J., Tao, R., Gu, R., Nieh, J.: {DuoAI}: Fast, automated inference of inductive invariants for verifying distributed protocols. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). pp. 485–501 (2022)
51. Yao, J., Tao, R., Gu, R., Nieh, J., Jana, S., Ryan, G.: DistAI: Data-Driven automated invariant learning for distributed protocols. In: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). pp. 405–421. USENIX Association (Jul 2021), https://www.usenix.org/conference/osdi21/presentation/yao
52. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. SIGPLAN Not. **53**(4), 707–721 (Jun 2018). https://doi.org/10.1145/3296979.3192416